

Parametric Technology Corporation

**Pro/ENGINEER® Wildfire® 5.0
Pro/TOOLKIT® User's Guide**

July 2010

Copyright © 2010 Parametric Technology Corporation and/or Its Subsidiary Companies. All Rights Reserved.

User and training guides and related documentation from Parametric Technology Corporation and its subsidiary companies (collectively "PTC") are subject to the copyright laws of the United States and other countries and are provided under a license agreement that restricts copying, disclosure, and use of such documentation. PTC hereby grants to the licensed software user the right to make copies in printed form of this documentation if provided on software media, but only for internal/personal use and in accordance with the license agreement under which the applicable software is licensed. Any copy made shall include the PTC copyright notice and any other proprietary notice provided by PTC. Training materials may not be copied without the express written consent of PTC. This documentation may not be disclosed, transferred, modified, or reduced to any form, including electronic media, or transmitted or made publicly available by any means without the prior written consent of PTC and no authorization is granted to make copies for such purposes.

Information described herein is furnished for general information only, is subject to change without notice, and should not be construed as a warranty or commitment by PTC. PTC assumes no responsibility or liability for any errors or inaccuracies that may appear in this document.

The software described in this document is provided under written license agreement, contains valuable trade secrets and proprietary information, and is protected by the copyright laws of the United States and other countries. It may not be copied or distributed in any form or medium, disclosed to third parties, or used in any manner not provided for in the software licenses agreement except with written prior approval from PTC.

UNAUTHORIZED USE OF SOFTWARE OR ITS DOCUMENTATION CAN RESULT IN CIVIL DAMAGES AND CRIMINAL PROSECUTION. PTC regards software piracy as the crime it is, and we view offenders accordingly. We do not tolerate the piracy of PTC software products, and we pursue (both civilly and criminally) those who do so using all legal means available, including public and private surveillance resources. As part of these efforts, PTC uses data monitoring and scouring technologies to obtain and transmit data on users of illegal copies of our software. This data collection is not performed on users of legally licensed software from PTC and its authorized distributors. If you are using an illegal copy of our software and do not consent to the collection and transmission of such data (including to the United States), cease using the illegal version, and contact PTC to obtain a legally licensed copy.

Important Copyright, Trademark, Patent, Licensing, and Data Collection Information:

- For Windchill products, select About Windchill at the bottom of the product page.
- For CADD5, click the "i" button on the main menu.
- For InterComm products, on the Help main page, click the link for **Copyright**.
- For other products, click **Help > About** on the main menu of the product.
- For products with an Application button, click the button and then navigate to the product information.

UNITED STATES GOVERNMENT RESTRICTED RIGHTS LEGEND

This document and the software described herein are Commercial Computer Documentation and Software, pursuant to FAR 12.212(a)-(b) (OCT'95) or DFARS 227.7202-1(a) and 227.7202-3(a) (JUN'95), and are provided to the US Government under a limited commercial license only. For procurements predating the above clauses, use, duplication, or disclosure by the Government is subject to the restrictions set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 (OCT'88) or Commercial Computer Software-Restricted Rights at FAR 52.227-19(c)(1)-(2) (JUN'87), as applicable. 04012010

Parametric Technology Corporation, 140 Kendrick Street, Needham, MA 02494 USA

Contents

About This Guide

Purpose.....	ii
Audience and Prerequisite Experience	ii
Contents.....	ii
Documentation	iii
Conventions.....	iii

Chapter 1: Fundamentals

Introduction to Pro/TOOLKIT	1-2
Online Documentation — Pro/TOOLKIT APIWizard	1-2
To Install the APIWizard	1-3
To Run the APIWizard.....	1-3
Web Browser Environments	1-3
Troubleshooting	1-5
Automatic Index Tree Updating	1-5
The APIWizard Interface	1-6
Topic/Object/Category Selection Frame.....	1-6
Display Frame	1-6
Navigating the Topic/Object/Category Selection Tree.....	1-7
APIWizard Search Feature (Find).....	1-13
Search Dialog.....	1-13
Supported Search Types.....	1-15
Performing an APIWizard Search.....	1-16
Pro/TOOLKIT Style	1-17
Objects and Actions.....	1-17
Function Prototyping.....	1-19

Function Error Statuses.....	1-19
Installing Pro/TOOLKIT.....	1-20
Overview	1-20
Add or Update Pro/TOOLKIT Installation	1-23
Testing the Pro/TOOLKIT Installation	1-23
Building a Sample Application.....	1-26
Developing a Pro/TOOLKIT Application	1-30
How Pro/TOOLKIT Works.....	1-30
Compiling and Linking a Pro/TOOLKIT Application	1-32
Registering a Pro/TOOLKIT Application.....	1-33
Version Compatibility: Pro/ENGINEER and Pro/TOOLKIT	1-35
Application Compatibility: Pro/ENGINEER and Pro/TOOLKIT on Different Architecture.....	1-36
Stopping and Restarting a Pro/TOOLKIT Application	1-37
Structure of a Pro/TOOLKIT Application	1-41
User-Supplied Main	1-48
Asynchronous Mode	1-49
Pro/TOOLKIT Techniques	1-49
Object Handles.....	1-49
Expandable Arrays	1-53
Visit Functions	1-59
Wide Strings	1-64
Defining wchar_t.....	1-64
Setting the Hardware Type.....	1-65
Checking Your Declaration of wchar_t.....	1-65
String and Widestring Functions.....	1-66

Chapter 2: Core: Models and Model Items

Modes	2-2
Models	2-2
The ProMdl Object	2-2
Creating Models	2-3
Identifying Models	2-4
Models in Session	2-6
File Management Operations	2-7
Model Items	2-8
Version Stamps	2-12
Layers	2-13
Layouts	2-20
Visiting Displayed Entities.....	2-20

Chapter 3: Core: Solids, Parts, and Materials

Solid Objects	3-2
Creating a Solid	3-2
Contents of a Solid	3-2
Displaying a Solid	3-4
Regenerating a Solid	3-5
Evaluating Mathematical Expressions for a Solid.....	3-8
Solid Outline	3-8
Solid Accuracy	3-10
Solid Units	3-11
Mass Properties.....	3-18
Solid Postfix Identifiers	3-18
Part Objects	3-19
Density.....	3-19
Material Objects	3-21
Accessing Material Data	3-22
Material Types and Properties.....	3-23
Material Input and Output	3-27

Chapter 4: Core: Features

Feature Objects.....	4-2
Visiting Features	4-2
Feature Inquiry	4-3
Feature Geometry.....	4-6
Manipulating Features.....	4-7
Manipulating Features based on Regeneration Flags	4-12
Feature Dimensions	4-13
Manipulating Patterns	4-13
Patterns as Features	4-13
Pro_Pattern Structure	4-15
Pro_pattern_dir Structure	4-16
Pro_pattern_dim Structure	4-17
Table-Driven Patterns.....	4-18
Creating Local Groups	4-19
Read Access to Groups	4-20
Finding Groups	4-21
Group Information.....	4-22
Creating Groups	4-22
Deleting Groups.....	4-23
Updating or Replacing UDFs	4-23
Placing UDFs	4-24

The UDF Input Data Structure—ProUdfdata	4-26
Variable Parameters and Annotations.....	4-28
Variable Dimensions and Pattern Parameters	4-29
UDF References.....	4-30
Assembly Intersections	4-31
External Symbol: Parameters.....	4-32
External Symbol: Dimensions	4-32
Copied Model Names.....	4-33
Reading UDF Properties.....	4-34
Variable Dimensions	4-34
Variable Parameters.....	4-35
UDF References.....	4-36
External Symbols	4-36
Instance Names	4-37
Notification on UDF Library Creation	4-37

Chapter 5: Core: 3D Geometry

Geometry Objects.....	5-2
Visiting Geometry Objects	5-3
Visiting Feature Geometry.....	5-4
Visiting Solid Geometry	5-6
Tessellation.....	5-15
Curve and Edge Tessellation	5-15
Surface Tessellation.....	5-16
Part and Assembly Tessellation	5-17
Evaluating Geometry	5-18
Evaluating Surfaces, Edges, and Curves.....	5-18
Inverse Evaluation and Minimum Distances	5-19
Geometry at Points.....	5-20
Geometry Equations.....	5-21
Geometry of Solid Edges	5-22
Geometry of Surfaces	5-25
Geometry of Axes.....	5-27
Geometry of Coordinate System Datums.....	5-27
Geometry of Datum Planes.....	5-28
Geometry of Quilts	5-28
Geometry of Datum Surfaces.....	5-29
Geometry of Datum Points.....	5-29
Geometry of Datum Curves.....	5-29
Geometry of Composite Curves	5-29
Ray Tracing	5-30

Editing Datum Points.....	5-31
Measurement	5-32
Geometry as NURBS	5-34
Interference	5-35
Faceted Geometry	5-42
Visiting Facets and Facet Sets	5-43
Accessing Facet Properties	5-43
Chapter 6: Core: Relations	
Relations	6-2
Adding a Customized Function to the Relations Dialog in Pro/ENGINEER.....	6-4
Chapter 7: Core: Parameters	
Parameter Objects	7-2
Parameter Values	7-3
Accessing Parameters	7-4
Designating Parameters for Pro/INTRALINK or Windchill PDMLink.....	7-10
Restricted Parameters	7-10
Table-Restricted Parameters	7-11
Driven Parameters	7-13
Chapter 8: Core: Coordinate Systems and Transformations	
Coordinate Systems.....	8-2
Solid Coordinate System	8-2
Screen Coordinate System.....	8-2
Window Coordinate System	8-3
Drawing Coordinate System	8-3
Drawing View Coordinate System	8-3
Assembly Coordinate System	8-3
Datum Coordinate System	8-4
Section Coordinate System	8-4
Coordinate System Transformations.....	8-4
Transforming Solid to Screen Coordinates.....	8-5
Transforming Screen to Window Coordinates.....	8-8
Transforming from Drawing View to Screen Coordinates in a Drawing	8-9
Transforming from Screen to Drawing Coordinates in a Drawing	8-9
Transforming Coordinates of an Assembly Member	8-12
Transforming to Coordinate System Datum Coordinates.....	8-12
Transforming Coordinates of Sketched Entities	8-12
Chapter 9: Core: Family Tables	
Family Table Objects	9-2
Family Table Utilities	9-2

Visiting Family Tables	9-3
Operations on Family Table Instances	9-4
Operations on Family Table Items	9-7
Chapter 10: Core: External Data	
Introduction to External Data	10-2
Storing External Data.....	10-4
Retrieving External Data.....	10-5
Chapter 11: Core: Cross Sections	
Listing Cross Sections	11-2
Extracting Cross-Sectional Geometry.....	11-3
Visiting Cross Sections	11-6
Creating and Modifying Cross Sections.....	11-7
Mass Properties of Cross Sections.....	11-7
Chapter 12: Core: Utilities	
Configuration Options	12-2
Registry File Data	12-2
Trail Files	12-3
Pro/ENGINEER License Data	12-3
Current Directory	12-3
File Handling.....	12-4
Wide Strings	12-6
Running ModelCHECK	12-7
Creating Custom Checks	12-8
Chapter 13: Core: Asynchronous Mode	
Overview	13-2
Setting up an Asynchronous Pro/TOOLKIT Application.....	13-3
Simple Asynchronous Mode	13-3
Starting and Stopping Pro/ENGINEER	13-4
Connecting to a Pro/ENGINEER Process.....	13-5
Status of a Pro/ENGINEER Process.....	13-9
Full Asynchronous Mode	13-9
Setting Up a Non-Interactive Session	13-10
Chapter 14: User Interface: Messages	
Writing a Message Using a Popup Dialog	14-2
Writing a Message to the Message Window.....	14-4
Text Message File Format and Restrictions.....	14-5
Message Classification	14-7
Writing a Message to an Internal Buffer	14-8
Getting Keyboard Input.....	14-9

Using Default Values.....	14-9
Chapter 15: User Interface: Menus, Commands, and Popupmenus	
Introduction	15-2
Menu Bar Buttons and Menu.....	15-2
Using the Trail File to Determine UI Names	15-4
Adding a PushButton to a Menu Bar Menu	15-5
Adding a Check Button to a Menu Bar Menu	15-10
Adding a RadioButton Group to a Menu Bar Menu	15-13
Adding a Menu to a Menu Bar Menu	15-15
Adding a Menu to the Menu Bar.....	15-16
Manipulating Existing Commands	15-19
Designating Commands.....	15-20
Adding the Command.....	15-20
Designating the Icon.....	15-21
Designating the Command	15-21
Placing the Toolbar Button	15-22
Popup Menus.....	15-31
Adding a Popup Menu to the Graphics Window.....	15-31
Using the Trail File to Determine Existing Popup Menu Names.....	15-32
Registering Notifications to Create and Destroy Popup Menus	15-33
Accessing the Popup Menu	15-33
Creating Commands for the New Popup Menu Buttons.....	15-34
Checking the Access State of a Popup Menu Item	15-34
Adding Pro/ENGINEER Popup Menus.....	15-35
Adding a Button to the Model Tree Popup Menu	15-36
Mode-Specific Buttons and Menu	15-41
Menu Files	15-42
Adding a Menu Button	15-44
New Menus.....	15-47
Preempting Pro/ENGINEER Commands	15-53
Submenus	15-55
Manipulating Menu	15-56
Data Menu	15-56
Setting Menu Buttons	15-57
Controlling Accessibility of Menu Buttons.....	15-58
Pushing and Popping Menus.....	15-58
Run-time Menu	15-59
Customizing the Pro/ENGINEER Navigation Area	15-60
Adding Custom Web Pages	15-60
Adding Custom Dialog Box Components	15-61

Registering Notifications to Add and Destroy Content to a New Pane.....	15-68
Entering Pro/ENGINEER Commands.....	15-69
Execution Rules	15-70
Specifying Keyboard Input	15-73

Chapter 16: User Interface: Dialogs

Introduction	16-3
About Pro/TOOLKIT Support for User Interface.....	16-3
UI Components	16-4
Naming Convention for UI Components.....	16-7
Menubars and Menubar Components.....	16-8
Dialog Attributes	16-9
Dialog Operations.....	16-10
Adding and Removing Components.....	16-14
Dialog Action Callbacks.....	16-16
Cascade Button	16-17
Cascade Button Attributes.....	16-17
Checkbutton.....	16-18
Checkbutton Attributes	16-18
Checkbutton Operations.....	16-19
Checkbutton Action Callbacks.....	16-19
Drawing Area	16-20
Drawing Area Attributes	16-20
Adding and Removing Components.....	16-21
Drawing Area Action Callbacks	16-22
Drawing Area Operations	16-24
Input Panel.....	16-28
Input Panel Attributes	16-28
Input Panel Action Callbacks.....	16-29
Input Panel Operations.....	16-30
Label	16-31
Label Attributes	16-31
Label Operations	16-32
Layout	16-32
Layout Attributes	16-32
Adding and Removing Components.....	16-33
Layout Operations	16-34
List	16-35
List Attributes.....	16-35
List Action Callbacks	16-36
List Operations	16-37

Menubar	16-42
Menubar Attributes	16-42
Menupane	16-43
Menupane Attributes	16-43
Adding and Removing Components	16-44
Optionmenu	16-44
Optionmenu Attributes	16-44
Optionmenu Action Callbacks	16-45
Optionmenu Operations	16-46
Progressbar	16-47
Progressbar Attributes	16-47
Progressbar Operations	16-48
Pushbutton	16-48
Pushbutton Attributes	16-48
Pushbutton Operations	16-49
Pushbutton Action Callbacks	16-50
Radiogroup	16-52
Radiogroup Attributes	16-52
Radiogroup Operations	16-53
Radiogroup Action Callback	16-54
Separator	16-54
Separator Attributes	16-54
Slider	16-55
Slider Attributes	16-55
Slider Operations	16-56
Slider Action Callbacks	16-56
Spinbox	16-61
Spinbox Attributes	16-61
Spinbox Action Callbacks	16-62
Spinbox Operations	16-63
Tab	16-63
Tab Attributes	16-63
Tab Operations	16-64
Tab Action Callbacks	16-66
Table	16-66
Table Attributes	16-66
Adding and Removing Components	16-68
Table Cell Functions	16-68
Table Row Functions	16-72
Table Column Functions	16-73

Table Operations.....	16-77
Table Action Callbacks.....	16-78
Textarea.....	16-81
Textarea Attributes.....	16-81
Textarea Operations.....	16-82
Textarea Action Callbacks.....	16-82
Thumbwheel	16-83
Thumbwheel Attributes.....	16-83
Thumbwheel Operations	16-84
Thumbwheel Action Callbacks	16-85
Tree	16-85
Tree Attributes.....	16-85
Adding and Removing Components.....	16-87
Tree Column Functions	16-88
Tree Node Functions.....	16-89
Tree NodeType Functions.....	16-93
Tree Operations	16-94
Tree Action Callbacks	16-95
Master Table of Resource File Attributes	16-97
Using Resource Files.....	16-110
Location and Translation of Resource Files	16-111
Syntax of Resource Files	16-112

Chapter 17: User Interface: Dashboards

Introduction to Dashboards.....	17-2
Dashboard	17-3
Showing a Dashboard.....	17-3
Accessing a Dashboard	17-4
Dashboard Page.....	17-5
Dashboard Page Options	17-5
Accessing a Dashboard Page	17-6
Accessing Components in the Dashboard Pages	17-7

Chapter 18: User Interface: Basic Graphics

Manipulating Windows.....	18-2
Resizing Windows	18-2
Manipulating the Embedded Browser in Windows.....	18-2
Repainting Windows.....	18-3
Controlling Which Window is Current.....	18-4
Creating and Removing Windows	18-4
Retrieving the Owner of a Window.....	18-5
Visiting Windows	18-5

Activating Windows	18-5
Solid Orientation.....	18-6
Getting and Setting the View Matrix	18-7
Storing Named Views	18-7
Graphics Colors and Line Styles.....	18-10
Setting Colors to Desired Values.....	18-10
Setting Colors to Match Existing Entities.....	18-11
Modifying the Pro/ENGINEER Color Map	18-13
Pro/ENGINEER Color Schemes.....	18-14
Setting Line Styles for Pro/TOOLKIT Graphics	18-15
Displaying Graphics	18-15
Displaying Text.....	18-17
Controlling Text Attributes	18-18
Controlling Text Fonts	18-18
Display Lists	18-19
Getting Mouse Input.....	18-20
Cosmetic Properties.....	18-23
Surface Properties	18-23
Setting Light Sources	18-25

Chapter 19: User Interface: Selection

The Selection Object.....	19-2
Unpacking a ProSelection Object.....	19-3
Building a ProSelection Object	19-4
ProSelection Function Examples.....	19-4
Interactive Selection.....	19-5
Highlighting	19-9
Selection Buffer.....	19-10
Introduction to Selection Buffers.....	19-10
Reading the Contents of the Selection Buffer	19-11
Removing the Items from the Selection Buffer	19-11
Adding Items to the Selection Buffer	19-12

Chapter 20: User Interface: Curve and Surface Collection

Introduction to Curve and Surface Collection.....	20-2
The ProCollection object	20-3
Interactive Collection.....	20-3
Accessing Collection object from Selection Buffer.....	20-6
Adding a Collection Object to the Selection Buffer	20-7
Programmatic Access to Collections.....	20-7
Contents of Curve Collection	20-8

Creation and Modification of Curve Collections	20-10
Contents of Surface Collection.....	20-11
Creation and Modification of Surface Collections.....	20-14
Access of Collection Object from Feature Element Trees	20-15
Programmatic Access to Legacy Collections.....	20-16

Chapter 21: User Interface: Animation

Introduction	21-2
Animation Objects.....	21-3
Animation Frames.....	21-3
Playing Animations	21-4
Single Animation	21-4
Batch Animation	21-5

Chapter 22: Annotations: Annotation Features and Annotations

Overview of Annotation Features	22-3
Creating Annotation Features.....	22-4
Redefining Annotation Features	22-4
Visiting Annotation Features.....	22-5
Visiting Annotation Elements	22-5
Accessing Annotation Elements	22-6
Modification of Annotation Elements	22-9
Parameters Assigned to Annotation Elements.....	22-11
Automatic Propagation of Annotation Elements	22-12
Access to Annotations	22-14
Annotation Orientation	22-17
Annotation Associativity	22-19
Interactive Selection	22-21
Display Modes	22-21
Designating Dimensions and Symbols	22-21
Dimensions	22-22
The ProDimension Object	22-22
Visiting Dimensions	22-22
Modifying Dimensions	22-27
Dimension Tolerances.....	22-32
ISO/DIN Tolerance Table Use.....	22-35
Dimension Text	22-35
Dimension Text Style	22-36
Dimension Location.....	22-36
Dimension Orientation.....	22-44
Driving Dimension Annotation Elements	22-45
Accessing Reference and Driven Dimensions	22-45

Accessing Ordinate and Baseline Dimensions	22-48
Notes.....	22-50
Creating and Deleting Notes	22-50
Note Properties.....	22-50
Visiting Notes.....	22-52
Note Text Styles	22-52
Text Style Properties	22-52
Accessing the Note Placement.....	22-54
Modifying 3D Note Attachments	22-55
Geometric Tolerances.....	22-56
Accessing Set Datum Tags.....	22-56
Accessing Set Datums for Datum Axes or Planes	22-58
Surface Finish Annotations	22-59
Symbol Annotations	22-61
Creating, Reading and Modifying 3D Symbols.....	22-61
Locating 3D Symbols and Symbol Definitions.....	22-62
Chapter 23: Annotations: Geometric Tolerances	
Geometric Tolerance Objects	23-2
ProGtol	23-2
ProGtoldata	23-2
Visiting Geometric Tolerances	23-2
Reading Geometric Tolerances	23-3
Creating a Geometric Tolerance	23-7
Editing a Geometric Tolerance.....	23-15
Deleting a Geometric Tolerance	23-16
Geometric Tolerance Layout.....	23-16
Additional Text for Geometric Tolerances.....	23-17
Geometric Tolerance Text Style.....	23-18
Prefix and Suffix for Geometric Tolerances	23-18
Parameters for Geometric Tolerance Attributes.....	23-19
Chapter 24: Annotations: Designated Area Feature	
Introduction to Designated Area Feature	24-2
Feature Element Tree for the Designated Area	24-2
Accessing Designated Area Properties.....	24-5
Chapter 25: Data Management: Windchill and Pro/INTRALINK Operations	
Introduction	25-2
Non-Interactive Mode Operations.....	25-2
Accessing a Windchill Server from a Pro/ENGINEER Session	25-3
Accessing Information Before Registering a Server.....	25-3

Registering and Activating a Server	25-4
Accessing Information From a Registered Server.....	25-5
Accessing the Workspace	25-6
Workspace Data.....	25-6
Creating and Modifying the Workspace	25-7
Workflow to Register a Server	25-8
Aliased URL.....	25-8
Server Operations.....	25-10
Save	25-11
Upload.....	25-12
CheckIn	25-12
Retrieval	25-15
Checkout and Download	25-16
Undo Checkout.....	25-19
Import and Export.....	25-19
File Copy	25-21
Server Object Status	25-22
Object Lock Status	25-23
Delete Objects.....	25-25
Conflicts During Server Operations.....	25-25
Utility APIs	25-26
Sample Batch Workflow.....	25-27

Chapter 26: Interface: Data Exchange

Exporting Information Files	26-2
Exporting 2D Models	26-5
Automatic Printing of 3D Models	26-16
Exporting 3D Models	26-21
Shrinkwrap Export	26-31
Setting Shrinkwrap Options.....	26-32
Exporting to PDF and U3D	26-37
Importing Parameter Files	26-46
Importing 2D Models.....	26-48
Importing 3D Models.....	26-48
Modifying the Imported Layers.....	26-51

Chapter 27: Interface: Importing Features

Creating Import Features from Files	27-2
Creating Import Features from Arbitrary Geometric Data.....	27-4
Allocating ProInterfacedata	27-4
Adding Surfaces.....	27-5
Adding Edges.....	27-9

Adding Quilts	27-12
Adding Datums	27-13
Creating Features from the Interface Data	27-14
Import Feature Attributes	27-15
Redefining the Import Feature	27-16
Import Feature Properties	27-17
Extracting Pro/ENGINEER Geometry as Interface Data.....	27-19
Chapter 28: Interface: Customized Plot Driver	
Using the Plot Driver Functionality	28-2
Chapter 29: Element Trees: Principles of Feature Creation	
Overview of Feature Creation	29-2
References to Feature Creation Data.....	29-2
Feature Creation.....	29-2
Feature Element Values	29-9
Feature Element Paths	29-11
Feature Elements	29-13
Access to ProElement Data.....	29-14
Feature Element Diagnostics	29-17
Calling ProFeatureCreate()	29-18
Example of Complete Feature Creation	29-21
Feature Inquiry	29-26
Feature Redefine	29-27
XML Representation of Feature Element Trees.....	29-28
Introduction to Feature Element Trees in XML	29-28
Validation Using XML Schema	29-29
XML Representations for Common Elements	29-32
Tips for Recycling XML Output of Element Trees	29-38
Chapter 30: Element Trees: References	
Overview of Reference Objects	30-2
Reading References	30-2
Modifying References	30-5
Chapter 31: Element Trees: Datum Features	
Datum Plane Features	31-2
Datum Point Features	31-10
Sketched Datum Point.....	31-10
Field Datum Point	31-19
Offset Csys Datum Point	31-24
General Datum Point	31-32
Datum Axis Features.....	31-56

Datum Coordinate System Features	31-68
Chapter 32: Element Trees: Datum Curves	
Datum Curve Features	32-2
Common Elements.....	32-2
Datum Curve Types.....	32-2
Sketched Datum Curves	32-3
Trim Datum Curves	32-3
Intersect Datum Curves.....	32-3
Wrap Datum Curves.....	32-5
Offset Datum Curves.....	32-6
Tangent Offset Datum Curves.....	32-7
Other Datum Curve Types.....	32-7
Chapter 33: Element Trees: Edit Menu Features	
Mirror Feature	33-2
The Feature Element Tree for Mirror feature in Pro/ENGINEER	33-3
Creating a Mirror Feature	33-4
Redefining a Mirror Feature	33-4
Accessing a Mirror Feature	33-4
Move Feature.....	33-4
The Feature Element Tree for Move feature in Pro/ENGINEER.....	33-6
Creating a Move Feature.....	33-7
Redefining a Move Feature	33-7
Accessing a Move Feature	33-8
Fill Feature	33-8
The Feature Element Tree for Fill feature in Pro/ENGINEER	33-8
Creating a Fill Feature.....	33-10
Redefining a Fill Feature	33-10
Accessing a Fill Feature	33-10
Intersect Feature.....	33-11
Merge Feature	33-11
Feature Element Tree for Merge feature in Pro/ENGINEER.....	33-12
Creating a Merge Feature	33-13
Redefining a Merge Feature.....	33-13
Accessing a Merge Feature	33-14
Pattern Feature.....	33-14
Wrap Feature.....	33-14
Trim Feature	33-14
The Feature Element Tree for Trim feature in Pro/ENGINEER	33-15
Creating a Trim Feature	33-20
Redefining a Trim Feature.....	33-21

Accessing a Trim Feature	33-21
Offset Feature	33-21
Thicken Feature	33-21
The Feature Element Tree for Thicken feature in Pro/ENGINEER	33-22
Creating a Thicken Feature	33-23
Redefining a Thicken Feature	33-23
Accessing a Thicken Feature	33-24
Solidify Feature	33-24
The Feature Element Tree for Solidify Feature in Pro/ENGINEER	33-24
Creating a Solidify Feature	33-26
Redefining a Solidify Feature	33-27
Accessing a Solidify Feature	33-27
Remove Feature	33-27
Feature Element Tree for the Remove Feature	33-28
Creating the Remove Feature	33-31
Redefining the Remove Feature	33-31
Accessing the Remove Feature	33-31
Chapter 34: Element Trees: Replace	
Introduction	34-2
The Feature Element Tree	34-2
Chapter 35: Element Trees: Draft Features	
Draft Feature	35-2
Feature Element Tree for the Draft Feature	35-2
Creating a Draft	35-8
Redefining a Draft	35-8
Accessing a Draft	35-8
Variable Pull Direction Draft Feature	35-35
Feature Element Tree for the Variable Pull Direction Draft Feature	35-35
Element Details of the Subtree PRO_E_VPDD_SET_CMP	35-39
Creating a VPDD	35-40
Redefining a VPDD	35-40
Accessing a VPDD	35-41
Chapter 36: Element Trees: Round and Chamfer	
Round Feature	36-2
Feature Element Tree for Round Feature	36-2
Creating a Round	36-12
Redefining a Round	36-13
Accessing a Round	36-13
Auto Round Feature	36-21

Chamfer Feature.....	36-22
Feature Element Tree for Chamfer Feature	36-22
Creating a Chamfer	36-29
Redefining a Chamfer	36-30
Accessing a Chamfer	36-30
Corner Chamfer Feature.....	36-46
Feature Element Tree for Corner Chamfers.....	36-46
Creating a Corner Chamfer	36-47
Redefining a Corner Chamfer	36-47
Accessing a Corner Chamfer	36-47

Chapter 37: Element Trees: Hole

Overview	37-2
Feature Element Tree for Hole Features	37-3
Feature Element Data Types.....	37-5
Common Element Values	37-8
PRO_E_HLE_COM Values	37-9
Straight Hole.....	37-9
Sketched Hole	37-18
Standard Threaded Hole	37-40
Standard Clearance Hole	37-52
Custom Hole.....	37-63
Valid PRO_E_HLE_COM Sub-Elements.....	37-64
Hole Placement Types.....	37-69
Hole Placement	37-69
Linear Hole on a Plane.....	37-70
Radial Hole on Plane with Radial Dimensioning	37-71
Radial Hole on Plane with Diameter Dimensioning.....	37-71
Radial Hole on Plane with Linear Dimensioning	37-71
Radial Hole on Cone or Cylinder.....	37-72
Coaxial Hole with Axis as Primary Reference	37-72
Coaxial Hole with Primary Reference not Axis.....	37-72
Onpoint Hole	37-73
Miscellaneous Information	37-73
Hole Parameter Files.....	37-73
Hole Diameter	37-74
Order of Element Specification.....	37-74
Hole-specific Functions	37-74

Chapter 38: Element Trees: Shell

Introduction to Shell Feature.....	38-2
Feature Element Tree for the Shell Feature	38-3

Creating a Shell Feature	38-5
Redefining a Shell Feature.....	38-5
Accessing a Shell Feature	38-5
Chapter 39: Element Trees: Patterns	
Introduction	39-2
The Element Tree for Pattern Creation	39-2
Reference Patterns.....	39-6
Dimension Patterns	39-6
Table Patterns	39-8
Fill Patterns.....	39-10
Direction Patterns	39-12
Axis Patterns	39-15
Curve Patterns.....	39-17
Point Patterns	39-18
NC Sequence Pattern.....	39-19
Obtaining the Element Tree for a Pattern	39-22
Visiting and Creating a Pattern	39-23
Chapter 40: Element Trees: Sections	
Overview	40-2
Creating Section Models	40-2
Allocating a Two-Dimensional Section	40-3
Setting the Mode of a Section	40-4
Copying the Current Section	40-4
Section Constraints	40-5
Solving and Regenerating a Section	40-7
Automatic Section Dimensioning.....	40-9
Adding Section Entities.....	40-9
Accessing Selection Reference of the Entity.....	40-11
Construction Entities.....	40-11
Modifying Entities	40-12
Adding Section Dimensions.....	40-12
Error Reporting	40-24
Retrieving and Saving a Section	40-26
Chapter 41: Element Trees: Sketched Features	
Overview	41-2
Element Tree for Sketched Features.....	41-2
Creating Features Containing Sections	41-4
Creating Features with 2D Sections.....	41-5
Verifying Section Shapes.....	41-5

Creating Features with 3D Sections	41-6
3D Section Location in the Owning Model	41-7
Reference Entities and Use Edge.....	41-8
Creating Geometry by Offsetting.....	41-9
Reusing Existing Sketches	41-10
Chapter 42: Element Trees: Extrude and Revolve	
The Element Tree for Extruded Features	42-2
The Element Tree for Revolved Features.....	42-68
The Element Tree for First Features.....	42-106
Chapter 43: Element Trees: Basic Sweep	
Sweeps in Pro/ENGINEER Wildfire	43-2
The Element Tree for Sweeps.....	43-2
Creating a Swept Feature	43-5
Chapter 44: Element Trees: ECAD Area Feature	
Introduction to ECAD Area Feature	44-2
Feature Element Tree for the ECAD Area	44-2
Chapter 45: Assembly: Basic Assembly Access	
Structure of Assemblies and Assembly Objects	45-2
Visiting Assembly Components	45-5
Properties Related to Component Purpose.....	45-5
Component Placement.....	45-7
Simplified Representations.....	45-7
Modifying Component Properties	45-8
Locations of Assembly Components	45-11
Assembling Components	45-13
Redefining and Rerouting Components.....	45-13
Deleting Components	45-13
Exploded Assemblies	45-14
Explode State Objects.....	45-14
Visiting Explode States.....	45-14
Explode State Access	45-15
Merge and Cutout.....	45-15
Automatic Interchange	45-15
Chapter 46: Assembly: Top-down Design	
Overview	46-2
Defining Design Intent.....	46-2
Defining Preliminary Product Structure	46-2
Introducing Skeleton Models	46-2
Communicating Design Intent Throughout the Assembly Structure.....	46-3

Continued Population of the Assembly	46-3
Managing Part Interdependencies	46-3
Skeleton Model Functions	46-5
Assembly Component Functions	46-5
External Reference Control Functions	46-6
Feature and CopyGeom Feature Functions	46-7
External Reference Data Gathering	46-9
Chapter 47: Assembly: Assembling Components	
Assembling Components by Functions	47-2
Assembling a Component Parametrically	47-3
Redefining Components Interactively	47-16
Assembling Components by Element Tree	47-17
The Element Tree for an Assembly Component	47-17
Model	47-17
Attributes	47-18
Initial Position	47-18
Constraint Sets and Mechanism Connections	47-18
Placement Constraints	47-23
Component Movement in Assembly	47-25
Placement via Interface	47-26
Assembling Components Using Intent Datums	47-27
Chapter 48: Assembly: Kinematic Dragging and Creating Snapshots	
Connecting to a Kinematic Drag Session	48-2
Performing Kinematic Drag	48-4
Creating and Modifying Snapshots	48-5
Snapshot Constraints	48-5
Snapshot Transforms	48-9
Snapshots in Drawing Views	48-9
Chapter 49: Assembly: Simplified Representations	
Overview	49-2
Simplified Representations in Session	49-3
Retrieving Simplified Representations	49-5
Creating and Deleting Simplified Representations	49-6
Extracting Information About Simplified Representations	49-9
Modifying Simplified Representations	49-14
Adding Items to and Deleting Items from a Simplified Representation	49-14
Gathering Components by Rule	49-15
Gathering by Model Name	49-16
Gathering by Parameters	49-16

Gathering by Zone.....	49-17
Gathering by Distance from a Point	49-17
Gathering by Size.....	49-17
Gathering by Simplified Representation.....	49-18

Chapter 50: Assembly: Data Sharing Features

Copy Geometry, Publish Geometry, and Shrinkwrap Features.....	50-2
Feature Element Tree for the Copy Geometry, Publish Geometry, and Shrinkwrap Features	50-2
Element Details of the Subtree PRO_E_CG_LOCATION.....	50-9
Element Details of PRO_E_DSF_PROPAGATE_ANNOTS	50-10
Saved Queries for Copy Geometry and Publish Geometry Features	50-11
General Merge (Merge, Cutout and Inheritance Feature)	50-12
Feature Element Tree	50-12
Inheritance Feature and Flexible Component Variant Items	50-15
Variant Feature Model.....	50-15
Accessing Properties of Variant Features	50-16
Variant Model Items	50-21
Variant Parameters	50-22
Variant References.....	50-23

Chapter 51: Drawings

Creating Drawings from Templates	51-2
Diagnosing Drawing Creation Errors	51-4
Drawing Setup	51-6
Context in Drawing Mode	51-6
Access Drawing Location in Grid.....	51-7
Drawing Tree	51-7
Merge Drawings.....	51-11
Drawing Sheets	51-11
Drawing Views and Models	51-16
Listing Drawing Views	51-17
Modifying Views	51-22
Creating Views	51-24
Background Views.....	51-27
Detailed Views.....	51-28
Auxiliary Views	51-29
Revolved Views.....	51-30
View Orientation	51-30
Visible Areas of Views.....	51-32
Sections of a View	51-35
View States	51-37

Drawing Models	51-47
Detail Items	51-49
Listing Detail Items	51-51
Displaying Detail Items	51-52
Creating, Modifying and Reading Detail Items	51-53
Draft Entity Data	51-56
Accessing OLE Objects	51-60
Detail Note Text Data	51-61
Detail Note Line Data	51-62
Detail Attachments and Leaders	51-64
Detail Note Data	51-66
Read-Only Notes	51-71
Parameterized Note Text	51-71
Cross-referencing Gtols and Drawing Annotations	51-72
Cross-referencing 3D Notes and Drawing Annotations	51-72
Symbol Definition Attachments	51-73
Symbol Definition Data	51-74
Creating a Symbol Definition	51-75
Retrieving a Symbol Definition from Disk	51-79
Symbol Instance Variable Text	51-80
Symbol Instance Data	51-80
Cross-referencing Weld Symbols and Drawing Annotations	51-83
Detail Group Data	51-85
Drawing Symbol Groups	51-87
Identifying Symbol Groups in an Instance	51-88
Identifying Symbol Groups in a Definition	51-88
Manipulating Symbol Groups	51-89
Drawing Edges	51-90
Drawing Tables	51-91
Selecting Drawing Tables and Cells	51-91
Creating Drawing Tables	51-92
Reading Drawing Tables	51-93
Modifying Drawing Tables	51-95
Drawing Table Segments	51-103
Repeat Regions	51-103
Drawing Dimensions	51-104
Drawing Dimension Attachments and Dimension Creation	51-105
Ordinate Dimensions	51-109
Other Drawing Dimension Properties	51-115

Chapter 52: Production Applications: Sheetmetal

Geometry Analysis.....	52-2
Bend Tables and Dimensions.....	52-9
Sheetmetal Flat Wall Feature	52-10
Sheetmetal Flat Wall based on the Fill Tool.....	52-10
Feature Element Tree for the Sheetmetal Flat Wall Feature.....	52-10
Creating a Sheetmetal Flat Wall Feature	52-16
Redefining a Sheetmetal Flat Wall Feature.....	52-16
Accessing a Sheetmetal Flat Wall Feature	52-17
Sheetmetal Flange Wall Feature	52-30
Feature Element Tree for the Sheetmetal Flange Wall Feature.....	52-30
Creating a Sheetmetal Flange Wall Feature	52-42
Redefining a Sheetmetal Flange Wall Feature.....	52-43
Accessing a Sheetmetal Flange Wall Feature	52-43
Sheetmetal Extruded Wall Features.....	52-57
Sheetmetal Cut Features	52-58
Punch Form Feature.....	52-58
Feature Element Tree for the Punch Form Feature	52-58
Creating a Punch Form Feature.....	52-63
Redefining a Punch Form Feature	52-63
Accessing a Punch Form Feature	52-64
Quilt Form Feature.....	52-64
Feature Element Tree for the Quilt Form Feature	52-64
Creating a Quilt Form Feature.....	52-67
Redefining a Quilt Form Feature	52-67
Accessing a Quilt Form Feature.....	52-67

Chapter 53: Production Applications: Manufacturing

Manufacturing Models	53-2
Creating a Manufacturing Model.....	53-3
Analyzing a Manufacturing Model.....	53-3
Traversing Manufacturing Components	53-5
Identifying the Storage Solid	53-6
Visiting Manufacturing Tools	53-8
Creating Manufacturing Objects	53-9
Creating Tools.....	53-9
Manufacturing Parameters	53-16
Using External Functions to Define Parameters in the Manufacturing Step Table	53-21
Creating Manufacturing Features.....	53-25
Creating Fixtures	53-26

Creating Workcells	53-27
Creating Operations	53-33
Creating NC Sequences	53-36
Analyzing Manufacturing Features.....	53-45
Chapter 54: Production Applications: Customized Tool Database	
Overview	54-2
Setting up the Database and Custom Search Parameters	54-2
Registering the External Database	54-3
Querying the External Database	54-4
Returning the Search Results	54-8
Chapter 55: Production Applications: NC Sequences and CL Commands	
Auxiliary and Custom NC Sequences	55-2
CL Commands	55-3
Customizing CL Commands	55-4
Chapter 56: Production Applications: Process Planning	
Process Step Objects.....	56-2
Visiting Process Steps	56-2
Process Step Access	56-2
Creating Process Steps	56-3
Feature Elements	56-4
Types of Process Step	56-5
Optional Elements	56-6
General Process Steps.....	56-6
Reposition Process Steps	56-6
Chapter 57: Production Applications: NC Process Manager	
Overview	57-2
Accessing the Process Manager.....	57-2
Accessing the Process Manager when the Dialog Box is not Active.....	57-2
Accessing the Process Manager User Interface	57-3
Manufacturing Process Items.....	57-4
Steps, Operations and Workcells	57-4
Step Models.....	57-5
Determining the Subtype of a Process Item	57-5
Accessing Details of Process Items	57-5
Visiting Steps, Operations and Workcells.....	57-6
Creating Steps, Operations, and Workcells.....	57-6
Accessing the Properties of Manufacturing Process Items	57-7
Modifying Process Items	57-9
Parameters.....	57-9

Manufacturing Parameters	57-9
Annotation Element Parameters	57-10
Step Parameters and Relations	57-10
Template Parameters	57-10
Special Parameters	57-10
Global Parameters and Relations	57-12
Manufacturing Features	57-12
Import and Export of Process Table Contents	57-13
Notification	57-13

Chapter 58: Production Applications: Diagrams and Cabling

Diagrams	58-2
Listing the Contents of a Diagram	58-2
Diagram Item Color and Line Style	58-3
Diagram Parameters	58-4
Diagram Wire Spools	58-5
Diagram Connectivity	58-5
Pro/ENGINEER Cabling Design	58-6
Creating a Harness	58-6
Finding a Harness	58-7
Finding the Cables in a Harness	58-7
Managing Spools	58-7
Spool Parameters	58-8
Finding Harness Connectors	58-9
Connectors Parameters	58-9
Managing Cables and Bundles	58-12
Cable Parameters	58-13
Cable Identifiers and Types	58-14
Cable Cosmetic Features Create	58-15
Cable Connectivity	58-15
Cable Routing Locations	58-16
Cable Geometry	58-17
Measuring Harness Clearance	58-17
Cable Routing	58-18

Chapter 59: Production Applications: Piping

Piping Terminology	59-2
Linestock Management Functions	59-3
Linestocks	59-3
Linestock Parameters	59-3
Pipeline Connectivity Analysis	59-5
Networks	59-5

Extensions	59-6
Members.....	59-6
Terminators	59-7
Junctions	59-7
Series	59-7
Objects	59-8
Segments	59-9
Chapter 60: Production Applications: Welding	
Read Access to Weld Features.....	60-2
Customizing Weld Drawing Symbols	60-3
Chapter 61: Mechanica: Mechanica Items	
Entering the Pro/ENGINEER Mechanica Environment.....	61-3
Selection of Mechanica Items	61-4
Accessing Mechanica Items.....	61-6
Mechanica Object References	61-7
Geometric References	61-8
Y-directions	61-11
Functions.....	61-12
Mechanica Expressions	61-15
Accessing the Properties used for Loads and Constraints	61-15
Mechanica Loads	61-20
Accessing Mechanica Loads	61-20
Modifying Mechanica Loads	61-22
Force and Moment Loads.....	61-29
Pressure, Gravity, and Bearing Loads.....	61-30
Centrifugal Loads	61-32
Temperature Loads	61-32
Heat Loads	61-37
Mechanica Load Sets.....	61-39
Mechanica Constraints.....	61-40
Accessing Mechanica Constraints.....	61-40
Modifying Mechanica Constraints.....	61-41
Convection Constraints	61-42
Displacement Constraints.....	61-43
Symmetry Constraints	61-54
Mechanica Constraint Sets	61-56
Mechanica Beams.....	61-56
Mechanica Beam Sections.....	61-60
Sketched Beam Section.....	61-65

General Beam Section	61-66
Beam Orientations	61-68
Beam Releases	61-70
Mechanica Spring Items	61-71
Mechanica Spring Property Items.....	61-73
Mechanica Mass Items	61-74
Mechanica Mass Properties	61-77
Material Orientations.....	61-78
Mechanica Shells.....	61-82
Shell Properties	61-88
Shell Pairs.....	61-96
Interfaces	61-99
Bonded Interface	61-102
Contact Interface	61-103
Thermal Resistance Interface	61-105
Free Interface	61-106
Gaps	61-106
Mesh Control	61-108
Accessing AutoGEM Edge Distribution and Minimum Edge Mesh Control Data	61-113
Accessing the AutoGEM Edge Length by Curvature Mesh Control Data ...	61-114
Accessing the AutoGEM Maximum Element Size Mesh Control Data.....	61-115
Accessing Edge Distribution Mesh Control Data.....	61-116
Accessing AutoGEM Isolation Data	61-117
Accessing the Displacement Coordinate System Data	61-118
Accessing the Mesh Control Element Size Data.....	61-119
Accessing the Mesh Control Shell Coordinate System Data	61-119
Accessing the Mesh Control Hard Point Data	61-119
Accessing the Mesh Control ID Offset Data.....	61-120
Accessing the Mesh Control Numbering Data	61-120
Accessing the Suppressed Mesh Control Data.....	61-121
Welds	61-121

Chapter 62: Mechanica: Mechanica Geometry

Introduction	62-2
Obtaining Mechanica Geometry from Pro/TOOLKIT	62-4
Accessing the ProMechModel.....	62-6
Accessing the ProMechSolid.....	62-7
Accessing Mechanica ProMechSurface.....	62-8
Geometry Evaluation of ProMechSurface	62-9
Accessing ProMechContour.....	62-10

Accessing ProMechEdge	62-11
Geometry Evaluation of ProMechEdge	62-12
Accessing ProMechVertex	62-13
Accessing ProMechPoint.....	62-14
Accessing ProMechCompositeCurve	62-15
Accessing ProMechCurve	62-16
Geometry Evaluation of ProMechCurves	62-17
To Create a Surface Region Feature	62-18
Chapter 63: Mechanica: Finite Element Modeling (FEM)	
Overview	63-2
Exporting an FEA Mesh	63-2
Getting Constraint Case Names	63-5
Getting Mesh Controls	63-5
Pro_fem_region Data Structure	63-6
Getting Constraints	63-7
Pro_fem_con_value Data Structure	63-10
Getting Bar Elements and Contacts.....	63-10
Getting Mass Elements	63-11
Pro_fem_mass_props Data Structure	63-12
Getting Shell Pairs	63-12
Chapter 64: Mechanism Design: Mechanism Features	
Mechanism Spring Feature.....	64-2
Feature Element Tree for the Mechanism Spring Feature	64-2
Mechanism Damper Feature.....	64-4
Feature Element Tree for the Mechanism Damper Feature	64-4
Mechanism Belt Feature	64-5
Feature Element Tree for the Mechanism Belt Feature	64-6
Mechanism 3D Contact Feature	64-8
Feature Element Tree for the Mechanism 3D Contact Feature.....	64-9
Chapter 65: Event-driven Programming: Notifications	
Using Notify.....	65-2
Notification Types.....	65-2
File Management Events.....	65-2
Post All File Management Events.....	65-3
File Management Failure Events	65-4
Model and Feature Modification Events	65-4
Context Change Events.....	65-6
Graphics Events	65-6
Pro/NC Output Events	65-7

CL Command Events	65-7
Mold Layout Events	65-7
Weld Events	65-7
Chapter 66: Event-driven Programming: External Objects	
Summary of External Objects	66-2
External Objects and Object Classes	66-3
Creating External Object Classes	66-3
Creating External Objects	66-3
External Object Owners	66-4
Recycling External Object Identifiers	66-4
External Object Parameters	66-5
External Types and Identifiers for External Objects	66-5
Visiting External Objects	66-6
External Object Data	66-6
Display Data for External Objects	66-6
Selection Data for External Objects	66-10
Manipulating External Object Data	66-11
External Object References	66-12
Creating External Object References	66-13
Visiting External Object References	66-13
Callbacks for External Objects	66-14
Warning Mechanism for External Objects	66-16
Chapter 67: Event-driven Programming: Toolkit-Based Analysis	
Overview	67-2
Interactive Creation of Toolkit-Based Analysis	67-3
Interactive Creation of Toolkit-Based Analysis Feature	67-3
Storage of Toolkit-Based Analysis Feature in Pro/ENGINEER	67-4
Registering a Toolkit-Based Analysis with Pro/ENGINEER	67-5
Analysis Callbacks	67-5
Pro/TOOLKIT Analysis Information	67-8
Results Data	67-9
ProAnalysisSrfData Structure	67-11
Analysis Attributes	67-27
Visiting Saved Toolkit-Based Analyses	67-27
Visiting Toolkit-Based Analyses Features	67-28
Using the Model without Pro/TOOLKIT	67-28
Chapter 68: Event-driven Programming: Foreign Datum Surfaces and Curves	
Foreign Datum Surfaces	68-2

Importing a Foreign Datum Surface	68-2
Foreign Datum Curves	68-3
Providing an Evaluation Function	68-4
Binding the Evaluation Function to a Class	68-7
Chapter 69: Task Based Application Libraries	
ProArgument and Argument Management	69-2
Creating Pro/TOOLKIT DLL Task Libraries	69-3
Memory Management in Task Library Functions	69-4
Launching Pro/TOOLKIT DLL Functions	69-8
Custom Pro/TOOLKIT DLL Tasks for Distributed Pro/BATCH	69-9
Launching Synchronous J-Link Applications	69-17
Appendix A: Summary of Technical Changes	
Critical Technical Changes	A-2
Compiling and Linking on Windows	A-2
3D Import and 2D PDF Export Using Profiles	A-2
Creating Section Constraints	A-2
Mechanica Updates	A-3
Modifications to Sketched Features	A-4
New Pattern Types	A-4
No-Resolve Mode	A-4
Obsolete HP-GL Pen Plotters	A-5
System Parameters for Geometric Tolerance Attributes	A-5
Ready-to-use Pro/TOOLKIT Application Using Visual Studio	A-5
New Functions	A-6
2D Export	A-6
3D Export	A-6
3D Import	A-7
Animation	A-7
Annotations	A-7
Annotation Planes	A-8
Cross Sections	A-8
Data Sharing Features	A-8
Designated Area Features	A-8
Dimensions	A-9
Drawings	A-9
Family Tables	A-13
Feature	A-13
Geometry: Quilts	A-14
Import Feature	A-14

Layers.....	A-14
Manufacturing Process Manager	A-15
Mechanica Items	A-16
Mechanism Design: Kinematic Drag and Snapshots	A-18
Model.....	A-19
Parameter.....	A-20
Patterns	A-20
Sections.....	A-20
Simplified Representations.....	A-22
Solid	A-22
Quick Print.....	A-24
User Interface: Dialog Boxes.....	A-24
User Interface: Navigation Area	A-25
User Interface: Main Window	A-25
User Interface: Accessory Window	A-26
User Interface: Properties of Layout.....	A-26
Windchill Server Connectivity.....	A-27
Superseded Functions.....	A-27
Miscellaneous Technical Changes	A-29
2D Export Formats	A-29
3D Contact Feature Element Tree	A-29
Annotations and Reference Planes.....	A-29
Belt Feature Element Tree	A-30
Color Schemes.....	A-30
Cross Sections	A-30
Datum Coordinate System Element Tree.....	A-30
Datum Target Annotation Features	A-31
Designated Area Element Tree	A-31
Dimensions in Draft Groups	A-31
Enhancements to ProModelItemNameGet() and Set().....	A-32
Hole Element Tree	A-32
Inheritance Feature Support.....	A-32
Intent Datums.....	A-33
Issue with Legacy Toolkit Applications.....	A-34
Materials and Material Properties.....	A-34
Mechanica Updates.....	A-34
New Types for ProFileList().....	A-35
NC Sequence Pattern Support.....	A-36
Obsolete Data Exchange Formats	A-36
PDF/A Format	A-37

PRO_FEAT_UNREGENERATED Status	A-37
Punch Form Feature Element Tree	A-37
Quilt Form Feature Element Tree	A-37
Revision Field in the Registry File	A-37
Round Element Tree	A-38
Section Dimensions	A-38
Set Datums in Inheritance Features	A-38
Simplified Representations	A-38
SolidWorks and Inventor 3D Import Formats	A-38
Shrinkwrap Feature Element Tree	A-39
Spring Feature Element Tree	A-39
Surface Region Element Tree	A-39
Support for the Detailed Drawing User Interface	A-39
True Type Font Support	A-40
Variable Pull Direction Draft Feature Element Tree	A-40
Weld Symbol Annotation Elements	A-40

Appendix B: Unicode Encoding

Introduction to Unicode Encoding	B-2
Unicode Encoding and Pro/TOOLKIT	B-3
Necessity of Unicode Compliance	B-4
Platform Native Encoding and Unicode	B-5
Legacy Encoding and Unicode	B-10
Programming Considerations	B-11
External Interface Handling	B-15
Special External Interface: printf() and scanf() Functions	B-16
Special External Interface: Windows-runtime Functions	B-17
Special External Interface: Hardcoded Strings	B-17
Support for Legacy Encoding in Pro/TOOLKIT	B-21
Using Legacy Encoding Mode	B-23
Legacy Applications	B-24

Appendix C: ptk_revtool Utility

Overview	C-2
ptk_revtool Utility Features	C-2
Running ptk_revtool	C-3
Syntax	C-3
Output Messages	C-4

Appendix D: Pro/TOOLKIT Registry File

Registry File Fields	D-2
Sample Registry Files	D-4

Appendix E: Pro/TOOLKIT Library Types

Standard Libraries	E-2
Alternate Libraries.....	E-2
PIC Libraries for Some UNIX platforms.....	E-3

Appendix F: Pro/TOOLKIT Sample Applications

Installing Sample Applications	F-2
Details on Sample Applications	F-2

Appendix G: Advanced Licensing Options

Advance Licensing Options for Pro/TOOLKIT	G-2

Appendix H: Digital Rights Management

Introduction	H-2
Implications of DRM on Pro/TOOLKIT.....	H-2
Error Return Types.....	H-3
Copy Permission to Interactively Open Models.....	H-5
Additional DRM Implications.....	H-6

Appendix I: Pro/DEVELOP to Pro/TOOLKIT Function Mapping

The Relationship Between Pro/TOOLKIT and Pro/DEVELOP	I-2
Converting from Pro/DEVELOP	I-2
Using Pro/DEVELOP Applications with Pro/TOOLKIT	I-3
Techniques of Conversion and Mixing	I-4
Equivalent Pro/DEVELOP Functions	I-16
Data Structures for Pro/DEVELOP-Style Functions	I-30
Prodtl_item Structure Definition.....	I-30
Ptc_surf Structure Definition.....	I-33
Select3d Structure Definition.....	I-38

Appendix J: Geometry Traversal

Overview	J-2
Geometry Terms.....	J-2

Appendix K: Geometry Representations

Domain of Evaluation.....	K-2
Surface Data Structures	K-3
Plane	K-4
Cylinder	K-5
Cone.....	K-6
Torus	K-6
General Surface of Revolution	K-7
Ruled Surface.....	K-8
Tabulated Cylinder	K-9

Coons Patch	K-10
Fillet Surface.....	K-10
Spline Surface	K-11
Second Derivative Spline Surface	K-12
NURBS Surface.....	K-13
Cylindrical Spline Surface.....	K-14
Foreign Surface	K-16
Edge and Curve Data Structures	K-16
Arc	K-17
Line.....	K-17
NURBS	K-18
Spline.....	K-19
Ellipse	K-19

Appendix L: Debugging Pro/TOOLKIT Applications

Building a Pro/TOOLKIT Application for Debugging	L-2
Debugging Techniques	L-2
Debugging an Interactive DLL	L-2
Debugging a Batch Mode DLL	L-3
Debugging Pro/TOOLKIT DLLs on Windows	L-4
Debugging a Multiprocess Application	L-5
Debugging a Synchronous Spawn Mode Application.....	L-5
Debugging an Asynchronous Spawn Mode Application	L-5

Glossary

Glossary-1

About This Guide

This section contains information about the contents of this user's guide and the conventions used.

Topic	Page
Purpose	ii
Audience and Prerequisite Experience	ii
Contents	ii
Documentation	iii

Purpose

The *Pro/TOOLKIT User's Guide* describes how to use Pro/TOOLKIT, the C-language customization toolkit for Pro/ENGINEER from PTC (Parametric Technology Corporation). Pro/TOOLKIT provides customers and third-parties the ability to expand Pro/ENGINEER capabilities by writing C-language code and seamlessly integrating the resulting application into Pro/ENGINEER.

Pro/TOOLKIT provides a large library of C functions that enables the external application to access the Pro/ENGINEER database and user interface in a controlled and safe manner.

This manual introduces Pro/TOOLKIT, the features it offers, and the techniques and background knowledge users require to use it.

Before approaching Pro/TOOLKIT, it is important to have a good practical knowledge of the C language, and also to have practical experience with using Pro/ENGINEER, especially in areas that you intend to customize. Chapter 1 deals with Fundamentals, and should be studied thoroughly by beginners. Chapter 2, Converting from Pro/DEVELOP, is aimed at users who are already familiar with Pro/DEVELOP, the predecessor to Pro/TOOLKIT; other users can skip Chapter 2. Chapters 3 through 5 contain basic information useful to almost all Pro/TOOLKIT users. After that, you may jump to chapters that describe functionality specific to your needs.

Audience and Prerequisite Experience

Pro/TOOLKIT is aimed at software engineers with experience in C programming. They should also be trained in the basic use of Pro/ENGINEER.

Contents

This manual contains the chapters that describe how to work with different functions provided by Pro/TOOLKIT.

Documentation

The documentation for Pro/TOOLKIT includes an online browser that contains the Pro/TOOLKIT User's Guide and describes Pro/TOOLKIT function syntax.

Conventions

The following table lists conventions and terms used throughout this book.

Convention	Description
UPPERCASE	Pro/ENGINEER-type menu name (for example, PART).
Boldface	Windows-type menu name or menu or dialog box option (for example, View), or utility (for example, promonitor). Function names also appear in boldface font.
Monospace (Courier)	Code samples appear in <code>courier</code> font.
SMALLCAPS	Key names appear in smallcaps (for example, ENTER).
<i>Emphasis</i>	Important information appears <i>in italics</i> . Italic font also indicates file names and function arguments.
Choose	Highlight a menu option by placing the arrow cursor on the option and pressing the left mouse button.
Select	A synonym for “choose” as above, Select also describes the actions of selecting elements on a model and checking boxes.
Element	An element describes redefinable characteristics of a feature in a model.
Mode	An environment in Pro/ENGINEER in which you can perform a group of closely related functions (Drawing, for example).
Model	An assembly, part, drawing, format, layout, case study, sketch, and so on.
Option	An item in a menu or an entry in a configuration file or a setup file.
Solid	A part or an assembly.

- Important information that should not be overlooked appears in notes like this.
- All references to mouse clicks assume use of a right-handed mouse.

1

Fundamentals

This chapter describes fundamental Pro/TOOLKIT concepts and functions.

Topic	Page
Introduction to Pro/TOOLKIT	1 - 2
Online Documentation — Pro/TOOLKIT APIWizard	1 - 2
The APIWizard Interface	1 - 6
APIWizard Search Feature (Find)	1 - 13
Pro/TOOLKIT Style	1 - 17
Installing Pro/TOOLKIT	1 - 20
Developing a Pro/TOOLKIT Application	1 - 30
User-Supplied Main	1 - 48
Asynchronous Mode	1 - 49
Pro/TOOLKIT Techniques	1 - 49
Visit Functions	1 - 59
Wide Strings	1 - 64
String and Widestring Functions	1 - 66

Introduction to Pro/TOOLKIT

Pro/TOOLKIT is the customization toolkit for Pro/ENGINEER from Parametric Technology Corporation (PTC). It gives customers and third-parties the ability to expand Pro/ENGINEER capabilities by writing C programming language code and then seamlessly integrating the resulting application into Pro/ENGINEER.

Pro/TOOLKIT provides a large library of C functions to provide the external application safe and controlled access to the Pro/ENGINEER database and applications. Pro/TOOLKIT is the primary PTC application programmer's interface (API) for Pro/ENGINEER.

Online Documentation — Pro/TOOLKIT APIWizard

Pro/TOOLKIT provides an online browser called the APIWizard that displays detailed documentation data. This browser displays information from the Pro/TOOLKIT *Users' Guide* and API specifications derived from Pro/TOOLKIT header file data.

The Pro/TOOLKIT APIWizard contains the following:

- Definitions of Pro/TOOLKIT objects and their hierarchical relationships
- Definitions of Pro/TOOLKIT functions
- Declarations of data types used by Pro/TOOLKIT functions
- The Pro/TOOLKIT *Users' Guide*, which users can browse by topic or by object
- Code examples for Pro/TOOLKIT functions (taken from applications provided as part of the Pro/TOOLKIT installation)

Review the Release Notes and file

`<proe_loadpoint>/protoolkit/README` file for the most up-to-date information on documentation changes.

Note: The User's Guide is also available in PDF format. This file is located at:

`<proe_loadpoint>/protoolkit/tkuse.pdf`

To Install the APIWizard

The Pro/ENGINEER product CD installation procedure automatically installs the Pro/TOOLKIT APIWizard. The files reside in a directory under the Pro/ENGINEER load point. The location for the Pro/TOOLKIT APIWizard files is:

```
<proe_loadpoint>/protoolkit/protkdoc
```

To load the APIWizard manually, copy all files from

```
<proe_loadpoint>/protoolkit/protkdoc
```

to your target directory.

To Run the APIWizard

Start the Pro/TOOLKIT APIWizard by pointing your browser to:

```
<proe_loadpoint>/protoolkit/protkdoc/index.html
```

Your web browser will display the Pro/TOOLKIT APIWizard data in a new window. See the Web Browser Environments section for more information on requirements for the APIWizard software environment.

Web Browser Environments

The APIWizard supports Netscape Navigator version 4 and later, and Internet Explorer version 5 and later according to the following requirements:

- Use of the APIWizard with Internet Explorer requires installation of the Java2 plug-in.
- Use of Netscape Navigator requires installation of the Java Swing foundation class. If this class is not loaded on your computer, the APIWizard loads it for you. Loading can take several minutes, and the library does not persist between sessions. See Loading the Swing Class Library for the procedure to load Swing permanently.
- Loading the Swing Class Library

If you access the APIWizard with Internet Explorer, download and install Internet Explorer's Java2 plug-in. This plug-in is preferred over installing the Swing archive for Internet Explorer because Swing degrades access time for the APIWizard Search feature.

If you access the APIWizard with Netscape Navigator use the following procedures in this section to download and install the Java Foundation Class (Swing) archive:

- To Download the Java Foundation Class (Swing) Archive
- To Modify the Java Class Path on UNIX Platforms
- To Modify the Java Class Path on NT Platforms

To Download the Java Foundation Class (Swing) Archive

1. Navigate to the Java Foundation Class Download Page.
2. Navigate to the heading Downloading the JFC/Swing X.X.X Release, where X.X.X is the latest JFC version.
3. Click the standard TAR or ZIP file link. The heading Download the Standard Version is displayed.
Note: Do not download the “installer” version.
4. Select a file format, click Continue, and follow the download instructions on the subsequent pages.
5. Uncompress the downloaded bundle.
6. After downloading the swing-X.X.Xfcs directory (where X.X.X is the version of the downloaded JFC) created when uncompressing the bundle, locate the *swingall.jar* archive.
7. Add the *swingall.jar* archive to the Java Class Path as shown in the following sections.

To Modify the Java Class Path on UNIX Platforms

To make the Java Foundation Class (Swing) available in UNIX shell environments copy file *swingall.jar* to the `java/classes` directory under your Netscape installation point.

Follow the installation instructions listed on the Java Foundation Class Download Page.

To Modify the Java Class Path on NT Platforms

To make the Java Foundation Class (Swing) available on Windows NT Platforms:

For Netscape

1. Copy *swingall.jar* from the location to which you downloaded it.
2. Paste *swingall.jar* into the following location:

`[system_disk]/Program Files/Netscape/Communicator/Program/java/classes`

Note: You do not need to reboot your machine for this change to take effect.

For Internet Explorer

Install the APIWizard Internet Explorer Java2 plug-in.

Troubleshooting

When attempting to run the APIWizard in Internet Explorer, the browser detects the presence of the Java2 plug-in, though it might not be installed on the computer. The browser jumps to the “loading” page and waits for the launch of the Java applet. The applet is not launched since the Java2 plug-in is not found. The browser remains at the “loading” state indefinitely.

The workaround for this is to enable Java console in the Internet Explorer options. This can be done as follows:

1. Launch the Internet Explorer browser.
2. Click **Tools>Internet Options**. The **Internet Options** dialog box opens.
3. Click the **Advanced** tab.
4. Under **Microsoft VM** check **Java console enabled**.
5. Click **OK**.
6. Click **File>Close**.
7. Restart the Internet Explorer browser.

Automatic Index Tree Updating

With your browser environment configured correctly, following a link in an APIWizard HTML file causes the tree in the Selection frame to update and scroll the tree reference that corresponds to the newly displayed page. This feature is called automatic tree scrolling.

Netscape on NT

If you access the APIWizard through Netscape’s Java2 plug-in, this feature is not available. You must install the Java foundation class called Swing for this function to work. See Loading the Swing Class Library for the procedure on loading Swing.

Internet Explorer on NT

If you access the APIWizard with Internet Explorer, download and install the Internet Explorer Java2 plug-in to make automatic tree scrolling available.

The APIWizard starts and displays pages stating it is checking for the proper environment, loading the data, or that you need to set up your environment as described in the Web Browser Environments section.

The APIWizard Interface

The APIWizard interface consists of two frames. The following sections describe how to display and use the APIWizard frames in your Web browser.

Topic/Object/Category Selection Frame

This frame, located on the left of the screen, controls what is presented in the Display frame. Specify the data to view by choosing the Pro/TOOLKIT **Categories**, or Pro/TOOLKIT **Objects**, or Pro/TOOLKIT **Users' Guide** menu items.

In Categories mode, this frame displays an alphabetical list of the Pro/TOOLKIT categories. It also displays the Pro/TOOLKIT objects and functions as subnodes of the categories.

In Objects mode, this frame displays an alphabetical list of the Pro/TOOLKIT objects. It also displays Pro/TOOLKIT functions as subnodes of the objects.

In User's Guide mode, this frame displays the Pro/TOOLKIT User's Guide table of contents in a tree structure. All chapters are displayed as subnodes of the main Pro/TOOLKIT User's Guide node.

The Topic/Object/Category Selection frame includes a **Find** button for data searches of the Pro/TOOLKIT User's Guide or of API specifications taken from header files. See the section APIWizard Search Feature (Find) for more information on the Find feature.

Display Frame

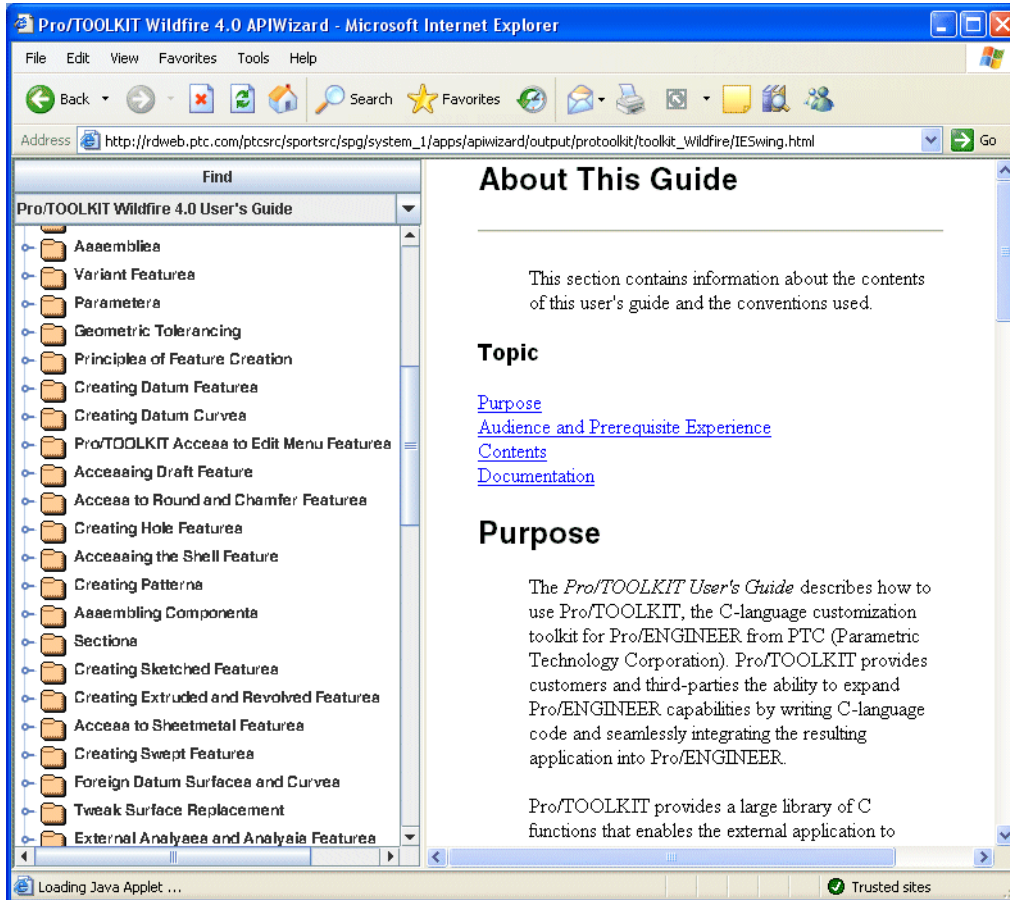
This frame, located on the right of the screen, presents:

- User's Guide content

- Pro/TOOLKIT object and object hierarchy descriptions
- Pro/TOOLKIT function descriptions
- Code examples for Pro/TOOLKIT functions

The following figure shows the APIWizard interface layout.

Figure 1-1: APIWizard Interface Layout



Navigating the Topic/Object/Category Selection Tree

Use the Topic/Object/Category Selection frame to access all Pro/TOOLKIT APIWizard online documentation data for objects, functions, or the Pro/TOOLKIT *Users' Guide*. This frame displays a tree structure of the data. Expand and collapse the tree to navigate this data.

To expand the tree structure, first select Pro/TOOLKIT **Categories**, Pro/TOOLKIT **Objects**, or Pro/TOOLKIT **User's Guide** at the top of the frame. The APIWizard displays the tree structure for the categories, objects, or User's Guide in collapsed form. The **Switch** icon to the far left of a node (that is, a category, an object, or a chapter name) signifies that this node contains subnodes. If a node has no **Switch** icon, it has no subnodes. Clicking the **Switch** icon (or double-clicking the node text) moves the switch to the down position. The APIWizard then expands the tree to display the subnodes. Select a node or subnode, and the APIWizard displays the online data in the Display frame.

Browsing the Pro/TOOLKIT Categories

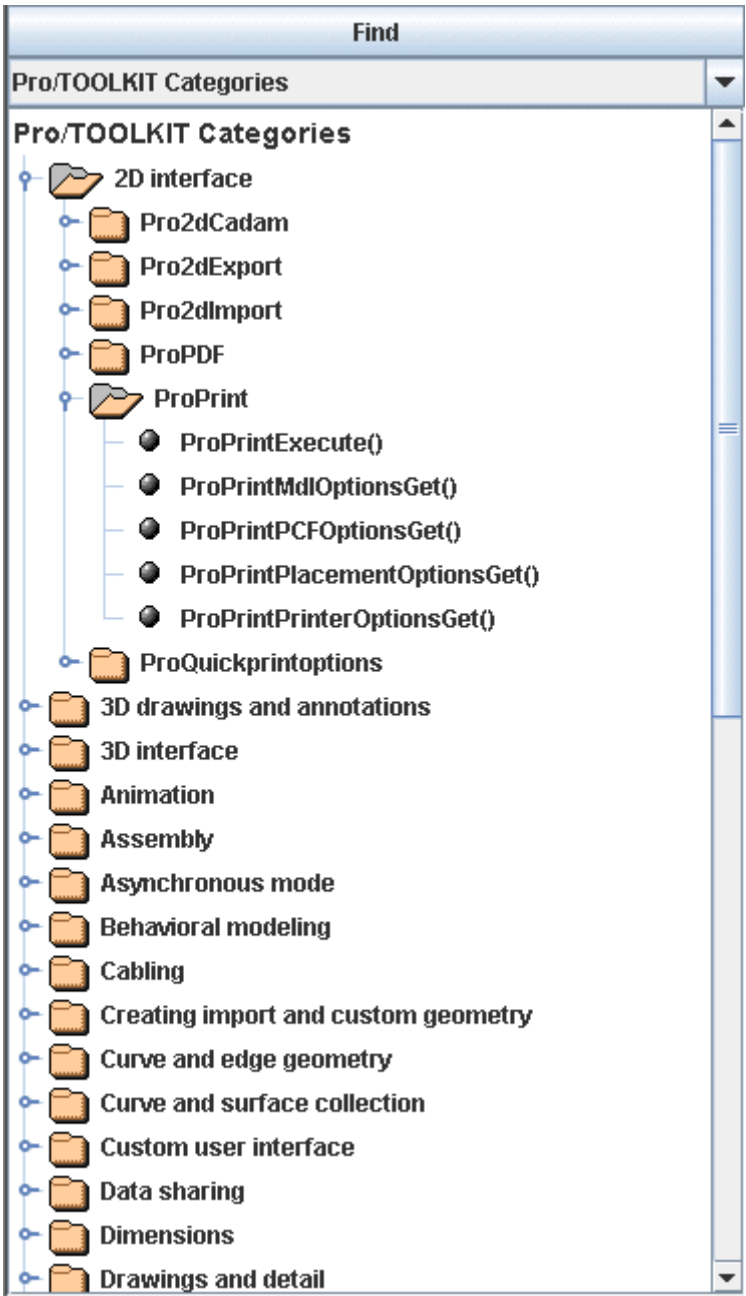
View the Pro/TOOLKIT categories by choosing Pro/TOOLKIT **Categories** at the top of the Topic/Object/Category frame. In this mode, the APIWizard displays the Pro/TOOLKIT categories in alphabetical order.

The definition page for each Pro/TOOLKIT category displays all the objects and functions that belong to that category.

Click the **Switch** icon next to the desired category name, or double-click the category name text to view the objects that belong to that category. You can also view the functions for each object in the expanded tree by clicking the **Switch** icon next to the object name, or by double-clicking the object name.

The following figure shows the collapsed tree layout for the Pro/TOOLKIT categories.

Figure 1-2: Pro/TOOLKIT Categories



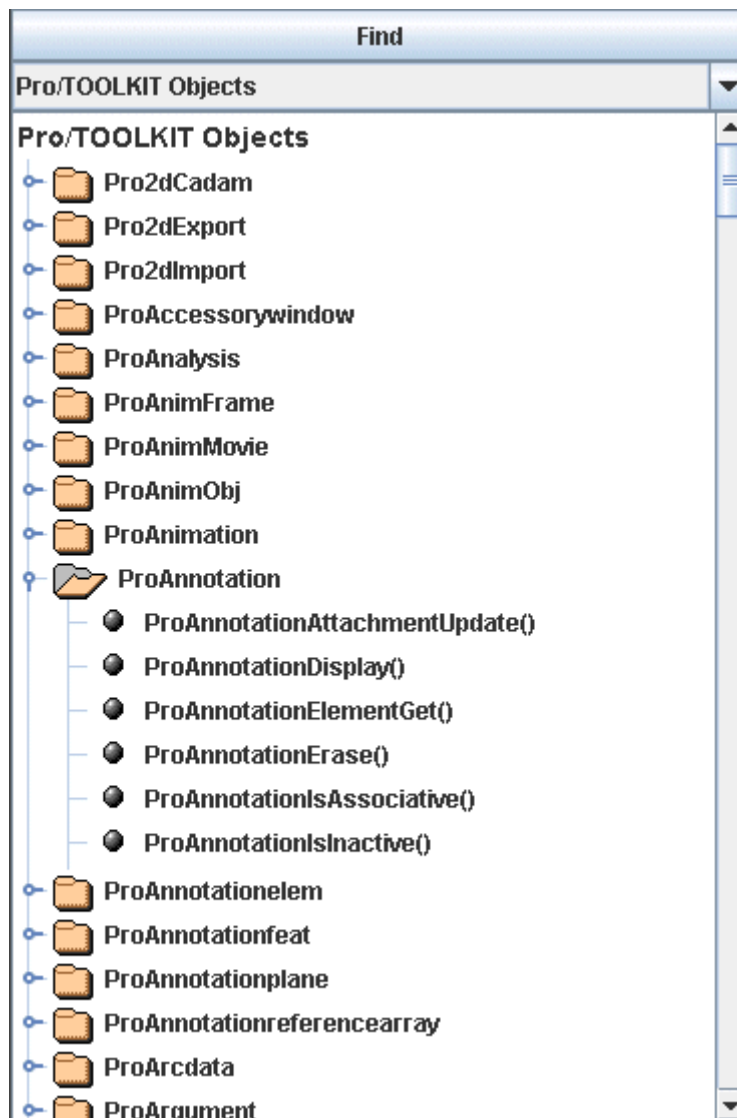
Fundamentals

Browsing the Pro/TOOLKIT Objects

View Pro/TOOLKIT objects by choosing Pro/TOOLKIT **Objects** at the top of the Topic/Object/Category Selection frame. In this mode, the APIWizard displays Pro/TOOLKIT objects in alphabetical order.

The following figure shows the Pro/TOOLKIT object tree layout.

Figure 1-3: Pro/TOOLKIT Object Tree Layout



The definition page displayed for each toolkit object contains all hierarchical information for that object. Each object displays data on related superobjects, subobjects, attributes, and inherited functions.

- Superobjects—Objects from which the current object inherits attributes and functions
- Subobjects—Objects which inherit attributes and functions from the current object
- Attributes—Pro/TOOLKIT objects contained in the current object
- Inherited functions—All functions inherited from Pro/TOOLKIT superobjects. For example, the ProAssembly object lists functions inherited from ProSolid and ProMdl.
- User's Guide references—Links to instances of this object name in the Pro/TOOLKIT *Users' Guide*.

View Pro/TOOLKIT functions by clicking the **Switch** icon to the left of an object name or by double-clicking the object name. The APIWizard then displays functions for that object as shown the preceding figure Pro/TOOLKIT Object Tree Layout.

Browsing the Pro/TOOLKIT User's Guide

View the Pro/TOOLKIT *Users' Guide* by clicking Pro/TOOLKIT **Users' Guide** at the top of the Topic/Object/Category Selection frame. In this mode, the APIWizard displays the chapter headings of the User's Guide.

View a chapter by clicking the **Switch** icon next to the desired section name or by double-clicking the chapter name. The APIWizard then displays a tree of sections under the selected chapter. The text for the selected chapter and its sections appear in the Display frame. Click the **Switch** icon again (or double-click the node text) to collapse the subnode listing and display only the main nodes.

The following figure shows how the Display frame presents the collapsed tree layout for the Pro/TOOLKIT *Users' Guide* table of contents.

Figure 1-4: Table of Contents

The image shows a screenshot of a software window titled "Find". Inside the window, there is a section titled "Pro/TOOLKIT Wildfire 4.0 User's Guide" with a dropdown arrow on the right. Below this title, a list of topics is displayed, each preceded by a folder icon and a blue circular icon with a right-pointing arrow. The topics are: About This Guide, Fundamentals, Messages, Menus, User Interface Components, Creating Dashboards, Modes, Models, and Model Items, Solids and Parts, Graphics and Object Display, Geometry, Curve and Surface Collections, The Selection Object, Geometric References, Coordinate Systems and Transformations, Relations, Basic Access to Features, Assemblies, Variant Features, Parameters, Geometric Tolerancing, Principles of Feature Creation, Creating Datum Features, Creating Datum Curves, Pro/TOOLKIT Access to Edit Menu Features, Accessing Draft Feature, and Access to Round and Chamfer Features. A vertical scrollbar is on the right side of the list, and a horizontal scrollbar is at the bottom.

Find	
Pro/TOOLKIT Wildfire 4.0 User's Guide ▼	
Pro/TOOLKIT Wildfire 4.0 User's Guide ▲	
•	About This Guide
•	Fundamentals
•	Messages
•	Menus
•	User Interface Components
•	Creating Dashboards
•	Modes, Models, and Model Items
•	Solids and Parts
•	Graphics and Object Display
•	Geometry
•	Curve and Surface Collections
•	The Selection Object
•	Geometric References
•	Coordinate Systems and Transformations
•	Relations
•	Basic Access to Features
•	Assemblies
•	Variant Features
•	Parameters
•	Geometric Tolerancing
•	Principles of Feature Creation
•	Creating Datum Features
•	Creating Datum Curves
•	Pro/TOOLKIT Access to Edit Menu Features
•	Accessing Draft Feature
•	Access to Round and Chamfer Features

APIWizard Search Feature (Find)

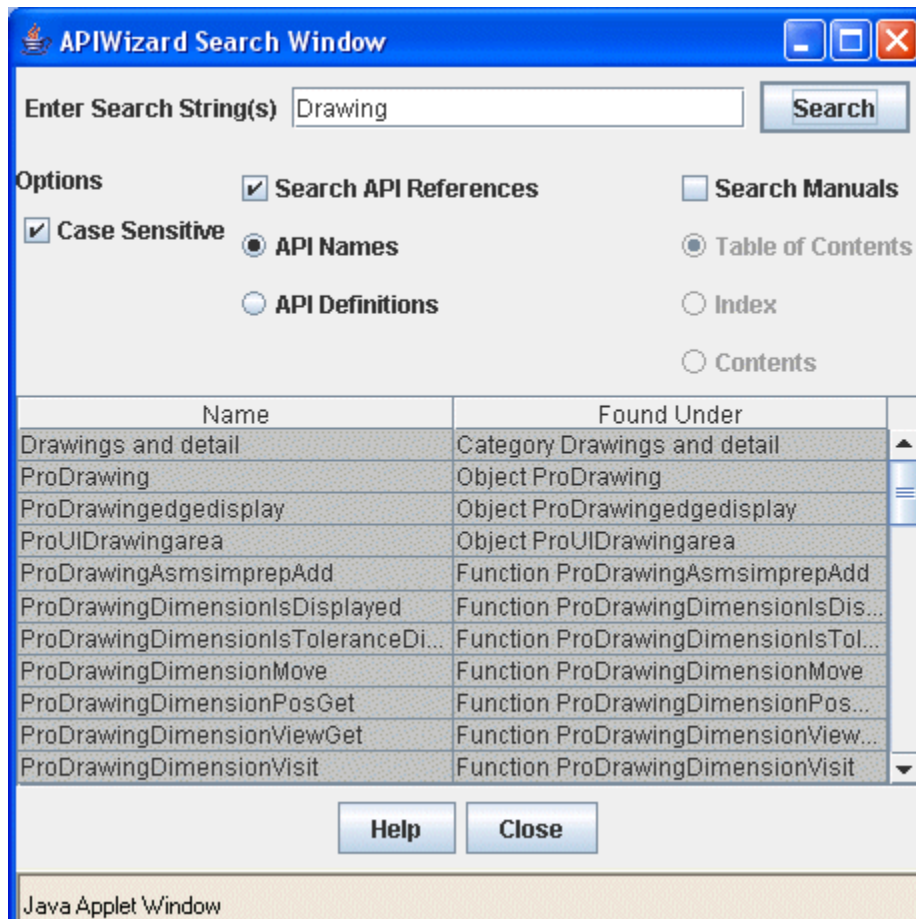
The APIWizard supports searches for specified strings against both the Pro/TOOLKIT User's Guide and API definition files.

Note: The APIWizard Search feature is slow when accessed through Internet Explorer's Default Virtual Machine. For better performance, access the APIWizard through Internet Explorer's Java2 plug-in.

Search Dialog

Click **Find** on the Topic/Object/Category Selection frame to display the APIWizard **Search** dialog box as shown in the following figure.

Figure 1-5: APIWizard Search Dialog Box



The **Search** dialog box contains the following fields, buttons and frames:

- **Enter Search String (s)**

Enter the specific search string or strings in this field. By default, the browser performs a noncase-sensitive search. The section Supported Search Types lists the supported search types and search string syntax. The section Performing an APIWizard Search describes search procedures.
- **Search/Stop**

Click **Search** to begin a search. During a search, this button name changes to Stop. Click **Stop** to stop the search.
- **Help**

Click **Help** for help about the APIWizard search feature. The APIWizard presents this help data in the Display frame.
- **Case Sensitive**

Select this button to specify a case-sensitive search.
- **Search API References**

Click **Search API References** to search for data on API functions. Click **API Names** to search for function names only. Click **API Definitions** to search the API function names and definitions for specific strings. The APIWizard displays function descriptions as shown in the following figure APIWizard Function Declaration Layout.
- **Search Manuals**

Click **Search Manuals** to search the Pro/TOOLKIT *Users' Guide* data. Click **Table of Contents** to search on Table of Contents entries only. Click **Index** to search only the Index. Click **Contents** to search on all text in the Pro/TOOLKIT *Users' Guide*.
- **Name**

This frame displays a list of strings found by the APIWizard search.
- **Found Under**

This frame displays the location in the online help data where the APIWizard found the string.

Supported Search Types

The APIWizard Search supports the following:

- Case-sensitive searches
- Search of API definitions, Pro/TOOLKIT User's Guide data, or both
- Search of API data by API names only or by API names and definitions
- Search of Pro/TOOLKIT User's Guide by Table of Contents only, by TOC and section titles, or on the User's Guide contents (the entire text)
- Wildcard searches—valid characters are:
 - * (asterisk) matches zero or more nonwhite space characters
 - ? (question mark) matches one and only one nonwhite space character

To match on an asterisk or a question mark character, place a \ (backslash) character before the desired character. For example, to search for `*wchar_t`, enter

```
\*wchar_t
```

To match on a double-quote character (") or a single-quote character ('), place a \ (backslash) character before it.

To search for any string containing the characters Pro, any number of other characters, and the characters Name

```
Pro*Name
```

To search for any string containing the characters Pro, one other character, and the characters Name

```
Pro?Name
```

To search for any string containing the characters Pro, one or more other characters, and the characters Name

```
Pro?*Name
```

To search on the string PRO, followed by an *

```
PRO\*
```

To search on the string PRO, followed by a ?

```
PRO\?
```

To search on the string PRO, followed by a \

```
PRO\
```

- Search strings containing white space— Search on strings that contain space characters (white space) by placing double- or single-quote characters around the string.

```
"asynchronous mode"  
'Pro*Data functions'
```

- Search on multiple strings—Separate multiple search strings with white space (tabs or spaces). Note that the default logical relationship between multiple search strings is OR.

To return all strings matching Pro*Get OR ProMdl*, enter:

```
Pro*Get ProMdl*
```

Note that this search specification also returns strings that match both specified search targets, for example:

```
ProMdlNameGet
```

If a string matches two or more search strings, the APIWizard displays only one result in the search table, for example:

```
Pro*Mdl Pro*Session
```

returns only one entry for ProSessionMdlListGet.

Mix quoted and nonquoted strings as follows:

```
Pro*Name "asynchronous mode"
```

This syntax returns all instances of strings containing Pro and Name, or strings containing asynchronous mode.

Performing an APIWizard Search

Follow these steps to search for information in the APIWizard online help data:

1. Click **Find** at the top of the Topic/Object Selection frame.
2. Specify the string or strings to be searched for in the **Enter Search String** field.
3. Click **Case Sensitive** to specify a case-sensitive search.
Note: The default search is non case-sensitive.
4. Click either or both of the **Search API References** and **Search User's Guide** buttons. Select the desired options under these buttons.
5. Click **Search**. The APIWizard turns this button red and is renamed it **Stop** for the duration of the search. Click **Stop** to stop a search.

6. If the APIWizard finds the search string in the specified search area or areas, it displays the string in the Name frame. In the Where Found frame, the APIWizard displays links to the online help data that contains the found string.
7. During the search or after the search ends, select an entry in the Name or Where Found frames to display the online help data for that string. The APIWizard first updates the Topic/Object Selection frame and tree, and then presents in the Display frame the online help data for the selected string.

Pro/TOOLKIT Style

Pro/TOOLKIT uses an object-oriented programming style. Data structures for the transfer information between Pro/ENGINEER and the application are not directly visible to the application. These data structures are accessible only with Pro/TOOLKIT functions.

Objects and Actions

The most basic Pro/TOOLKIT concepts are objects and actions.

Each Pro/TOOLKIT library C function performs an action on a specific type of object. The Pro/TOOLKIT convention for function names is the prefix “Pro” the name of the object type, and the name of the action it performs, for example:

ProSectionLocationGet()

A Pro/TOOLKIT object is a well-defined and self-contained C structure used to perform actions relevant to that object. Most objects are items in the Pro/ENGINEER database, such as features and surfaces. Others, however, are more abstract or transient, such as the information resulting from a select action.

In Pro/TOOLKIT, each type of object has a standard name consisting of a “Pro” plus capitalized word that describes the object. Simple examples of Pro/TOOLKIT object types and their Pro/ENGINEER equivalents are as follows:

- *ProFeature*—A feature
- *ProSurface*—A surface
- *ProSolid*—An abstract object created to exploit the commonality between parts and assemblies
- *ProWcell*—A workcell feature in a manufacturing assembly

Pro/TOOLKIT provides a C typedef for each object used for variables and arguments that refer to those objects. Pro/TOOLKIT objects have a hierarchical relationship that reflects the Pro/ENGINEER database. For example, a *ProFeature* object can contain objects of type *ProSurface* (among others).

For example, the following functions have actions that are single verbs:

ProSolidRegenerate()

ProFeatureDelete()

Some Pro/TOOLKIT functions require names that include more than one object type. The function names have the object types first, then the action. For example:

ProFeatureParentsGet()

ProWcellTypeGet()

The action verbs indicate the type of action being performed, as shown in the following table.

Action Verb	Type of Action
Get	Read information directly from the Pro/ENGINEER database.
Eval	Provide the result of a simple calculation.
Compute	Provide the result of a computation that typically involves numerical analysis of the model geometry.

Examples are:

- **ProEdgeLengthEval()**
- **ProSurfaceAreaEval()**
- **ProSolidRayIntersectionCompute()**

To illustrate further, function **ProSolidOutlineGet()** reads from Pro/ENGINEER the currently stored solid outline, but **ProSolidOutlineCompute()** invokes a recomputation of that information. Use **ProSolidOutlineCompute()** to compute an accurate outline of a solid.

Note: Do not use **ProSolidOutlineGet()** to calculate the outline of a solid. It will not return a properly calculated outline.

Other Pro/TOOLKIT function conventions are that the first argument identifies the object, and input arguments come before output arguments.

Function Prototyping

Each Pro/TOOLKIT function has an ANSI function prototype. (The C compilers on platforms supported by Pro/TOOLKIT provide at least the option of function prototype checking.) All function prototypes for a particular Pro/TOOLKIT object reside in a header file named for that object. For example, the prototype for function **ProEdgeLengthEval()** is located in the header file *ProEdge.h*.

Note: PTC strongly recommends that you use prototyping. Make sure you include the appropriate header files in your Pro/TOOLKIT application.

Function Error Statuses

The return type of most Pro/TOOLKIT functions is *ProError*. *ProError* is an enumerated type with a value for each common case where Pro/TOOLKIT functions succeeds or fails.

The normal value for success is PRO_TK_NO_ERROR. The other “failure” statuses occur when there is a genuine problem, or for more benign reasons. For example, these error statuses denote genuine problems:

- PRO_TK_BAD_INPUTS—The Pro/TOOLKIT program called the function incorrectly.
- PRO_TK_OUT_OF_MEMORY or PRO_TK_COMM_ERROR—System failure.

The following statuses are more benign:

- PRO_TK_USER_ABORT—A function that supports user interaction was aborted by the Pro/ENGINEER user.
- PRO_TK_E_NOT_FOUND—A function attempted operation on an empty object list.

Users must pay careful attention to how their program reacts to a Pro/TOOLKIT function error status—there can be several types of failure and success, each requiring different handling.

The subset of *ProError* values that a particular Pro/TOOLKIT function can return is described in the browser under that function. Possible errors are also included in a comment under each function prototype in the corresponding Pro/TOOLKIT header file.

Installing Pro/TOOLKIT

The next sections describe how to install Pro/TOOLKIT.

Overview

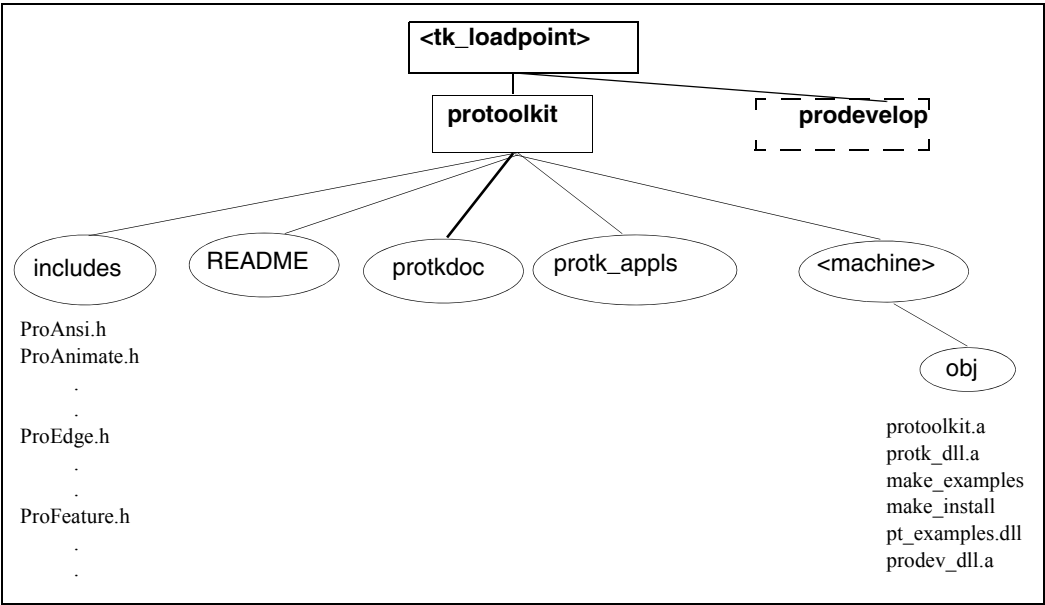
Pro/TOOLKIT is on the same CD as Pro/ENGINEER, so you do not need to arrange a special delivery from your supplier. When Pro/ENGINEER is installed using PTC.Setup, one of the optional components is “API Toolkits”. This includes Pro/TOOLKIT, Pro/WebLink, J-Link, and VB.

If you select Pro/TOOLKIT, it is installed automatically under the loadpoint of Pro/ENGINEER. Two directories are added under the chosen Pro/ENGINEER load point:

- `protoolkit`—Contains all the headers, libraries, example applications, and documentation specific to Pro/TOOLKIT.
- `prodevelop`—Contains the equivalent files for Pro/DEVELOP: the Pro/ENGINEER API until Revision 17. This directory allows support of Pro/TOOLKIT applications which continue to use Pro/DEVELOP functions.

The following figure shows the tree of directories found under the Pro/TOOLKIT loadpoint after installation.

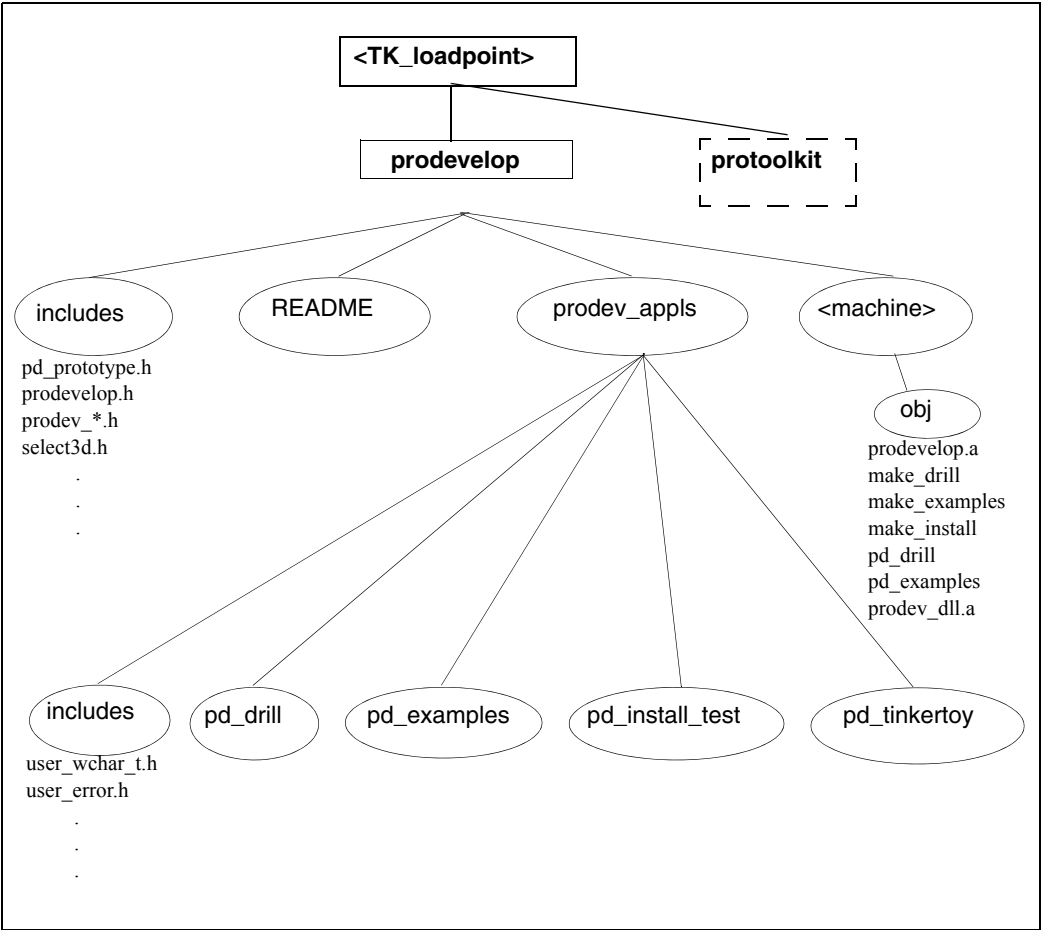
Figure 1-6: Pro/TOOLKIT Installation Directory Tree



The directory *protk_appls* contains sample Pro/TOOLKIT applications. For more information regarding the sample applications refer to the Appendix on Sample Applications.

The following figure shows the tree of directories found under the Pro/DEVELOP loadpoint after installation.

Figure 1-7: Pro/DEVELOP Installation Directory Tree



Add or Update Pro/TOOLKIT Installation

Add a Pro/TOOLKIT installation to an existing Pro/ENGINEER installation using the Update option in PTC.Setup. For a description of using PTC.Setup, refer to the *Pro/ENGINEER Installation and Administration Guide*.

Be sure your system administrator reinstalls Pro/TOOLKIT each time they update your Pro/ENGINEER installation from a new CD. PTC recommends that, when possible, you use a Pro/TOOLKIT from the same build number as Pro/ENGINEER.

Note: The Pro/ENGINEER library functions work by invoking functions inside the Pro/ENGINEER executable, so an update to Pro/TOOLKIT often involves a change to Pro/ENGINEER rather than Pro/TOOLKIT itself. So when you receive a Pro/ENGINEER CD that contains an update to Pro/TOOLKIT, always reinstall Pro/ENGINEER from that CD.

In many situations it will be inconvenient or impossible to ensure that the users of your Pro/TOOLKIT application will use the same build of Pro/ENGINEER that you used to compile and link the Pro/TOOLKIT application. Refer to section “Version Compatibility: Pro/ENGINEER and Pro/TOOLKIT” for the rules to mix versions of Pro/ENGINEER and Pro/TOOLKIT.

Testing the Pro/TOOLKIT Installation

After your system administrator has installed Pro/TOOLKIT, you should compile, link, and run a simple Pro/TOOLKIT application as soon as possible on each machine you intend to use for development. This provides an independent test of the following items:

- The installation of Pro/TOOLKIT is present, complete, and visible from your machine.
- The version of Pro/ENGINEER you plan to use during development has the Pro/TOOLKIT license option added to it.
- The machine you will use for development has access to all the necessary C program development tools, in versions supported by Pro/TOOLKIT (especially, of course, the C compiler and linker).

Running the Microsoft Visual Studio Solution

PTC provides a ready-to-use Visual Studio solution on the Windows platform to build and test Pro/TOOLKIT applications by using an appropriate makefile. For the version of Visual Studio compatible with the release of Pro/TOOLKIT, refer to the hardware notes at <http://www.ptc.com/partners/hardware/current/support.htm>.

This ready-to-use Visual Studio solution has the following advantages:

- Provides an effective way to build and test sample applications provided by PTC.
- Provides a preconfigured Visual Studio development environment for use with Pro/TOOLKIT.
- Supports **Intellisense** for Pro/TOOLKIT functions.

Note: The supported version of Visual Studio changes with every release of Pro/TOOLKIT, and hence the compiler flags and libraries also change. For every release, you must download the latest version of the ready-to-use Visual Studio solution from the Pro/TOOLKIT loadpoint.

When you install Pro/TOOLKIT, the file *protk_install_example.zip* is installed under the Pro/TOOLKIT load point at `protoolkit/<machine_type>/obj`. To use this solution:

1. Unzip *protk_install_example.zip*. The following files and directories are available:

Directory or File	Description
<i>make_install.sln</i>	Specifies the ready-to-use Visual Studio solution file.
<i>make_install</i>	Contains the makefile project and the <i>protk.dat</i> file.

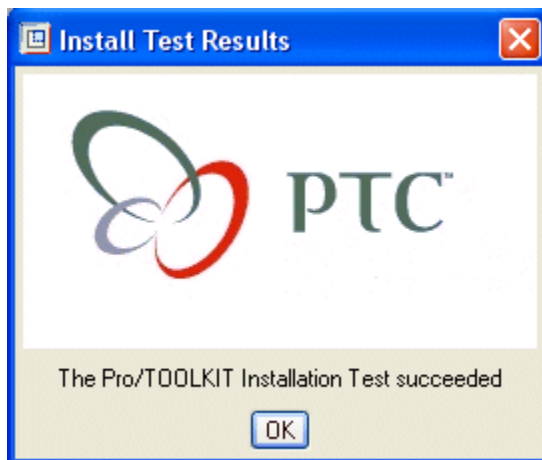
2. Set the environment variable `PROE_INSTALL_PATH` to point to the Pro/ENGINEER installation directory.
3. Open Microsoft Visual Studio.
4. Click **File>Open>Project/Solution**. The **Open Project** dialog opens.
5. Browse the *protk_install_example* directory and select *make_install.sln*.
6. Click **Open** to access the solution file.

The *make_install* makefile project is available in Visual Studio.

Running the Makefile Project

1. Click **Build>Build make_install**. The application should build without errors. This creates the Pro/TOOLKIT DLL file called *pt_inst_test.dll*. If the application fails, check that the environment variable `PROE_INSTALL_PATH` is set correctly.
2. Modify the *exec_file* and *text_dir* fields in the *protk.dat* file located in the *make_install* directory to specify the full path to *pt_inst_test.dll* and “\text”, respectively. For example,

```
exec_file <full_path>\pt_inst_test.dll  
text_dir <full_path>\text
```
3. Start Pro/ENGINEER and click **Tools>Auxiliary Applications>Register** to register the updated *protk.dat* file. The **Register Auxiliary application** dialog box opens.
4. Browse to the *<full_path>* and select *protk.dat*. The Pro/TOOLKIT application adds the command **Install Test** to the Pro/ENGINEER **File** menu.
5. Click **File>Install Test**. The Pro/TOOLKIT **Install Test Results** message window opens, indicating that the installation test has succeeded.



6. Click **OK**.

To run other sample applications provided by PTC, follow these steps:

1. Copy the required makefile from `<proe_loadpoint>/protoolkit/i486_nt/obj` to the `make_install` directory of the ready-to-use Visual Studio solution.

If you are working on a 64-bit Windows platform, copy the file from `<proe_loadpoint>/protoolkit/x86e_win64/obj`.
2. Copy the text directory associated with the sample application from `<proe_loadpoint>/protoolkit` to the `make_install` directory.
3. Open Visual Studio and set the values of the following variables in the makefile:

```
PROTOOL_SRC = $(PROE_INSTALL_PATH)/protoolkit
PROTOOL_SYS = $(PROTOOL_SRC)/$(PRO_MACHINE_TYPE)
PRODEV_SRC = $(PROE_INSTALL_PATH)/prodevelop
PRODEV_SYS = $(PROTOOL_SRC)/$(PRO_MACHINE_TYPE)
```
4. Click **Project>Properties** to update the **NMake** properties of the project.
5. Click **Build>Rebuild make_install**. The application builds and creates a new `.dll` file.
6. Update the `protk.dat` file located in the `make_install` directory with the name of the sample application and the DLL file.
7. Modify the `exec_file` and `text_dir` fields in the `protk.dat` file to specify the full path to the `.dll` file and “\text” directory, respectively.
8. Start Pro/ENGINEER and click **Tools>Auxiliary Applications>Register** to register the updated `protk.dat` file. The **Register Auxiliary application** dialog box opens.
9. Browse to the full path and select `protk.dat`. The Pro/TOOLKIT application runs.

Building a Sample Application

The Pro/TOOLKIT loadpoint includes the source of a simple application designed specifically to test the Pro/TOOLKIT installation. The steps required to build and run the test application are described in the following sections.

In this explanation, <TK_LOADPOINT> refers to the directory that forms the loadpoint of Pro/TOOLKIT, and <MACHINE> refers to the name of the type of platform you are using (for example, i486_nt).

Step 1—Compile and Link

Compile and link the Pro/TOOLKIT installation test application using the following makefile:

```
<TK_LOADPOINT>/<MACHINE>/obj/make_install
```

On Windows systems, the makefile is in the equivalent location, and is intended for nmake instead of make.

The makefile is designed to be run in that location, and creates a Pro/TOOLKIT application file in the directory from which it is run. If you do not have root privileges, you probably need to copy the makefile to a directory of your own so the output file can be created. If you do this, you also need to edit the makefile to correct the macro that refers to the Pro/TOOLKIT loadpoint.

If you copy the makefile to another directory, replace the line:

```
PROTOOL_SRC = ../..
```

with:

```
PROTOOL_SRC = <TK_LOADPOINT>
```

In this line, <TK_LOADPOINT> is the loadpoint for your Pro/TOOLKIT installation.

To run the makefile, type the following command:

```
make -f make_install
```

This creates the Pro/TOOLKIT executable file called *pt_inst_test*.

If you are using a Windows system, type the following command:

```
nmake -f make_install
```

This creates a file called *pt_inst_test.exe*.

If you experience any error messages at this stage, it might be due to the Pro/TOOLKIT installation being incomplete, or, more likely, the C compiler and linker being unavailable or unsupported by Pro/TOOLKIT.

Step 2—Register

In the same directory, create a text file called *protk.dat*. This file is the “registry file” that tells Pro/ENGINEER about the Pro/TOOLKIT application. Refer to the “Registering a Pro/TOOLKIT Application” and “Sample Registry Files” sections for syntax requirements for this file. The *protk.dat* file should contain the following lines:

```
name install_test
exec_file pt_inst_test
text_dir <TK_LOADPOINT>/protk_appls/pt_install_test
end
```

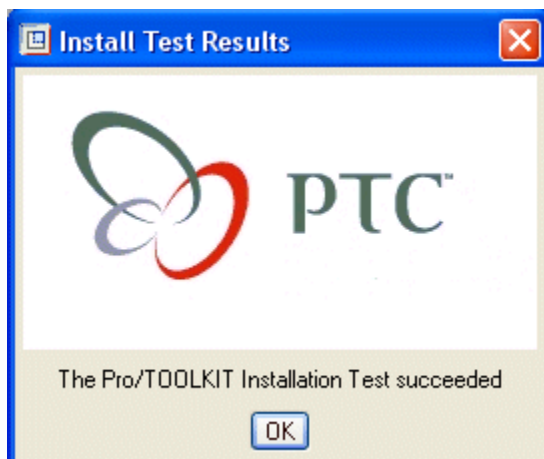
Note: For delimiter characters in *protk.dat*, use *'* on UNIX platforms and ** on Windows NT platforms.

On Windows systems, use the executable name *pt_inst_test.exe* in the second line, and the Windows directory syntax in the third line.

Step 3—Run Pro/ENGINEER

Run Pro/ENGINEER from the directory that contains the *protk.dat* file; Pro/ENGINEER starts the Pro/TOOLKIT application in multiprocess mode (see the section “How Pro/TOOLKIT Works” for more information on multiprocess mode). You should see that the Pro/ENGINEER FILE menu has a new button, added by the Pro/TOOLKIT application, called “**Install Test**”. When you choose this button, the Pro/TOOLKIT application displays a custom dialog indicating whether the installation test has succeeded:

Figure 1-8: Install Test Results Dialog Box



Failure or error messages at this stage could be due to the following reasons:

- You made a mistake when creating the *protk.dat* file. If the syntax or contents are wrong, you should see a self-explanatory message in the window from which you started Pro/ENGINEER.
- The Pro/ENGINEER you ran is not licensed for Pro/TOOLKIT. This also causes an explanatory message to be displayed in the startup window.
- The Pro/TOOLKIT executable you created in Step 1 is wrong in some way: it is for the wrong platform, for example, or might not have execute access. You can check this by trying to execute the file directly by typing its name. If the file is correct, the program prints the following messages and then terminates:

```
pt_inst_test: insufficient arguments; need 2 arguments:
(1) own RPC program #
(2) root directory path for Pro/TOOLKIT text files.
```

If the file is incorrect, the exact message will depend on which platform you are using, but should explain the cause of the problem.

Step 4—Repeat the Test in DLL Mode

To build for DLL mode, use the same makefile, but use the following line instead of the line “make -f make_install”:

```
make -f make_install dll
```

This creates a file called *pt_inst_test.dll*, which is the library to be dynamically linked.

Next, make these two changes to the *protk.dat* file:

Add this line after the first line:

```
startup dll
```

Change the *exec_file* statement to reference the new Pro/TOOLKIT file.

On UNIX systems, the file may now look something like this:

```
name install_test
startup dll
exec_file pt_inst_test.dll
text_dir <TK_LOADPOINT>/protk_appls/pt_install_test
end
```

Note: For delimiter characters in *protk.dat*, use *'* on UNIX platforms and ** on Windows NT platforms.

On Windows systems, use the executable name *pt_inst_test.exe* in the second line, and the Windows directory syntax in the third line.

You can run Pro/ENGINEER and look at the behavior of the Pro/TOOLKIT application exactly as in Step 3.

See the section “How Pro/TOOLKIT Works” for more information on DLL mode.

Developing a Pro/TOOLKIT Application

This section describes how Pro/TOOLKIT works and the steps you need to take after installing Pro/TOOLKIT to create a Pro/TOOLKIT application. The topics are as follows:

- How Pro/TOOLKIT Works
- Compiling and Linking a Pro/TOOLKIT Application
- Registering a Pro/TOOLKIT Application
- Version Compatibility: Pro/ENGINEER and Pro/TOOLKIT
- Stopping and Restarting a Pro/TOOLKIT Application
- Structure of a Pro/TOOLKIT Application
- User-Supplied Main
- Pro/TOOLKIT Techniques

How Pro/TOOLKIT Works

The standard method by which Pro/TOOLKIT application code is integrated into Pro/ENGINEER is through the use of dynamically linked libraries (DLLs). When you compile your Pro/TOOLKIT application C code and link it with the Pro/TOOLKIT library, you create an object library file designed to be linked into the Pro/ENGINEER executable when Pro/ENGINEER starts up. This method is referred to as “DLL mode.”

Pro/TOOLKIT also supports a second method of integration: the “multiprocess,” or spawned mode. In this mode, the Pro/TOOLKIT application code is compiled and linked to form a separate executable. This executable is designed to be spawned by Pro/ENGINEER and runs as a child process of the Pro/ENGINEER session. In DLL mode, the exchanges between the Pro/TOOLKIT

application and Pro/ENGINEER are made through direct function calls. In multiprocess mode, the same effect is created by an inter-process messaging system that simulates direct function calls by passing the information necessary to identify the function and its argument values between the two processes.

Multiprocess mode involves more communications overhead than DLL mode, especially when the Pro/TOOLKIT application makes frequent calls to Pro/TOOLKIT library functions, because of the more complex method of implementing those calls. However, it offers the following advantage: it enables you to run the Pro/TOOLKIT application with a source-code debugger without also loading the whole Pro/ENGINEER executable into the debugger. See the section Using a Source-Code Debugger on a Pro/TOOLKIT Application for more details.

You can use a Pro/TOOLKIT application in either DLL mode or multiprocess mode without changing any of the C source code in the application. (The methods of setting the mode are described in detail later in this chapter.)

It is also possible to use more than one Pro/TOOLKIT application within a single session of Pro/ENGINEER, and these can use any combination of modes.

If you use multiprocess mode during development of your application to debug more easily, you should switch to DLL mode when you install the application for your end users because the performance is better in that mode. However, take care to test your application thoroughly in DLL mode before you deliver it. Any programming errors in your application that cause corruption to memory used by Pro/ENGINEER or Pro/TOOLKIT are likely to show quite different symptoms in each mode, so new bugs may emerge when you switch to DLL mode.

Although multiprocess mode involves two processes running in parallel, these processes do not provide genuine parallel processing. There is, however, another mode of integrating a Pro/TOOLKIT application that provides this ability, called “asynchronous mode.” (Asynchronous mode is described in detail in the chapter Core: Asynchronous Mode.) The DLL and multiprocess modes are given the general name “synchronous mode.” An asynchronous Pro/TOOLKIT application is fundamentally different in its architecture from a synchronous mode application, so you should choose between these methods before writing any application code. As a general rule, synchronous mode should be the default choice unless there is some unavoidable reason to use asynchronous mode, because the latter mode is more complex to use.

Note: All Pro/TOOLKIT calls running in either synchronous (DLL or multiprocess) mode or asynchronous mode always clear the Undo/Redo stack in the Pro/ENGINEER session. The Pro/ENGINEER user interface reflects this by making the **Undo** and **Redo** menu options unavailable.

Compiling and Linking a Pro/TOOLKIT Application

This section describes compiling and linking Pro/TOOLKIT applications.

Makefiles

The C compiler options and system libraries needed to compile and link a Pro/TOOLKIT application are generally different on each platform, even between different varieties of UNIX. To ensure that the makefile you use for building your Pro/TOOLKIT application uses the correct options, you should base your makefile on one of the makefiles located under the Pro/TOOLKIT loadpoint. These are designed for building the various Pro/TOOLKIT applications whose source is included in the Pro/TOOLKIT installation.

An example of one of the Pro/TOOLKIT applications provided is the installation test, whose source code is under the directory `<TK_LOADPOINT>/protk_appls/pt_install_test`, where `<TK_LOADPOINT>` is the loadpoint directory of the Pro/TOOLKIT installation. The makefile for the installation test application is `<TK_LOADPOINT>/<MACHINE>/obj/make_install`.

To use this as the model for your own makefile, copy it to the directory that will contain your Pro/TOOLKIT source code, then make the following changes to it:

- Change the macro `MAKEFILENAME` to refer to the makefile by its new name.
- Change the macros `EXE` and `EXE_DLL` to define output file names more suitable for your own application.
- Change the macro `PROTOOL_SRC` to refer to the loadpoint of Pro/TOOLKIT.
- Change the macro `OBJS` to refer to the object files that will result from compiling your Pro/TOOLKIT source files.
- Add targets for those object files. These contain instructions for compiling your C source files. The form of these target definitions can be copied from the ones in the original makefile. They generally take the following form:

```
myfile.o: myfile.c
$(CC) $(CFLAGS) myfile.c
```

Note: The second line *must* start with a tab character.

If you want to use a debugger with your Pro/TOOLKIT application, you can also add the appropriate compiler switch (usually “-g”) to the CCFLAGS macro.

If you need further explanation of how to use makefiles, refer to the documentation supplied with your computer system, or a textbook on UNIX systems.

If you are rebuilding an existing Pro/TOOLKIT application with a new version of Pro/TOOLKIT, remember to repeat these steps to set up a *new* makefile—do not continue to use a makefile created for the previous version. You must do this in case the compiler switches or system libraries needed to build a Pro/TOOLKIT application have changed in the new version.

Registering a Pro/TOOLKIT Application

Registering a Pro/TOOLKIT application means providing information to Pro/ENGINEER about the files that form the Pro/TOOLKIT application. To do this, create a small text file, called the Pro/TOOLKIT registry file, that Pro/ENGINEER will find and read.

Pro/ENGINEER searches for the registry file as follows:

- In the absolute path specified in the “PROTKDAT”, “PRODEV DAT”, and “TOOLKIT_REGISTRY_FILE” statements in the Pro/ENGINEER configuration file.
- For the files named *protk.dat* or *prodev.dat* in the following locations:
 1. Current directory
 2. <Pro/ENGINEER>/<MACHINE>/text
 3. <Pro/ENGINEER>/text

In the preceding locations, the variables are as follows:

- <Pro/ENGINEER>—The Pro/ENGINEER loadpoint (*not* the Pro/TOOLKIT loadpoint).
- <MACHINE>—The machine-specific subdirectory (such as i486_nt).

If more than one registry file having the same filename exists in this search path, Pro/ENGINEER stops searching after finding the first instance of the file and starts all the Pro/TOOLKIT applications specified in it. If more than one registry file having different filenames exist in this search path, Pro/ENGINEER stops searching after finding one instance of each of them and starts all the Pro/TOOLKIT applications specified in them.

Note:

- Option 1 is normally used during development, because the Pro/TOOLKIT application is seen only if you start Pro/ENGINEER from the specific directory that contains the registry file.
- Option 3 is recommended when making an end-user installation, because it makes sure that the registry file is found no matter what directory is used to start Pro/ENGINEER.

The registry file is a simple text file, where each line consists of one of a predefined set of keywords, followed by a value.

The standard form of the registry file in DLL mode is as follows:

```
name YourApplicationName
startup dll
exec_file $LOADDIR/$MACHINE_TYPE/obj/filename.dll
text_dir $LOADDIR
end
```

The fields of the registry file are as follows:

- *name*—Assigns a unique name to this Pro/TOOLKIT application.
- *startup*—Specifies the method Pro/ENGINEER should use to communicate with the Pro/TOOLKIT application. The example above specifies the DLL mode.
- *exec_file*—Specifies the full path and name of the file produced by compiling and linking the Pro/TOOLKIT application. The example above shows a typical use of environment variables to make the reference to the executable file more flexible.
- *text_dir*—Specifies the full path name to text directory that contains the language-specific directories. The language-specific directories contain the message files, menu files, resource files and UI bitmaps in the language supported by the Pro/TOOLKIT application.

Note: The fields *exec_file* and *text_dir* have a character limit of `PRO_PATH_SIZE-1` wide characters (`wchar_t`).

- *unicode_encoding*—Allows a Pro/TOOLKIT synchronous mode application to execute in the legacy string encoding format in the Pro/ENGINEER Wildfire 4.0 environment. The default value of this option is `true`, indicating that any application will execute using Unicode encoding by default.
- *end*—Indicates the end of the description of this Pro/TOOLKIT application.

If you want to run the application in multiprocess mode, make the following changes to the registry file:

- Change the *startup* statement to:

```
startup spawn
```
- Make the *exec_file* statement refer to the Pro/TOOLKIT program executable.

Note: For more information about the registry file, refer to the Appendix Pro/TOOLKIT Registry File.

Limit on the Number of Loaded Applications

Previous versions of Pro/ENGINEER limited the number of applications that could be specified in the registry files; there is no such limit for Pro/ENGINEER Wildfire 2.0. However, most platforms do have limits for the size of a process, the total size of all processes, and the number of processes that a single process can spawn. PTC recommends that you combine related applications into the same binary file wherever possible to avoid running into these limits.

Version Compatibility: Pro/ENGINEER and Pro/TOOLKIT

In many situations it will be inconvenient or impossible to ensure that the users of your Pro/TOOLKIT application use the same build of Pro/ENGINEER used to compile and link the Pro/TOOLKIT application. This section summarizes the rules for mixing Pro/TOOLKIT and Pro/ENGINEER versions. The Pro/TOOLKIT version is the Pro/ENGINEER CD version from which the user installed the Pro/TOOLKIT version used to compile and link the application.

Note: From Pro/ENGINEER Wildfire 4.0 onwards applications built with libraries older than Pro/ENGINEER 2001 will not run. You must recompile these applications with later versions of the Pro/TOOLKIT libraries.

- Pro/ENGINEER release older than Pro/TOOLKIT release:
Not supported
- Pro/ENGINEER release newer than a Pro/TOOLKIT release:
This works in many, but not all, cases. The communication method used to link Pro/TOOLKIT to Pro/ENGINEER provides full compatibility between releases. However, there are occasional cases where changes internal to Pro/ENGINEER may require changes to the source code of a Pro/TOOLKIT application in order that it continue to work correctly. Whether you need to convert Pro/TOOLKIT applications depends on what functionality it uses and what functionality changed in Pro/ENGINEER and Pro/TOOLKIT. PTC makes every effort to keep these effects to a minimum. The Release Notes for Pro/TOOLKIT detail any conversion work that could be necessary for that release.
- Pro/ENGINEER build older than Pro/TOOLKIT build:
During Pro/TOOLKIT Release 20 and Release 2000i, this was not supported. From Release 2000i² it will generally be possible, unless your application calls a Pro/TOOLKIT function that was added or modified since the build of the Pro/ENGINEER being used. PTC will avoid modifications to Pro/TOOLKIT function definitions from Release 2000i² onwards and keep additions to a minimum.
- Pro/ENGINEER build newer than Pro/TOOLKIT build
This is always supported.

Application Compatibility: Pro/ENGINEER and Pro/TOOLKIT on Different Architecture

In some situations it will be inconvenient or impossible to ensure that the users of your Pro/TOOLKIT application use a machine with the same operating system and architecture as the machine on which it was compiled. An example might be an application integrating with a third party library which is only available as 32-bit architecture, but needs to be run with Pro/ENGINEER on a

64-bit architecture machine with the same operating system. Pro/TOOLKIT provides limited capability to support these situations in spawn and asynchronous mode only. DLL applications must always be compiled on machines with the same operating system and architecture as the Pro/ENGINEER executable.

The following situations might occur:

- Pro/TOOLKIT application compiled on the same architecture and operating system as Pro/ENGINEER. This is always supported.
- Pro/TOOLKIT application compiled on a machine with a smaller pointer size (native data size) than the machine on which the application is run. For example, a Pro/TOOLKIT application built on Windows 32-bit running on an Windows-64 bit installation of Pro/ENGINEER. This is supported for spawn and asynchronous mode only.
- Pro/TOOLKIT application compiled on a machine with a larger pointer size (native data size) than the machine on which the application is run. For example, a Pro/TOOLKIT application built on Windows-64 bit machine running on an Windows-32 bit installation of Pro/ENGINEER. This is not supported.

Stopping and Restarting a Pro/TOOLKIT Application

Pro/TOOLKIT supports the ability to stop and restart a synchronous application within a single session of Pro/ENGINEER. This is particularly useful during development of an application because it enables you to make changes to your source code and retest it without having to restart Pro/ENGINEER and reload your test models. Use the Auxiliary Applications dialog box to stop and restart applications.

To make this option available, the registry file (default name *protk.dat*) should contain one of the following lines:

Multiprocess mode:

startup spawn

DLL mode:

startup DLL

If you want to be able to stop and restart your Pro/TOOLKIT application within Pro/ENGINEER, you must also add the following statement to the definition of the application in the registry file:

allow_stop TRUE

To access the Auxiliary Applications dialog box, select the Tools menu from the Pro/ENGINEER menu bar, then choose **Auxiliary Applications**. The dialog box displays a list of Pro/TOOLKIT applications identified by the name defined in the *name* statement in its registry file. Only applications that a user can start or stop are displayed. This dialog box also shows the current state of an application and allows an application to be started or stopped.

When a user starts an application from the Auxiliary Applications dialog box, Pro/ENGINEER freezes the user interface until the application connects to it.

If you use the *allow_stop* option, you might also set Pro/ENGINEER to not start the Pro/TOOLKIT application until you explicitly request it. To do this, you must add the following statement in your registry file:

```
delay_start TRUE
```

To start your application in Pro/ENGINEER, choose **Auxiliary Applications** from the Tools menu, select your application from the list, then click the **Start** button.

In addition to **Start**, **Stop**, and **Close**, the dialog box includes the following buttons:

- **Register**—Enables you to register a Pro/TOOLKIT application whose registry file was not present when Pro/ENGINEER was started.
- **Info**—Reports the following information about each currently registered Pro/TOOLKIT application:
 - The names of the executable file and text directory
 - The version number used to build the application
 - Whether the application is currently running

There are a few other, less commonly used options in the registry file. All the options are described fully in the Pro/TOOLKIT Registry File appendix.

Note: You can delete registration information on any application that is not running.

When stopping an application, make sure that no application-created menu buttons are current. To do this, before you exit an application you must choose a command that interrupts the current menu command.

Using a Source-Code Debugger on a Pro/TOOLKIT Application

For a full description of how to debug Pro/TOOLKIT Applications on Linux, UNIX, and Windows systems, see Appendix 'Debugging Pro/TOOLKIT Applications'. This section also describes optional methods of debugging a Pro/TOOLKIT application.

Unlocking a Pro/TOOLKIT Application

Before you distribute your application executable to the end user, you must unlock it. This enables the end user (your customer) to run your applications without having Pro/TOOLKIT as an option.

To unlock your application, enter the following command:

```
<Pro/ENGINEER Loadpoint>/bin/protk_unlock <path to  
executables or DLLs to unlock>
```

More than one Pro/TOOLKIT binary file may be supplied on the command line.

Note: Once you have unlocked the executable, you can distribute your application program to Pro/ENGINEER users in accordance with the license agreement.

Using `protk_unlock` requires a valid Pro/TOOLKIT license be present and unused on your license server. If the Pro/ENGINEER license server is configured to add a Pro/TOOLKIT license as a Startup option, `protk_unlock` will cause the license server to hold only the Pro/TOOLKIT option for 15 minutes. The license will not be available for any other development activity or unlocking during this period.

If the only available Pro/TOOLKIT license is locked to a Pro/ENGINEER license, the entire Pro/ENGINEER license including the Pro/TOOLKIT option will be held for 15 minutes. PTC recommends you configure your Pro/TOOLKIT license option as startup options to avoid tying up your Pro/ENGINEER licenses.

Note: Only one license will be held for the specified time period, even if multiple applications were successfully unlocked.

Unlocking the application may also require one or more advanced licensing options. The `protk_unlock` application will detect whether any functions using advanced licenses are in use in the application, and if so, will make a check for the availability of the advanced license option. If that option is not present, unlocking will not be permitted. If they are present, the unlock will proceed. Advanced options are not held on the license server for any length of time. For more information refer to the “Advanced Licensing” chapter.

If the required licenses are available, the `protk_unlock` application will unlock the application immediately. An unlocked application does not require any of the Pro/TOOLKIT license options to run. Depending on the functionality invoked by the application, it may still require certain Pro/ENGINEER options to work correctly.

Note: Once an application binary has been unlocked, it should not be modified in any way (which includes statically linking the unlocked binary with other libraries after the unlock). The unlocked binary must not be changed or else Pro/ENGINEER will again consider it "locked".

Unlock Messages

The following table lists the messages that can be returned when you unlock a Pro/TOOLKIT application.

Message	Cause
<application name>:Successfully unlocked application.	The application is unlocked successfully.
Usage: <code>protk_unlock</code> <one or more Pro/TOOLKIT executables or DLLs>	No arguments supplied.
<application name>:ERROR: No READ access <application name>:ERROR: No WRITE access	You do not have READ/WRITE access for the executable.
<application name>:Executable is not a Pro/TOOLKIT application.	The executable is not linked with the Pro/TOOLKIT libraries, or does not use any functions from those libraries.
<application name>:Executable is already unlocked.	The executable is already unlocked.

Message	Cause
Error: Licenses do not contain Pro/TOOLKIT License Code.	A requirement for unlocking a Pro/TOOLKIT application.
ERROR: No Pro/ENGINEER licenses are available for the startup command specified	Could not contact the license server.
<application name>:Unlocking this application requires option Pro/TOOLKIT-for-3D_Drawings.	The application uses functions requiring an advanced option; and this option is not available.
<application name>:Unlocking this application requires option Pro/TOOLKIT-for-Mechanica.	The application uses functions requiring an advanced option; and this option is not available.

Structure of a Pro/TOOLKIT Application

The contents of this section refer to the use of synchronous mode. For information on asynchronous mode applications, see the chapter ‘Core: Asynchronous Mode’.

Essential Pro/TOOLKIT Include Files

The only header file you *must* always include in every source file of your Pro/TOOLKIT application is *ProToolkit.h*. This file must always be present, and must be the first include file because it defines the value of *wchar_t*, the type for characters in a wide string, referenced from many other include files. *ProToolkit.h* also includes these standard include files:

- *stdio.h*
- *string.h*
- *stddef.h*
- *stdlib.h*

Therefore, you do not need to include these header files explicitly in your application.

When you use functions for a particular Pro/TOOLKIT object, you should always include the header file that contains the function prototypes for those functions. If you do not do this, or omit some, you lose the benefit of function argument type-checking during compilation. The header file *ProObjects.h*, which contains the declarations of the object handles, is included indirectly in each of the header files that contains function prototypes, and so does not need to be included explicitly.

For example, if you are using the function **ProSurfaceAreaEval()**, you should include the file *ProSurface.h*, which contains the prototype of that function, but you do not need to include *ProObjects.h* in order to see the definition of *ProSurface*, because *ProObjects.h* is included in *ProSurface.h*.

Core of a Pro/TOOLKIT Application

Functions Introduced:

- **user_initialize()**
- **ProEngineerDisplaydatecodeGet()**
- **user_terminate()**

A Pro/TOOLKIT application must always contain the functions **user_initialize()** and **user_terminate()**. These functions have the prefix “user_” because they are written by the Pro/TOOLKIT application developer, but they are called from within Pro/ENGINEER at the start and end of the session.

Note: **user_initialize()** must contain at least one Pro/TOOLKIT API call. Failure to do so causes the Pro/TOOLKIT application to fail and return PRO_TK_GENERAL_ERROR.

The function **user_initialize()** initializes a synchronous-mode Pro/TOOLKIT application. This function must be present in any synchronous mode application in order for it to be loaded into Pro/ENGINEER. Use this function to setup user interface additions, or to run the commands required for a non-interactive application. **user_initialize()** is called after the Pro/ENGINEER application has been initialized and the graphics window has been created. It should contain any initializations that your Pro/TOOLKIT application needs, including any modification of Pro/ENGINEER menus (such as adding new buttons).

Note:

- Applications must contain at least one Pro/TOOLKIT function call in order to be registered with Pro/ENGINEER through this function.
- When coding a Pro/TOOLKIT application in C++ you must label **user_initialize()** as a C function by putting `extern "C"` before the function description.

The **user_initialize()** function is called with a number of optional arguments that can add to your function definition. All input and output arguments to this function are optional and do not need to be in the function signature. These arguments provide information about command-line arguments entered when Pro/ENGINEER was invoked, and the revision and build number of the Pro/ENGINEER in session. Refer to section the section **user_initialize() Arguments** for more information on the function arguments.

The initialization function must return 0 to indicate that the Pro/TOOLKIT application was initialized successfully. Any other return value will be interpreted as a failure, and the system will notify the Pro/ENGINEER user that the Pro/TOOLKIT application failed. Use the optional output argument to **user_initialize()** to specify the wording of this error message.

Note: The Pro/ENGINEER visible datecode format has changed. **user_initialize()** continues to receive the classic format based on the year and week of the Pro/ENGINEER build.

The function **ProEngineerDisplaydatecodeGet()** returns the user-visible datecode string from Pro/ENGINEER. Applications that present a datecode string to users in messages and information should use the new format for the Pro/ENGINEER displayed datecode.

The function **user_terminate()** is called at the end of the Pro/ENGINEER session, after the user selects **Yes** on the Exit confirmation dialog box. Its return type is void.

The following example is the empty core of a Pro/TOOLKIT application. This code should always be the starting point of each new application you develop.

```
#include "ProToolkit.h"
int user_initialize()
{
    return (0);
}
void user_terminate()
{
}
```

If you use the options to start and stop a multiprocess-mode Pro/TOOLKIT application within a Pro/ENGINEER session, **user_initialize()** and **user_terminate()** are called upon starting and stopping the Pro/TOOLKIT process only. However, any menu modifications defined in **user_initialize()** will be made, even if this involves repainting menus that are already displayed. All of these modifications will be reset when the Pro/TOOLKIT application is stopped.

user_initialize() Arguments

user_initialize() is called with a number of input and output arguments. As always in C, if you don't need to use an argument, your function does not need to declare it, provided that it declares all the arguments up to the last one used.

The input arguments are:

int arg_num	Number of command-line arguments.
char *argc[]	Command-line arguments passed by Pro/ENGINEER. (See further explanation below.)
char* version	Release name of the Pro/ENGINEER being used. Note: From Pro/ENGINEER Wildfire 4.0 onwards applications built with libraries older than Pro/ENGINEER 2001 will not run. You must recompile these applications with later versions of the Pro/TOOLKIT libraries.
char* build	The build number of the Pro/ENGINEER being used.

The output argument is:

wchar_t err_buff[80]	The text of an error message passed to Pro/ENGINEER if the Pro/TOOLKIT fails to initialize. Pro/ENGINEER displays this text when it reports the Pro/TOOLKIT failure (if user_initialize() returns non-zero).

The first command-line argument passed to Pro/TOOLKIT is the same one seen by Pro/ENGINEER; that is, it is the name of the Pro/ENGINEER executable. The remaining command-line arguments passed to **user_initialize()** are a subset of those given on the command line that invoked Pro/ENGINEER. The rule is that Pro/ENGINEER passes on to **user_initialize()** any command-line argument that starts with a “+”, or with a “-” followed by an upper-case character.

For example, these command-line arguments will be passed to Pro/TOOLKIT:

```
+batch=mybatchfile.txt
-Level=expert
```

Command-line arguments such as -g:no_graphics are interpreted by Pro/ENGINEER but not passed on to Pro/TOOLKIT.

Threading in Pro/TOOLKIT Applications

Calling Pro/TOOLKIT applications from within multiple threads of any application in any mode is not supported. Extra threads created by the application are to be used only for completion of tasks that do not directly call the Pro/TOOLKIT functions.

Function Introduced:

- **ProEngineerMultithreadModeEnable()**

Call the function **ProEngineerMultithreadModeEnable()** from within the initialization function **user_initialize()**, if your Pro/TOOLKIT application creates additional threads for processing. This function notifies Pro/ENGINEER to execute in the multithread enabled mode. Running in this mode eliminates the possibility of a memory corruption due to interaction between Pro/ENGINEER's thread libraries and the threads created by your application. This function does not work for multiprocess and asynchronous mode applications.

Note: Running Pro/ENGINEER in the multithread enabled mode may slow down performance. Therefore, **ProEngineerMultithreadModeEnable()** should be used only for applications that actually create multiple threads.

Using Pro/TOOLKIT to Make a Batch Pro/ENGINEER Session

Function Introduced:

- **ProEngineerEnd()**

If you want to use your Pro/TOOLKIT application to perform operations on Pro/ENGINEER objects that do not require interaction with the user, you can make all the necessary calls to Pro/TOOLKIT functions in **user_initialize()**. When your operations are complete, call the function **ProEngineerEnd()** to terminate the Pro/ENGINEER session.

A useful technique when designing a batch-mode Pro/TOOLKIT application is to use command-line arguments to Pro/ENGINEER as a way of signaling the batch mode and passing in the name of a batch control file. Consider the following command to start Pro/ENGINEER:

```
pro +batch=<filename>
```

In this example, the option will be ignored by Pro/ENGINEER, but will be passed as an input argument to **user_initialize()**. Inside that function, your code can recognize the switch, and get the name of the file that could contain, for example, the names of Pro/ENGINEER models to be processed, and operations to be performed on each one.

A batch-mode operation should also run without displaying any graphics. To ensure that the Pro/ENGINEER main Graphics Window and Message Window are not displayed, you should use either the command-line option `-g:no_graphics` (or the configuration file option “graphics NO_GRAPHICS”) to turn off the Pro/ENGINEER graphics. See the *Introduction to Pro/ENGINEER* manual for more details of these options.

Example 1: Batch Mode Operation

This example shows how to use the arguments to **user_initialize()** and the function **ProEngineerEnd()** to set up a batch mode session of Pro/ENGINEER. The application retrieves a part specified in the Pro/ENGINEER command line, performs an action on it (using the dummy function *UserAddHoles()*), saves the parts, and terminates Pro/ENGINEER.

```
/*=====*\nFUNCTION: UserAddHoles\nPURPOSE:  Find the circular datum curves and replace them with\n          holes.\n\\*=====*/\nUserAddHoles (ProMdl p_part)
```

```

{
/* .
.
.
*/
}
/*=====*\
    Load the part specified by the command line argument, and
    replace its datum curves with holes.
\=====*/
int user_initialize (int argc, char *argv[])
{
    ProMdl      p_part;
    ProName     name_wchar;
    ProError    err;
    char        *part_name;
/*-----*\
    Set up the part name from the argument list. Note that the
    Pro/ENGINEER arguments for Pro/TOOLKIT have a leading
    "+" or "-."
\-----*/
    part_name = argv[1];
    part_name++;
    ProStringToWstring (name_wchar, part_name);
/*-----*\
    Retrieve the part.
\-----*/
    err = ProMdlRetrieve (name_wchar, PRO_PART, &p_part);
    if (err != PRO_TK_NO_ERROR)
    {
        printf ("*** Failed to retrieve part %s\n", part_name);
        ProEngineerEnd();
    }
/*-----*\
    Add the holes to the part.
\-----*/
    UserAddHoles (p_part);
/*-----*\
    Save the part.
\-----*/
    ProMdlSave (p_part);
/*-----*\
    Terminate the Pro/ENGINEER session.
\-----*/
    ProEngineerEnd();
    return (0);
}
/*=====*\
FUNCTION: user_terminate()
PURPOSE:  Report successful termination of the program.
\=====*/

```

```
void user_terminate()
{
    printf ("Pro/TOOLKIT application terminated successfully\n");
}
```

User-Supplied Main

Function Introduced:

- **ProToolkitMain()**

In synchronous mode, the *main()* function of the Pro/TOOLKIT program is not written by you, the application developer. In DLL mode, the *main()* is the root of the Pro/ENGINEER program itself; in multiprocess synchronous mode, the *main()* is taken from the Pro/TOOLKIT library, and its job is to set up the communication channel with the separate Pro/ENGINEER executable.

If you are using a language such as C++ in your Pro/TOOLKIT application, it can be advantageous to compile the *main()* function with the C++ compiler to ensure that the program structure is correct for C++. In DLL mode, you cannot do this because you do not have access to the Pro/ENGINEER *main()*. But in multiprocess mode, you can substitute the Pro/TOOLKIT *main()* with your own, if you observe the following rules:

- Your *main()* must call the function **ProToolkitMain()** as its last statement. This function contains all the necessary setup code that needs to be run when the Pro/TOOLKIT application starts up in multiprocess mode.
- You *must* pass on the *argc* and *argv* arguments input to *main()* as the input arguments to **ProToolkitMain()** without modifying them in any way.
- You cannot make calls to any other Pro/TOOLKIT functions before the call to **ProToolkitMain()**, because the communications with Pro/ENGINEER have not yet been set up. You may, however, make other non-Pro/TOOLKIT function calls before calling **ProToolkitMain()**.

The following example shows a user-defined *main()* for use in multiprocess mode.

```
#include "ProToolkit.h"
main(
    int argc,
    char *argv[])
{
```

```

    .
    .
    .
    ProToolkitMain (argc, argv);
    /* The program exits from within ProToolkitMain().
       Any code here is not executed. */
}

```

Asynchronous Mode

For more information on the asynchronous mode, see the chapter ‘Core: Asynchronous Mode’.

Pro/TOOLKIT Techniques

This section describes the basic techniques you use when writing Pro/TOOLKIT applications. The topics are as follows:

- Object Handles
- Expandable Arrays

Also see the Visit Functions section for information on techniques used when writing Pro/TOOLKIT applications.

Object Handles

Each object in Pro/TOOLKIT has a corresponding C typedef, called a “handle”, whose name is always the name of the object itself with the prefix “Pro.” The handle is used as the type for all variables and arguments that refer to an object of that type. For example, any Pro/TOOLKIT function that performs an action on a solid has an input argument of type *ProSolid*.

Handles are classified into two types, depending on the way in which they are defined and have to be used. The two types are opaque handle (OHandle) and database handle (DHandle). The following sections describe these handles in detail.

OHandles

The simplest way to reference an object in Pro/ENGINEER is to use the memory address of the Pro/ENGINEER data structure that describes that object. To prevent the Pro/TOOLKIT application from accessing the content of the data structure for the object directly, the declaration of the structure is not provided. For example, the object handle *ProSurface* is defined as follows:

```
typedef struct geom* ProSurface;
```

The structure *struct geom* is used to describe a surface in Pro/ENGINEER, but the declaration of the structure is not included in Pro/TOOLKIT. This type of handle is called an opaque handle or opaque pointer for this reason.

Opaque handles have the advantage of simplicity and efficiency—they can be directly dereferenced inside the Pro/TOOLKIT function without any searching. They can also reference items that are transient and not in the Pro/ENGINEER database at all, such as the surfaces and edges that result from an interference volume calculation.

Other examples of Pro/TOOLKIT objects that are given OHandles are as follows:

```
typedef void* ProMdl;  
typedef struct curve_header* ProEdge;  
typedef struct sld_part* ProSolid;  
typedef struct entity* ProPoint;  
typedef struct entity* ProAxis;  
typedef struct entity* ProCsys;  
typedef struct entity* ProCurve;
```

Because opaque handles are just memory pointers, they suffer the disadvantage of all pointers in that they are volatile—they become invalid if the database object they refer to moves to a different memory location. For example, a *ProSurface* handle (a pointer to a Pro/ENGINEER surface) may become invalid after regeneration of the owning part (because its memory has been reallocated).

However, most of the Pro/ENGINEER structures referenced by opaque handles contain an integer identifier that is unique for items of that type within the owning model. This identifier retains its value through the whole life of that item, even between sessions of Pro/ENGINEER. Pro/TOOLKIT provides functions such as **ProSurfaceIdGet()** and **ProAxisIdGet()** that enable your application to use these identifiers as a persistent way to reference objects. These integer identifiers are also used in DHandles, described in the following section.

In the case of models, it is the name and type that are persistent. The functions **ProMdlNameGet()** and **ProMdlTypeGet()** provide the name and type of a model, given its opaque handle.

DHandles

A further limitation of opaque handles is that they can be too specific in cases where the action you want to perform is more generic. For example, a function that provides the name of a geometrical item should, ideally, be able to act on any of the geometry objects (*ProSurface*, *ProEdge*, *ProCsys*, and so on). However, the opaque handles for those different geometry items are not mutually compatible, so the Pro/TOOLKIT function would also need to know the type of the object before it could internally de-reference the opaque pointer.

To solve this problem, Pro/TOOLKIT defines a new, generic object type in these cases and declares it using a data handle, or DHandle. A DHandle is an explicit data structure that carries just enough information to identify a database item uniquely: the type, integer identifier, and handle to the owning model. Because the DHandle must contain the integer identifier (not the too-specific opaque handle), it also has the advantage of being persistent.

The most important examples of DHandles are *ProGeomitem*, which is the generic type for the geometry items previously mentioned, and *ProModelitem*, which is an even more generic object that includes *ProGeomitem*.

The declaration is as follows:

```
typedef struct pro_model_item
{
    ProType  type;
    int      id;
    ProMdl   owner;
} ProModelitem, ProGeomitem;
```

Note: Although the field *owner* is defined using the OHandle *ProMdl*, and is therefore strictly speaking volatile, this handle is guaranteed to remain valid while the Pro/ENGINEER model it refers to remains in memory.

The generic object *ProGeomitem* can represent any of the geometrical objects in a solid model, such as *ProSurface*, *ProEdge*, *ProCurve*, and *ProCsys*. The specific object types are said to be “derived from” the most generic type, and also to be “instances” of that type. The object type *ProGeomitem* is in turn an instance of *ProModelitem*, which can represent database items other than geometrical ones.

The generic object types such as *ProModelitem* and *ProGeomitem* are used as inputs to Pro/TOOLKIT functions whose actions are applicable to all of the more specific types of object that are instances of the generic type. For example, the function **ProGeomitemFeatureGet()** has that name because it can act on any type of object that is an instance of *ProGeomitem*: *ProSurface*, *ProEdge*, *ProCsys*, and so on. The function **ProModelitemNameGet()** is applicable to a wider range of database objects, not just geometrical ones.

If you have the OHandle to an object, such as *ProSurface*, and you want to call a generic function such as **ProGeomitemFeatureGet()**, you need to convert the OHandle to the more generic DHandle. Functions such as **ProGeomitemInit()** and **ProModelitemInit()** provide this capability. Similarly, you can convert a *ProGeomitem* to a *ProSurface* using the function **ProSurfaceInit()**. These techniques are illustrated in Example 3: Listing Holes in a Model, in the Visit Functions section.

Workspace Handles

When you use Pro/TOOLKIT to create an object in Pro/ENGINEER that contains a lot of information, such as a feature, it is important to be able to set up all of that information before adding the object to the Pro/ENGINEER database. The object-oriented style of Pro/TOOLKIT does not allow explicit access to the contents of such a structure, however. Instead, you must use a special workspace object that is allocated and filled by the Pro/TOOLKIT application using functions provided for that purpose.

The “workspace” is a memory area in Pro/ENGINEER that contains data structures not yet part of the design database.

Note: Do not confuse the term “workspace” in Pro/TOOLKIT with the concept of Workspace in Pro/INTRALINK.

The workspace object is identified by a handle that contains the address of the memory for the object, which is therefore similar to an OHandle. To distinguish this from handles that refer to objects in the Pro/ENGINEER database, such handles are called workspace handles (WHandles).

Expandable Arrays

Functions Introduced:

- **ProArrayAlloc()**
- **ProArrayFree()**
- **ProArraySizeGet()**
- **ProArraySizeSet()**
- **ProArrayMaxCountGet()**
- **ProArrayObjectAdd()**
- **ProArrayObjectRemove()**

The functions in this section enable you to access a set of programming utilities in general use within Pro/ENGINEER. The utilities fill a need that is common in C and Pascal programming—to provide a storage method that provides the advantages of an array, but without its limitations.

When you use an array for storage for a group of items, you have the advantage over a linked list in that the members are contiguous in memory. This enables you to access a given member using its index in the array. However, if you need to make frequent additions to the members in a way that cannot be predicted (a common situation in MCAE applications), you must reallocate the memory for the array each time.

A common compromise is to allocate the memory in blocks large enough to contain several array members, then reallocate the memory only when a block becomes full. You would choose the size of the blocks such that the frequency of reallocation is significantly reduced, while the amount of unused memory in the last block is acceptably small. The difficulty of this solution is that you would normally need a new set of utilities for each item you want to store as an array, and additional static data for each array to keep track of the number of blocks and the number of members.

The “expandable array” utilities provide a set of functions that can be applied to items of any size. The utilities do this by keeping a private header at the start of the array memory to which the “bookkeeping” information (the number and size of its members, and of the blocks) is written. The pointer your application sees is the address of the first block, not the address of the preceding header.

The importance of the expandable array utilities in a Pro/TOOLKIT application is not only that you can use them for your own arrays, but that you *must* use them for arrays of data passed between your application and the internals of Pro/ENGINEER through the Pro/TOOLKIT functions.

Note that because the array pointer is not the start of the contiguous memory claimed by the array utility, this pointer is not recognized by the operating system as a valid location for dynamic memory. Therefore, you will cause a fatal error if you try to use the memory management library functions, such as *realloc()* and *free()*.

The basic type used for referring to expandable arrays is *ProArray*, declared as a *void**.

The function **ProArrayAlloc()** sets up a new expandable array. Its inputs are as follows:

- The initial number of members in the array
- The size, in bytes, of each array member
- Number of objects added to ProArray at each memory reallocation. A higher number means more memory is preallocated and fewer reallocations of the ProArray are required.

The function outputs a pointer to the contiguous memory that will contain the array members. You can write to that memory to fill the array using the usual memory functions (such as *memcpy()* and *memset()*). If you increase the array size beyond the limit returned by **ProArrayMaxCountGet()**, this function returns an out-of-memory message.

The maximum memory allocated is 2 Mb, except for alpha_unix (or other 64-bit platforms), where the maximum is twice that.

The function **ProArrayFree()** releases the memory for the specified *ProArray*.

The function **ProArraySizeGet()** tells you how many members are currently in the specified array.

The **ProArraySizeSet()** function enables you to change the number of members in the expandable array. This function is equivalent to *realloc()*.

Function **ProArrayMaxCountGet()**, when given the specified structure size in bytes, returns the maximum number of structure elements a ProArray can support for that structure size.

The function **ProArrayObjectAdd()** adds a contiguous set of new members to an array, though not necessarily to the end of the array. The function also sets the contents of the new members. If you increase the array size beyond the limit returned by **ProArrayMaxCountGet()**, this function returns an out-of-memory message.

The function **ProArrayObjectRemove()** removes a member from the array. The member does not necessarily have to be the last member of the array.

Functions **ProArraySizeSet()**, **ProArrayObjectAdd()**, and **ProArrayObjectRemove()** change the size of the array, and might therefore also change its location.

The Pro/TOOLKIT functions use expandable arrays in the following circumstances:

- The function creates a filled, expandable array as its output.
- The function needs a filled, expandable array as its input.
- The function needs an existing expandable array to which to write its output.

An example of the first type of function is the geometry function **ProEdgeVertexdataGet()**, which provides a list of the edges and surfaces that meet at a specified solid vertex. When you have finished using the output, you should free the arrays of edges and surfaces (using the function **ProArrayFree()**).

An example of the second type of function is **ProSolidNoteCreate()**, which creates a design note in a solid. Because the text lines to add to the note are passed in the form of an expandable array, your application must create and fill the array using the functions **ProArrayAlloc()** and **ProArrayObjectAdd()** before you call **ProSolidNoteCreate()**.

An example of the third type of function is **ProElementChildrenGet()**, which gets the number of feature elements that are the children of the specified compound element. The feature elements form a tree that contains all the necessary information about a particular feature. (This function is therefore used in both feature analysis and feature creation.) Before calling **ProElementChildrenGet()**, you must call **ProArrayAlloc()** to create an empty array. You can then use **ProArraySizeGet()** to find out how many elements were added to the array.

There is a fourth case, which is a variation of the first, in which a Pro/TOOLKIT function creates an expandable array as its output the first time it is called in an application, but overwrites the same array on subsequent calls. An example of this is **ProSelect()**, whose output array of *ProSelection* structures must *not* be freed using **ProArrayFree()**. You must also make sure to copy the contents of the array if you need to use it to make another call to **ProSelect()**.

The conventions are chosen for each function according to its individual needs. For example, **ProElementChildrenGet()** is typically called in a recursive loop to traverse a tree, so the fourth method of allocation would be inconvenient.

The rules for each Pro/TOOLKIT function are documented in the browser.

Example 2: Expandable Arrays

This example shows how to use expandable arrays, not as input or output for a Pro/TOOLKIT function, but to create a utility that provides an alternative to a Pro/TOOLKIT visit function. To use Pro/TOOLKIT to access all the features in a solid, you call the function **ProSolidFeatVisit()**. However, you might prefer to use a function that provides an array of handles to all of the features, then traverse this array. This kind of function is called a “collection” function, to distinguish it from a visit function. Although Pro/TOOLKIT does not provide collection functions, you can use the technique demonstrated in the example to write your own.

The utility function *UserFeatureCollect()* passes an empty, expandable array of feature handles as the application data to **ProSolidFeatVisit()**. The visit function *FeatVisitAction()* adds the handle to the visited feature to the array using **ProArrayObjectAdd()**.

```
/*----- Pro/Toolkit Includes -----*/
#include "ProToolkit.h"
#include "ProMdl.h"
#include "ProFeature.h"
#include "ProMenu.h"
#include "ProSolid.h"
#include "ProArray.h"
#include "ProFeatType.h"
#include "ProUtil.h"

/*----- Application Includes -----*/
#include "TestError.h"
```

```

/*----- Function Prototypes -----*/
int UserFeatCollectExample();
int UserFeatureCollect(ProFeature **fArray);
ProError FeatFilterAction(ProFeature* pFeature, ProAppData aData);
ProError FeatVisitAction(ProFeature* pFeature, ProError eStatus,
ProAppData aData);

/*=====*\
Function: UserButtonAction()
Purpose: Write feature type and id to file on disk and diplays in Info
Window
\*=====*/

int UserFeatCollectExample()
{
    ProFeature *fList;
    ProFeattype fType;
    int status, eStatus, aSize, i;
    FILE *fp;
    ProPathw_filename;

    status = ProArrayAlloc(0, sizeof(ProFeature), 1, (ProArray *)&fList);
    ERROR_CHECK("UserButtonAction", "ProArrayAlloc", status);
    eStatus = UserFeatureCollect(&fList);

    fp = PTApplsUnicodeFopen("fCollect.txt", "w");

    status = ProArraySizeGet(fList, &aSize);
    ERROR_CHECK("UserButtonAction", "ProArraySizeGet", status);

    for ( i = 0; i < aSize; i++)
    {
        status = ProFeatureTypeGet(&fList[i], &fType);
        ProTKFprintf(fp, "\nFeature type: %d\tFeature id: %d\n", fType,
                    fList[i].id);
    }
    eStatus = fclose(fp);

    ProStringToWstring(w_filename, "fCollect.txt");
    status = ProInfoWindowDisplay(w_filename, NULL, NULL);

    status = ProArrayFree((ProArray*)&fList);
    ERROR_CHECK("UserButtonAction", "ProArrayFree", status);

    return(eStatus);
}

/*=====*\
Function: UserFeatureCollect()
Purpose: Visit all features in part

```

```

\*=====*/

int UserFeatureCollect(ProFeature **fArray)
{
    ProMdlmodel;
    int status;

    status = ProMdlCurrentGet(&model);
    ERROR_CHECK("UserFeatureCollect", "ProMdlCurrentGet", status);

    status = ProSolidFeatVisit((ProSolid)model, FeatVisitAction,
                               FeatFilterAction, (ProAppData)fArray);
    ERROR_CHECK("UserFeatureCollect", "ProSolidFeatureVisit", status);

    return(status);
}

/*=====*\
Function: FeatFilterAction()
Purpose: Eliminates parts that are not visible from visit action
\*=====*/
ProError FeatFilterAction(ProFeature* pFeature, ProAppData aData)
{
    int status;
    ProBoolean iVisible;

    status = ProFeatureVisibilityGet(pFeature, &iVisible);
    ERROR_CHECK("FeatFilterAction", "ProFeatureVisibilityGet", status);

    if (iVisible == PRO_B_TRUE)
        return(PRO_TK_NO_ERROR);
    else
        return(PRO_TK_CONTINUE);
}

/*=====*\
Function: FeatVisitAction()
Purpose: Collects feature handles in expandable array
\*=====*/
ProError FeatVisitAction(ProFeature *pFeature, ProError eStatus,
                        ProAppData aData)
{
    int status;

    status = ProArrayObjectAdd((ProArray *)aData, PRO_VALUE_UNUSED, 1,
                               pFeature);
    ERROR_CHECK("FeatVisitAction", "ProArrayObjectAdd", status);

    return(status);
}

```


}

Visit Functions

In a Pro/TOOLKIT application, you often want to perform an operation on all the objects that belong to another object, such as all the features in a part, or all the surfaces in a feature. For each case, Pro/TOOLKIT provides an appropriate “visit function.” A visit function is an alternative to passing back an array of data.

You write a function that you want to be called for each item (referred to as the “visit action” function) and pass its pointer to the Pro/TOOLKIT visit function. The visit function then calls your visit action function once for each visited item.

Most visit functions also provide for a second callback function, the filter function, which is called for each visited item before the action function. The return value of the filter function controls whether the action function is called. You can use the filter function as a way of visiting only a particular subset of the items in the list.

For example, the visit function for visiting the features in a solid is declared as follows:

```
ProError ProSolidFeatVisit (
    ProSolid          solid,
    ProFeatureVisitAction visit_action,
    ProFeatureFilterAction filter_action,
    ProAppData         app_data);
```

The first argument is the handle to the solid (the part or assembly) whose features you want to visit.

The second and third arguments are the visit action function and filter function, respectively.

The type of the final argument, *ProAppData*, is a typedef to a *void**. This argument is used to pass any type of user-defined application data down to the *visit_action* and *filter_action* functions through the intervening Pro/TOOLKIT layer. You might want to use this as an alternative to allowing global access to the necessary data.

Although you write the visit action and filter functions, they are called from within the Pro/TOOLKIT visit function, so their arguments are defined by Pro/TOOLKIT. To enable the C compiler to check the arguments, Pro/TOOLKIT provides a typedef for each of these functions.

For example, the type for the action function for **ProSolidFeatVisit()** is as follows:

```
typedef ProError (*ProFeatureVisitAction)(  
    ProFeature *feature,  
    ProError status,  
    ProAppData app_data);
```

It takes three arguments:

- The handle to the feature being visited
- The status returned by the preceding call to the filter function
- The application data passed as input to the visit function itself

The type for the filter function is as follows:

```
typedef ProError (*ProFeatureFilterAction)(  
    ProFeature *feature,  
    ProAppData app_data);
```

Its two arguments are the handle to the feature being visited and the application data.

The filter action function should return one of the following values:

- **PRO_TK_CONTINUE**—Do not call the visit action for this object, but continue to visit the subsequent objects.
- Any other value—Call the visit action function for this object and pass the return value as the status input argument.

The visit action function should return one of the following values:

- **PRO_TK_NO_ERROR**—Continue visiting the other objects in the list.
- **PRO_TK_E_NOT_FOUND**—For visit functions, this value indicates that no items of the desired type were found and no functions could be visited.
- Any other value (including **PRO_TK_CONTINUE**)—Terminate the visits. Typically this status is returned from the visit function upon termination, so that the calling function knows the reason that visiting terminated abnormally.

Example 3: Listing Holes in a Model

This example demonstrates several of the principles used in Pro/TOOLKIT, including visit functions, the use of Ohandles and Dhandles, and the *ProSelection* object.

The example shows the function *UserDemoHoleList()*, which visits the axes in the current part that belong to features of type HOLE. It then writes the axis names and feature identifiers to a file, and highlights the hole features.

The top function, *UserDemoHoleList()*, calls **ProSolidAxisVisit()**. The function uses the *ProAppData* argument to pass to the visit action function, *UserDemoAxisAct()*, a structure that contains the file pointer and handle to the owning solid.

```

/*----- Pro/Toolkit Includes -----*/
#include "ProToolkit.h"
#include "ProAxis.h"
#include "ProFeature.h"
#include "ProFeatType.h"
#include "ProGeomitem.h"
#include "ProMdl.h"
#include "ProModelitem.h"
#include "ProObjects.h"
#include "ProSelection.h"
#include "ProSizeConst.h"
#include "ProSolid.h"
#include "ProUtil.h"

/*----- Application Includes -----*/
#include "TestError.h"

/*----- Global Data -----*/
typedef struct surface_visit_data
{
    FILE *fp;
    ProSolid part;
} AxisVisitData_t;

/*----- Function Prototypes -----*/
ProError UserDemoAxisAct();
ProError UserDemoHoleList();
int UserVisitDemo();

/*=====*\
    FUNCTION : UserDemoAxisAct()
    PURPOSE  : Axis-visit action function, to write the axis name to a
               file.
\*=====*/
ProError UserDemoAxisAct(
    ProAxis axis,
    ProError filt_status,
    ProAppData app_data)
{
    ProError status;
    AxisVisitData_t *p_data = (AxisVisitData_t*)app_data;

```

```

    ProSolid part=p_data->part;
    FILE *fp=p_data->fp;
    int id;
    ProModelitem modelitem;
    ProFeature feature;
    ProFeattype ftype;
    ProName wname;
    char name[PRO_NAME_SIZE];
    ProSelection selection;
/*-----*\
    Get the axis id
/*-----*/
    status = ProAxisIdGet(axis, &id);
    if(status != PRO_TK_NO_ERROR)
        return(status);
/*-----*\
    Make a Modelitem handle for the axis
/*-----*/
    status = ProModelitemInit(part, id, PRO_AXIS, &modelitem);
    if(status != PRO_TK_NO_ERROR)
        return(status);
/*-----*\
    Get the feature the axis belongs to
/*-----*/
    status = ProGeomitemFeatureGet(&modelitem, &feature);
    if(status != PRO_TK_NO_ERROR)
        return(status);
/*-----*\
    Get the type of the feature
/*-----*/
    status = ProFeatureTypeGet(&feature, &ftype);
    if(status != PRO_TK_NO_ERROR)
        return(status);
/*-----*\
    If the type was not HOLE, skip
/*-----*/
    if(ftype != PRO_FEAT_HOLE)
        return(PRO_TK_NO_ERROR);
/*-----*\
    Get the name of the axis
/*-----*/
    status = ProModelitemNameGet(&modelitem, wname);
    if(status != PRO_TK_NO_ERROR)
        return(status);
    ProWstringToString(name,wname);
/*-----*\
    Print out the axis name and hole id
/*-----*/
    ProTKFprintf(fp, "Axis %s belongs to hole feature %d\n", name,
        feature.id);
/*-----*\

```

```

        Highlight the owning hole
    \*-----*/
        ProSelectionAlloc(NULL, &feature, &selection);
        ProSelectionHighlight(selection, PRO_COLOR_HIGHLIGHT);
        ProSelectionFree(&selection);

        return(PRO_TK_NO_ERROR);
    }
    \*=====*\
        FUNCTION : UserDemoHoleList()
        PURPOSE  : Function to list the axes which belong to hole features
                   in a part, and report their names
    \*=====*/
ProError UserDemoHoleList(
    ProSolid part)
{
    ProError status;
    AxisVisitData_t data;

    data.part = part;
    \*-----*\
        Open the text file
    \*-----*\
        data.fp = PTApplsUnicodeFopen("visit_test.dat", "w");
    \*-----*\
        Visit all the axes using the visit and filter functions above.
        Pass the owning solid, and the text file pointer using the app_data
        argument.
    \*-----*\
        status = ProSolidAxisVisit(part, UserDemoAxisAct,
                                   NULL, (ProAppData)&data);
        ERROR_CHECK("UserDemoHoleList", "ProSolidAxisVisit", status);
    \*-----*\
        Close the file
    \*-----*\
        fclose(data.fp);
        return( PRO_TK_NO_ERROR );
    }
    \*=====*\
        FUNCTION : UserVisitDemo
        PURPOSE  : call UserDemoHoleList on current model
    \*=====*/
int UserVisitDemo( )
{
    ProMdl model;
    ProError status;

    status = ProMdlCurrentGet( &model );
    ERROR_CHECK( "UserDemoVisit", "ProMdlCurrentGet", status );

    if ( status == PRO_TK_NO_ERROR )

```

```

    status = UserDemoHoleList( (ProSolid)model );

    return( (int)status );
}

```

Wide Strings

Pro/TOOLKIT, like Pro/ENGINEER, has to work in environments where character strings use codes other than ASCII, and might use a bigger character set than can be coded into the usual 1-byte char type. The most important example of this is the Japanese KANJI character set.

For this reason, Pro/TOOLKIT uses the type *wchar_t* instead of *char* for all characters and strings that may be visible to the Pro/ENGINEER user. This includes all text messages, keyboard input, file names, and names of all dimensions, parameters, and so on, used within a Pro/ENGINEER object.

Defining wchar_t

Although most platforms supported by Pro/TOOLKIT provide a definition of *wchar_t* in a system include file, not all do. Those that do use definitions of different lengths; some provide definitions that are not suitable for all the character codes supported by Pro/ENGINEER. Therefore, Pro/ENGINEER takes considerable care to make sure it uses a suitable definition of *wchar_t* on each supported platform.

It is essential to make sure your Pro/TOOLKIT application is using the same definition of *wchar_t* as Pro/ENGINEER on each platform your application supports. To make this easier, Pro/TOOLKIT supplies the include file *pro_wchar_t.h*. This file ensures that, if a definition of *wchar_t* has not already been made in an earlier include file, one is provided that is consistent with the Pro/ENGINEER definition of the type. Because this file is included by the file *ProToolkit.h*, you should include *ProToolkit.h* as the very first include file in each source file.

Setting the Hardware Type

To make the handling of the wide character type *wchar_t* across different platforms simpler and more reliable, the include file *pro_wchar_t.h* is hardware dependent. It knows which platform is being used from the setting of the environment variable `PRO_MACHINE`; the recognized values are listed in the include file *pro_hardware.h*, included by *pro_wchar_t.h*.

You must make sure that the environment variable `PRO_MACHINE` is set to indicate the type of hardware you are using. Set it to same value used for the makefile macro `PRO_MACHINE` in the makefile taken from the Pro/TOOLKIT loadpoint.

Checking Your Declaration of *wchar_t*

Function Introduced:

- **ProWcharSizeVerify()**

The function **ProWcharSizeVerify()** checks to make sure you have the correct declaration of *wchar_t*. PTC recommends that you always call this function at the beginning of the **user_initialize()** function (or *main()* in asynchronous mode).

You pass as input the size of your *wchar_t* definition, in bytes, and the function outputs the correct size. It returns `PRO_TK_NO_ERROR` if your size is correct, and `PRO_TK_GENERAL_ERROR` otherwise. You can check for correctness as follows:

```
int proe_wchar_size;
int protk_wchar_size = sizeof (wchar_t);

if (ProWcharSizeVerify (protk_wchar_size, &proe_wchar_size) !=
    PRO_TK_NO_ERROR)
{
    ProMessageDisplay (msgfil, "USER wchar_t size is %0d,
        should be %1d", &protk_wchar_size, &proe_wchar_size);
    return (1);
}
```

String and Widestring Functions

Pro/ENGINEER provides many functions taking as inputs a fixed-length character or wide character array as a string. Due to some platform-specific behavior, PTC recommends that you do not use string literals in lieu of fixed length arrays. Always copy the literal strings to the full size array that the functions accepts as the input.

For example the following function will get warnings on certain platforms because the code expects that the arguments can be modified.

```
ProEngineerStart("proe_path","text_path");  
where ProCharPath proe_path="proe_path";  
ProCharPath text_path="text_path";
```

To overcome this error, it is recommended that you replace the literal strings in the function with defined arrays as follows:

```
ProEngineerStart(proe_path,text_path);
```

Functions Introduced:

- **ProStringToWstring()**
- **ProWstringToString()**
- **ProWstringLengthGet()**
- **ProWstringCopy()**
- **ProWstringCompare()**
- **ProWstringConcatenate()**

Wide character strings are not as easy to manipulate in C as ordinary character strings. In general, there are no functions for wide strings that correspond to the standard C *str*()* functions. *printf()* does not have a format for wide strings, and you cannot set a wide string to a literal value in a simple assignment. Because of this, it is frequently convenient to convert wide strings to character strings, and vice versa. This is the purpose of the functions **ProStringToWstring()** and **ProWstringToString()**.

The function **ProWstringLengthGet()** is used to find the length of a widestring.

The function **ProWstringCopy()** copies a wistring into another buffer. You should allocate enough memory in the target setting to perform the copy operation. The number of characters to be copied is provided as input through num_chars. Use PRO_VALUE_UNUSED to copy the entire string.

The function **ProWstringCompare()** compares two wistrings for equality. The two wistrings to be compared are given as inputs. The argument num_chars allows comparison of portions of the string, pass PRO_VALUE_UNUSED to compare the entire strings.

The function **ProWstringConcatenate()** concatenates two wistrings. You should allocate enough memory in the target string for the copy operation. The number of characters to concatenate is given as input through num_chars. Use PRO_VALUE_UNUSED to add the entire source string to the target string.

The source code for other useful utilities is located in the file <TK_LOADPOINT>protk_appls/pt_examples/pt_utils/Utils string.c

Example 4: String Conversion

The function *UsrModelFilenameGet()* shows how to convert wide strings to character strings.

```
/*----- Pro/Toolkit Includes -----*/
#include <ProToolkit.h>
#include <ProMdl.h>
#include <ProObjects.h>
#include <ProSizeConst.h>
#include <ProUtil.h>
#include <PTApplsUnicodeUtils.h>

/*----- Application Includes -----*/
#include <TestError.h>

/*----- Function Prototypes -----*/
ProError UsrModelFilenameGet();

ProError UsrModelFilenameGet(
    ProMdl      model,
    ProCharName filename)
{
    ProError      status;
    ProMdlData    data;
    ProCharName   obj_name, obj_type;

    status = ProMdlDataGet (model, &data);
```

```
    if (status != PRO_TK_NO_ERROR)
        return (status);

    ProWstringToString(obj_name, data.name);
    ProUtilStringLower(obj_name, obj_name);

    ProWstringToString(obj_type, data.type);
    ProUtilStringLower(obj_type, obj_type);

    ProTKSprintf(filename, "%s.%s;%d", obj_name,
        obj_type, data.version);

    return (PRO_TK_NO_ERROR);
}
```

2

Core: Models and Model Items

This chapter describes Pro/TOOLKIT modes, models, and model items.

Topic	Page
Modes	2 - 2
Models	2 - 2
Model Items	2 - 8
Version Stamps	2 - 12
Layers	2 - 13
Layouts	2 - 20
Visiting Displayed Entities	2 - 20

Modes

Functions Introduced:

- **ProModeCurrentGet()**
- **ProModeCurrentGet()**
- **ProSectionIsActive()**

The term “mode” in Pro/ENGINEER means the type of model currently being edited by the user. The possible modes are the options under the Pro/ENGINEER command **File, New**.

The *ProMode* object in Pro/TOOLKIT is therefore an enumerated type, declared in the file *ProMode.h*, as are the prototypes for the mode functions.

Find the name of the mode using the function **ProModeCurrentGet()**. The function **ProModeCurrentGet()** outputs the mode in which Pro/ENGINEER is being used, in the form of the *ProMode* enumerated type. If there is no current model—for example, because no model has been retrieved, or because the user has selected **Window>Close**—the function returns an error status (PRO_TK_BAD_CONTEXT).

The function **ProSectionIsActive()** checks if the sketcher is currently active even if the current mode is part or assembly.

Models

This section describes Pro/TOOLKIT models. The topics are as follows:

- The ProMdl Object
- Identifying Models
- Models in Session
- File Management Operations

The ProMdl Object

A model is a top-level object in a Pro/ENGINEER mode. For example, in Part mode, the model is a part; in Assembly mode, the model is an assembly.

The object *ProMdl* is therefore used for all those functions whose action applies to models of any type, such as file management operations and version stamps.

The declaration of *ProMdl* is as follows:

```
typedef void* ProMdl;
```

Instances of the *ProMdl* object are objects for the more specific Pro/ENGINEER modes. For example, *ProSolid* is an instance of *ProMdl*, and *ProAssembly* and *ProPart* are instances of *ProSolid*. All these object types are represented in Pro/TOOLKIT by opaque handles, and you can make conversions between the types by casting.

Creating Models

Functions Introduced

- **ProSolidCreate()**
- **ProMfgCreate()**
- **ProSection2DAlloc()**
- **ProDrawingFromTpltCreate()**
- **ProMdlStartAction()**

Pro/TOOLKIT supports creation of models for Solids, Manufacturing, Section (two-dimensional only), and Drawing.

See Creating a Solid for a complete description of **ProSolidCreate()**.

For more information on **ProMfgCreate()** see Creating a Manufacturing Model.

Allocating a Two-Dimensional Section gives more details on **ProSection2DAlloc()**.

Creating Drawings from Templates has more information on the function **ProDrawingFromTpltCreate()**.

The notification function **ProMdlStartAction()** is a type for a callback function for PRO_MDL_START. This function changes the way users can create models by replacing the Pro/ENGINEER model template dialog box with a user-specified action.

The user-specified action contains user-programmed activities that allow customization of new models by applying templates with more inputs than model creation “on-the-fly” or the standard Pro/ENGINEER template.

The callback function is activated after the user selects OK from the File—New dialog box, but only if the Use Default Template checkbox is not selected. The user's application must create a new model of the same type and subtype specified by the callback function.

Setting the configuration option "force_new_file_options_dialog" to "yes" forces the Use Default Template button to be hidden, and calls the callback for all models created through the File->New dialog.

This function supports all model types.

See Event-driven Programming: Notifications for more data on using callback functions.

Identifying Models

Functions Introduced:

- **ProMdlNameGet()**
- **ProMdlTypeGet()**
- **ProMdlCommonnameGet()**
- **ProMdlCommonnameSet()**
- **ProMdlInit()**
- **ProMdlIdGet()**
- **ProMdlActiveGet()**
- **ProMdlDataGet()**
- **ProMdlSubtypeGet()**
- **ProMdlToModelitem()**

The object *ProMdl* is an opaque handle, and is therefore volatile. It cannot be used to refer to models that are not in memory in Pro/ENGINEER, for example. To reference a model in a way that is valid for models not in memory, and also persistent across sessions of Pro/ENGINEER, use the model name and type.

The functions **ProMdlNameGet()** and **ProMdlTypeGet()** provide the name and type of a model, given its *ProMdl* handle. The type of a model is expressed in terms of the enumerated type *ProType*.

The functions **ProMdlCommonnameGet()** and **ProMdlCommonnameSet()** obtain and assign the common name of a model, respectively. This name is used to identify the model in a Product Database Management system such as Windchill PDMLink.

Note: **ProMdlCommonnameSet()** can modify the name only for models that are not yet owned by Windchill PDMLink, or in certain situations if the configuration option `let_proe_rename_pdm_objects` is set to yes.

The function **ProMdlInit()** does the opposite, and provides a valid *ProMdl* handle for a given name and type. The function fails if the specified model is not in memory in the Pro/ENGINEER session.

A third way to identify a model is by an integer identifier. Unlike the integer identifiers of objects within a model, such as surfaces and edges, the identifier of a model is not persistent between Pro/ENGINEER sessions. The function **ProMdlIdGet()** provides the identifier of a model, given its *ProMdl* handle.

The function **ProMdlActiveGet()** retrieves the model handle *ProMdl* for an active Pro/ENGINEER object.

The function **ProMdlDataGet()** provides a C structure that contains information about the name and location of the operating system file in which the model is saved.

The function **ProMdlSubtypeGet()** provides the subtype (such as sheet metal) of a specified model. Valid model subtypes are Part, Assembly, or Manufacturing. This is like finding subtypes at the Pro/ENGINEER **File>New>Model Type** menu.

The function **ProMdlToModelitem()** is used only when you need to represent the model as a *ProModelitem* object—the first step in building a *ProSelection* object that describes the role of a model in a parent assembly. Model item objects are described later in this chapter. See the chapter Fundamentals for information on the *ProSelection* object.

Example 1: Finding the Handle to a Model

The following example shows how to find a model handle, given its name and type.

```
ProName  name;
ProType  type;
ProMdl   part;
ProError status;

ProStringToWstring (name, "PRT0001");
type = PRO_PART;
status = ProMdlInit (name, type, &part);
```

Models in Session

Functions Introduced:

- **ProSessionMdlList()**
- **ProMdlCurrentGet()**
- **ProMdlDependenciesList()**
- **ProMdlDeclaredList()**
- **ProMdlModificationVerify()**
- **ProMdlIsModifiable()**

The function **ProSessionMdlList()** provides an array of *ProMdl* handles to models of a specified type currently in memory.

The function **ProMdlCurrentGet()** provides the *ProMdl* handle to the model currently being edited by the user.

The function **ProMdlDependenciesList()** provides an array of *ProMdl* handles to the models in memory upon which a specified model depends. One model depends on another if its contents reference that model in some way. For example, an assembly depends on the models that form its components, and a drawing model depends on the solid models contained in it. Sometimes, two models can be mutually dependent, such as when a model feature references a geometry item in a parent assembly.

The function **ProMdlDeclaredList()** provides an array of *ProMdl* handles to layout models that have been declared to a specified solid model.

The function **ProMdlModificationVerify()** tells you whether a specified model in memory has been modified since it was last saved or retrieved. See the section Version Stamps for a more flexible way of keeping track of changes to a model.

The function **ProMdlIsModifiable()** checks if the specified model is modifiable.

File Management Operations

Functions Introduced:

- **ProMdlCopy()**
- **ProMdlfileCopy()**
- **ProMdlRetrieve()**
- **ProMdlMultipleRetrieve()**
- **ProSolidRetrievalErrorsGet()**
- **ProMdlSave()**
- **ProMdlIsSaveAllowed()**
- **ProMdlErase()**
- **ProMdlEraseNotDisplayed()**
- **ProMdlEraseAll()**
- **ProMdlRename()**
- **ProMdlBackup()**
- **ProMdlDelete()**

These functions perform the same actions as the corresponding Pro/ENGINEER file management commands, with the following exceptions:

- **ProMdlCopy()** and **ProMdlfileCopy()** are equivalent to the **Save As** command in the **File** pull-down menu of the Pro/ENGINEER menu bar. **ProMdlCopy()** takes the model handle as input, whereas **ProMdlfileCopy()** takes the type and name of the model to copy.
- **ProMdlRetrieve()** retrieves the model into memory, but does not display it or make it the current model.
- **ProMdlMultipleRetrieve()** retrieves multiple models into memory. Use the *ui_flag* parameter to set model display to on or off.
- **ProSolidRetrievalErrorsGet()** returns the data structure containing errors that occur during model retrieval. While retrieving a complex assembly, Pro/ENGINEER sometimes encounters errors in retrieving particular components and assembling them appropriately in the assembly. In the user interface, you are informed of errors as they occur, through a dialog box. In Pro/TOOLKIT, the retrieval functions automatically suppress or freeze problem components and

return PRO_TK_NO_ERROR. To know whether errors have occurred during retrieval, use the function **ProSolidRetrievalErrorsGet()**. The errors are returned as the elements of the **ProSolidretrievalerrs** array. The retrieval error information must be obtained immediately after a call to the ProMdlRetrieve or equivalent retrieval function.

- **ProMdlIsSaveAllowed()** checks whether a given model can be saved.
- **ProMdlEraseNotDisplayed()** erases all the models that are not referenced in a window from the current session.
- **ProMdlEraseAll()** erases a model and all the models that it uses, except those that have cyclic dependencies (that is, models used by other models in the session). For example, **ProMdlEraseAll()** recursively erases all subassemblies of an assembly and all solids referenced from a drawing. This function also works in cases where some models to be erased have mutual dependencies, but only if the erased models are not used by other models. However, while erasing an active model, **ProMdlEraseAll()** only clears the graphic display immediately, it does not clear the data in the memory until the control returns to Pro/ENGINEER from the Pro/TOOLKIT application.

Model Items

A “model item” is a generic object used to represent any item contained in any type of model, for the purpose of functions whose actions are applicable to all these types of item. (Some items, such as “version stamp,” retain their own object types.)

The object type *ProModelitem* is a DHandle (data handle), a structure that contains the item type, the persistent integer identifier of the item, and the handle to the owning object.

The object *ProGeomitem*, a generic geometrical object described later in this guide, is an instance of *ProModelitem*, and is a DHandle that shares the same type declaration. Therefore, the functions in this section are also directly applicable to *ProGeomitem* objects.

The typedef for the *ProModelitem* data handle is as follows:

```
typedef struct pro_model_item
{
    ProType    type;
    int        id;
    ProMdl     owner;
} ProModelitem
```

Functions Introduced:

- **ProModelitemByNameInit()**
- **ProModelitemInit()**
- **ProModelitemMdlGet()**
- **ProModelitemDefaultnameGet()**
- **ProModelitemNameGet()**
- **ProModelitemNameSet()**
- **ProModelitemNameCanChange()**
- **ProModelitemUsernameDelete()**
- **ProModelitemHide()**
- **ProModelitemUnhide()**
- **ProModelitemIsHidden()**

The function **ProModelitemByNameInit()** returns a pointer to an item (structure), given the name and the type of the item. The valid item types are:

- Edge
- Surface
- Feature
- Co-ordinate System
- Axis
- Point
- Quilt
- Curve
- Layer
- Note

The function **ProModelitemInit()** is used to generate a *ProModelitem* object from the information contained in the structure. You can create such a structure directly, but using this function you can also confirm the existence of the item in the model database.

The function **ProModelitemMdlGet()** extracts the *ProMdl* handle from the structure.

The function **ProModelitemDefaultnameGet()** gets the default name for a new model item of a particular type if it was created taking the model handle as input.

The two functions **ProModelitemNameGet()** and **ProModelitemNameSet()** read and set the name of the Pro/ENGINEER database object referred to by the model item. These functions are therefore applicable to all the instances of *ProModelitem*, such as *ProGeomitem* and all its instances, including *ProSurface*, *ProEdge*, *ProCsys*, and *ProAxis*.

Note: In addition to notes of the type PRO_NOTE, the functions **ProModelitemNameGet()** and **ProModelitemNameSet()** can be used to read and set the name of the following annotation types:

- Driving or driven dimension of the type PRO_DIMENSION
- Reference dimension of the type PRO_REF_DIMENSION
- Symbol instance of the type PRO_SYMBOL_INSTANCE
- Surface finish of the type PRO_SURF_FIN
- Geometric tolerance of the type PRO_GTOL
- Set datum tag of the type PRO_SET_DATUM_TAG
(applicable only for **ProModelitemNameGet()**)

The function **ProModelitemNameCanChange()** identifies whether the name of the model item can be modified by the user or by Pro/TOOLKIT.

The function **ProModelitemUsernameDelete()** deletes the user-defined name of the model item from the Pro/ENGINEER database.

The functions **ProModelitemHide()** and **ProModelitemUnhide()** are equivalent to the **Hide** and **Unhide** commands in the Pro/ENGINEER menu, respectively. **ProModelitemHide()** hides the specified model item, whereas **ProModelitemUnhide()** unhides the model item.

The function **ProModelitemIsHidden()** identifies if the specified model item is hidden.

Example 2: Renaming a Selected Surface

This example shows how to use the functions **ProModelitemNameGet()** and **ProModelitemNameSet()**. See the Core: 3D Geometry chapter for an explanation of *ProSurface* and its functions.

```

/*----- Pro/Toolkit Includes -----*/
#include <ProToolkit.h>
#include <ProSelection.h>
/*----- Application Includes -----*/
#include <TestError.h>
/*----- Function Prototypes -----*/
ProError UserSurfRename();
/*=====*\
Function: UserSurfRename()
Purpose: Rename Selected Surface
\*-----*/
ProError UserSurfRename()
{
    int sel_count;
    ProError status;
    ProCharName name;
    ProModelitem p_mdl_item;
    ProFileName msgfile;
    ProName w_name;
    ProSelection *psels=NULL;
/*-----*\
    Prompt user for selection of surface
\*-----*/
    ProStringToWstring(msgfile,"msg_uggeom.txt");
    status = ProMessageDisplay(msgfile,"USER Select Surface to Rename:");
    ERROR_CHECK("UserSurfRename", "ProMessageDisplay",status);
    if((ProSelect("surface",1,NULL,NULL,NULL,NULL,&psels, &sel_count) !=
        PRO_TK_NO_ERROR) || (sel_count < 1))
        return((int) PRO_TK_GENERAL_ERROR);

    status = ProSelectionModelitemGet(psels[0],&p_mdl_item);
    ERROR_CHECK( "UserSurfRename", "ProSelectionModelitemGet", status );
    status = ProModelitemNameGet(&p_mdl_item,w_name);
    ERROR_CHECK( "UserSurfRename", "ProModelitemNameGet",status);
/*-----*\
    Display current name or "NONE" if surface is not named
\*-----*/
    if(status != PRO_TK_NO_ERROR)
        ProTKSprintf(name,"NONE");
    else

```

```

{
    ProWstringToString(name,w_name);
}
status = ProMessageDisplay(msgfile,"USER Enter Name [%0s]:",name);
ERROR_CHECK( "UserSurfRename", "ProMessageDisplay(Enter Name)",status);
status = ProMessageStringRead(PRO_NAME_SIZE,w_name);
ERROR_CHECK( "UserSurfRename", "ProMessageStringRead",status);
if(status == PRO_TK_NO_ERROR)
{
    status = ProModelitemNameSet(&p_mdl_item,w_name);
    ERROR_CHECK( "UserSurfRename", "ProModelitemNameSet",status);
}
ProMessageClear();
return(status);
}

```

Version Stamps

The version stamp object provides a way of keeping track of changes in a Pro/ENGINEER model to which your Pro/TOOLKIT application may need to respond. Pro/ENGINEER models and features contain an internal version stamp incremented each time some design change is made to that model or feature. The functions in this section enable you to read version stamps in order to look for design changes.

The version stamp object is called *ProWVerstamp* because it is a WHandle, or workspace handle. It is a workspace handle because the data structure it references is not the one in the Pro/ENGINEER database, but a copy taken from it, which is private to the Pro/TOOLKIT application.

Functions Introduced:

- **ProMdlVerstampGet()**
- **ProFeatureVerstampGet()**
- **ProVerstampAlloc()**
- **ProVerstampFree()**
- **ProVerstampStringGet()**
- **ProVerstampStringFree()**
- **ProStringVerstampGet()**
- **ProVerstampEqual()**

The functions **ProMdlVerstampGet()** and **ProFeatureVerstampGet()** enable you to make a workspace copy of the version stamp on a particular model or feature. The function **ProMdlVerstampGet()** is currently applicable to solids only (parts or assemblies). Both of these functions allocate the space for the workspace object internally. After using the contents of the version stamp object, you can free the workspace memory using **ProVerstampFree()**.

If you want to store a copy of a version stamp to compare to a newly read version later, you should use the nonvolatile representation, which is a C string. The function **ProVerstampStringGet()** allocates and fills a string that represents the contents of the specified *ProWVerstamp* object. The **ProStringVerstampGet()** function performs the reverse translation: it allocates a new *ProWVerstamp* object and fills it by copying the specified C string.

The function **ProVerstampEqual()** compares two *ProWVerstamp* objects to tell you whether the version stamps they represent are equal.

Note: The version stamp on a feature can change not only when the feature definition changes, but also when the feature geometry changes as a result of a change to a parent feature.

Layers

Pro/TOOLKIT implements two data types that enable access to layer information in Pro/ENGINEER:

- *ProLayer*—A DHandle that identifies a layer. The *ProLayer* object is an instance of *ProModelitem*.
- *ProLayerItem*—A DHandle that identifies a layer item. The valid types of layer item are contained in the enumerated type *ProLayerType*.

Functions Introduced:

- **ProMdlLayerGet()**
- **ProMdlLayerVisit()**
- **ProMdlLayersCollect()**
- **ProLayerCreate()**
- **ProLayerDelete()**

- **ProLayerItemsGet()**
- **ProLayerItemsPopulate()**
- **ProLayerItemarrayFree()**
- **ProLayerItemInit()**
- **ProDwgLayerItemInit()**
- **ProLayerItemAdd()**
- **ProLayerItemRemove()**
- **ProLayerItemLayersGet()**
- **ProLayerDisplaystatusGet()**
- **ProLayerDisplaystatusSet()**
- **ProDwgLayerDisplaystatusGet()**
- **ProDwgLayerDisplaystatusSet()**
- **ProLayerDisplaystatusSave()**
- **ProLayerDefLayerSet()**
- **ProLayerDefLayerGet()**
- **ProLayerViewDependencySet()**
- **ProLayerViewDependencyGet()**

To get the *ProLayer* object for a layer with the specified name and owner, call the function **ProMdlLayerGet()**. You must pass the name of the layer as a wide string.

To visit the layers in a model, use the function **ProMdlLayerVisit()**. As with other Pro/TOOLKIT visit functions, you supply the visit action and visit filter functions.

The function **ProMdlLayersCollect()** collects a *ProArray* of layers in the model.

The function **ProLayerCreate()** creates a new layer with a specified name. It requires as input the *ProMdl* handle for the model that will own the layer. The function **ProLayerDelete()** deletes the layer identified by the specified *ProLayer* object.

The function **ProLayerItemsGet()** allocates and fills an array of *ProLayerItem* objects that contains the items assigned to the specified layer.

Note: The function **ProLayerItemsGet()** is deprecated. For a large number of layer items, the function **ProLayerItemsGet()** may return an error **PRO_TK_OUT_OF_MEMORY** to indicate that the function was unable to allocate a *ProArray* to hold all of the layer items. To address this issue, use the new function **ProLayerItemsPopulate()**.

The function **ProLayerItemsPopulate()** allocates and fills an array of *ProLayerItem* objects that contain the type and identifier of the items assigned to the specified layer. This function can retrieve a large number of items specified on the layer. Use the function **ProLayeritemarrayFree()** to free the allocated memory.

To initialize a *ProLayerItem*, call the function **ProLayerItemInit()**. This function should be used in all cases, except when *all* of the following are true:

- The layer owner is a drawing.
- The layer item owner is an assembly.
- The layer item is a component.
- You want to control the display status of this component only in a subassembly with a given path.

When all of the above conditions are true, use the function **ProDwgLayerItemInit()** to initialize the *ProLayerItem*.

To add items to a layer, call the function **ProLayerItemAdd()**, and pass as input a *ProLayer* object and the *ProLayeritem* object for the new layer item. To remove an item from a layer, use the function **ProLayerItemRemove()** and specify the *ProLayeritem* object for the item to remove.

To find all the layers containing a given layer item, use the function **ProLayeritemLayersGet()**. This function supports layers in solid models and in drawings.

As in an interactive session of Pro/ENGINEER, one of the principal reasons to create a layer is to display or blank its member items selectively. The function **ProLayerDisplaystatusGet()** obtains the display status of the specified layer, in the form of the *ProLayerDisplay* enumerated type. The display status can be of following types:

- **PRO_LAYER_TYPE_NONE**—The selected layer is displayed. This is the default display status.
- **PRO_LAYER_TYPE_NORMAL**—The layer selected by the user is displayed.

- **PRO_LAYER_TYPE_DISPLAY**—The selected layer is isolated.
- **PRO_LAYER_TYPE_BLANK**—The selected layer is blanked.
- **PRO_LAYER_TYPE_HIDDEN**—The components in the hidden layers are blanked. This status is applicable only in the assembly mode.

To modify the display status of a layer, call the function **ProLayerDisplaystatusSet()**.

Note: **ProLayerDisplaystatusSet()** does not repaint the model after it modifies the display status. This is a temporary setting. It will be lost after you save or retrieve the model. To permanently change the display status, call the function **ProLayerDisplaystatusSave()**.

Unique functions are required to retrieve and set the status of layers in drawings. **ProDwgLayerDisplaystatusGet()** takes as input the layer handle and drawing view. The function **ProDwgLayerDisplaystatusSet()** takes an additional argument as input—the desired display status.

The function **ProLayerDisplaystatusSave()** saves the changes to the display status of all the layers in the specified owner. In addition, the display statuses are saved in the owner's submodels and drawing views.

To set up a default layer with a specified name, call the function **ProLayerDefLayerSet()**. This function requires the default layer type, which is defined in the enumerated type *ProDefLayerType*. To get the name of the default layer with the specified type, call the function **ProLayerDefLayerGet()**.

The function **ProLayerViewDependencySet()** sets the display of layers of the specified view to depend on the display of layers in the drawing. The syntax of this function is as follows:

```
ProLayerViewDependencySet (
    ProView      view,
    ProBoolean    depend);
```

If *depend* is set to PRO_B_TRUE, the layers in the view will be displayed when the layers in the drawing are displayed. If *depend* is set to PRO_B_FALSE, the layer display in the view will be independent of the display in the drawing. To determine whether the layer display in the view is dependent on the display in the drawing, call the function **ProLayerViewDependencyGet()**.

Example 3: Creating a Layer

This example shows how to create a layer and add items to it. This example streamlines the layer creation process (compared to interactive Pro/ENGINEER) because the application creates the layer and adds items to it in only one step. Note that this example does not allow users to add a subassembly to the new layer.

```
/*-----*\
Pro/TOOLKIT includes
\*-----*/
#include <ProToolkit.h>
#include <ProGraphic.h>
#include <ProLayer.h>
#include <ProObjects.h>
#include <ProSelection.h>
#include <ProMenu.h>

#include <TestError.h>

/*=====*\
FUNCTION:  UserLayerItemAction()
PURPOSE:  Close the menu and reports the selected menu action.
\*=====*/
int UserLayerItemAction (ProAppData dummy, int action)
{
    ProMenuDeleteWithStatus (action);
}

/*=====*\
FUNCTION:  UserLayerItemTypeChoose()
PURPOSE:  Prompt the user to select an element type to add to
          the layer.
\*=====*/
int UserLayerItemTypeChoose()
{
    int      menu_id;
    int      action;
    ProError err;

    err = ProMenuFileRegister ("ublack6", "ublack6.mnu", &menu_id);

    err = ProMenubuttonActionSet ("ublack6", "Part",
        UserLayerItemAction, NULL, PRO_LAYER_PART);
```

```

    err = ProMenubuttonActionSet ("ublack6", "Feature",
        UserLayerItemAction, NULL, PRO_LAYER_FEAT);

    err = ProMenubuttonActionSet ("ublack6", "Curve",
        UserLayerItemAction, NULL, PRO_LAYER_CURVE);

    err = ProMenubuttonActionSet ("ublack6", "Quilt",
        UserLayerItemAction, NULL, PRO_LAYER_QUILT);

    err = ProMenubuttonActionSet ("ublack6", "Point",
        UserLayerItemAction, NULL, PRO_LAYER_POINT);

    err = ProMenubuttonActionSet ("ublack6", "ublack6",
        UserLayerItemAction, NULL, -1);

    ProMenuCreate (PROMENUTYPE_MAIN, "ublack6", &menu_id);
    ProMenuProcess ("", &action);
    return (action);
}
/*=====*\
FUNCTION:  UserLayerCreate()
PURPOSE:  Create a layer and add items to it.
\*=====*/
int UserLayerCreate()
{
    #define MAX_BUFFER_LENGTH 20
    ProLayer      layer;
    ProName       layer_name;
    ProLayerItem  layer_item;
    ProMdl        object, member;
    ProSelection  *sel;
    ProModelitem  model_item;
    int           type;
    int           nsel;
    int           m;
    char          *option;
    ProFileName   msg_file;
    ProCharName   str1;
    ProCharLine   str2;
    ProError      err;

    ProStringToWstring (msg_file, "msg_ugfund.txt");
/*-----*\
    Get the handle for the active model.
\*-----*/
    err = ProMdlCurrentGet (&object);
    if (err != PRO_TK_NO_ERROR)
    {
        ProMessageDisplay (msg_file, "USER %0s",
            "Error getting current model.");
    }

```

```

        return (err);
    }
/*-----*\
    Get the layer name from the user.
\*-----*/
    ProMessageDisplay (msg_file, "USER %0s",
        "Enter name of new layer: ");
    ProMessageStringRead (MAX_BUFFER_LENGTH, layer_name);
/*-----*\
    Create the new layer.
\*-----*/
    err = ProLayerCreate (object, layer_name, &layer);
    sprintf (str2, "New layer %s created.",
        ProWstringToString (str1, layer_name));
    if (err == PRO_TK_NO_ERROR)
    {
        ProMessageDisplay (msg_file, "USER %0s", str2);
    }
    else
    {
        ProMessageDisplay (msg_file, "USER %0s",
            "Error creating new layer.");
        return (err);
    }
/*-----*\
    Choose the type of element to be selected.
\*-----*/
    ProMessageDisplay (msg_file, "USER %0s",
        "Select type of items to add.");
    type = UserLayerItemTypeChoose ();
    if (type < 0)
        return (0);
/*-----*\
    Set the appropriate ProSelect() option.
\*-----*/
    switch (type)
    {
    case PRO_LAYER_PART      : option = "part"      ; break;
    case PRO_LAYER_FEAT      : option = "feature"   ; break;
    case PRO_LAYER_CURVE     : option = "curve"     ; break;
    case PRO_LAYER_QUILT     : option = "dtmqlt"    ; break;
    case PRO_LAYERPOINT      : option = "point"     ; break;
    }
    while (ProSelect (option, 1, NULL, NULL, NULL, NULL, &sel, &nssel ) ==
        PRO_TK_NO_ERROR && nssel > 0)
    {
        err = ProSelectionModelitemGet (sel[0], &model_item);
        err = ProLayerItemInit (type, model_item.id, object,
            &layer_item);
        err = ProLayerItemAdd (&layer, &layer_item);
        ProWindowRepaint (-1);
    }

```

```

    }
    return (0);
}

```

Layouts

Functions Introduced:

- **prodb_declare_layout()**
- **prodb_undeclare_layout()**

These two functions retain the Pro/DEVELOP style.

Visiting Displayed Entities

Functions Introduced:

- **ProSolidDispCompVisit()**
- **ProAsmcomppathDispPointVisit()**
- **ProAsmcomppathDispCurveVisit()**
- **ProAsmcomppathDispCsysVisit()**
- **ProAsmcomppathDispQuiltVisit()**

The functions in this section enable you to find quickly all the entities (points, datum curves, coordinate systems, and quilts) currently displayed in an assembly. It is possible to do this using the regular Pro/TOOLKIT functions for visiting assembly components and entities, together with the *ProLayer* functions explained earlier in this chapter; but the functions described here are much more efficient because they make use of Pro/ENGINEER's internal knowledge of the display structures.

The function **ProSolidDispCompVisit()** traverses the components at all levels in an assembly which are not blanked by a layer. The visit action function is called on both the downward traversal and the upward one, and is given a boolean input to distinguish them. It is also given the assembly path and the solid handle to the current subassembly. The subassembly could be found from the path using **ProAsmcomppathMdlGet()**, of course, but Pro/ENGINEER passes this to the action function to allow greater efficiency.

The functions **ProAsmcomppathDisp*Visit()** visit the entities in a subassembly that are not blanked by a layer at any level in the root assembly.

Example 4: Collecting Displayed Assembly Datum Points

This example shows a utility function **UsrDisppointsCollect()** that collects datum points currently displayed in an assembly.

```

/*=====*\
FUNCTION: UsrPointAction()
PURPOSE:  Action callback for visiting displayed points -
          to add a point to ProArray of points
\*=====*/
ProError UsrPointAction(
ProPoint point,
ProError filt_status,
ProAppData data)
{
ProArrayObjectAdd(data, -1, 1, &point);

return(PRO_TK_NO_ERROR);
}

/*=====*\
FUNCTION: UsrAsmAction()
PURPOSE:  Action callback for visiting displayed components -
          Allocates a new ProSelection for the point and add
          it to the array
\*=====*/
ProError UsrAsmAction(ProAsmcomppath *path,
                     ProSolid solid,
                     ProBoolean down,
                     ProAppData data)
{
ProPoint *points;
int n_points, p, id;
ProModelitem modelitem;
ProSelection sel;

/*-----*\
    If the assembly traversal is on the way up, skip
    this component
\*-----*/
if(!down)
return(PRO_TK_NO_ERROR);

/*-----*\
    Collect all the displayed points in this component.
\*-----*/

```

```

ProArrayAlloc(0, sizeof(ProPoint), 1, (ProArray*)&points);
ProAsmcomppathDispPointVisit(path, solid, NULL,
                             UsrPointAction, &points);
/*-----*\
    Represent each point as a ProSelection and add it to
    the collection
\*-----*/
ProArraySizeGet(points, &n_points);
for(p=0;p<n_points;p++)
{
ProPointIdGet(points[p], &id);
ProModelitemInit(solid, id, PRO_POINT, &modelitem);
ProSelectionAlloc(path, &modelitem, &sel);
ProArrayObjectAdd(data, -1, 1, &sel);
}

ProArrayFree((ProArray*)&points);

return(PRO_TK_NO_ERROR);
}
/*=====*\
FUNCTION:UsrDisppointsCollect()
PURPOSE: Utility to collect display points in an assembly
\*=====*/
ProError UsrDisppointsCollect(
ProSolid assembly,
ProSelection **points)
{
int n_points;

ProArrayAlloc(0, sizeof(ProSelection), 1, (ProArray*)points);

ProSolidDispCompVisit(assembly, UsrAsmAction, NULL, points);

ProArraySizeGet(*points, &n_points);
return(n_points > 0 ? PRO_TK_NO_ERROR : PRO_TK_E_NOT_FOUND);
}

```


3

Core: Solids, Parts, and Materials

This chapter describes how to access solids, parts, and their contents.

Topic	Page
Solid Objects	3 - 2
Part Objects	3 - 19
Material Objects	3 - 21

Solid Objects

The Pro/ENGINEER term “solid” denotes a part or an assembly. The object is called *ProSolid* and is declared as an opaque handle. It is an instance of *ProMdl* and can be cast to that type to use functions that have the prefix “*ProMdl*.”

Creating a Solid

Function Introduced:

- **ProSolidCreate()**

The function **ProSolidCreate()** creates an empty part or assembly with the specified name, and provides a handle to the new object. It does not make the new solid current, nor does it display the solid.

Contents of a Solid

Functions Introduced:

- **ProSolidFeatVisit()**
- **ProSolidSurfaceVisit()**
- **ProSolidQuiltVisit()**
- **ProSolidAxisVisit()**
- **ProSolidCsysVisit()**
- **ProSolidFeatstatusGet()**
- **ProSolidFeatstatusSet()**
- **ProSolidFeatstatusWithoptionsSet()**
- **ProSolidFeatstatusflagsGet()**
- **ProSolidFailedFeatsList()**
- **ProSolidFailedfeaturesList()**

The following visit functions enable you to access the various types of objects inside a part or assembly:

- **ProSolidFeatVisit()**—Visits all the features, including those used internally (which are not visible to the Pro/ENGINEER user). You can also use this function to visit the components of an assembly.
- **ProSolidSurfaceVisit()**—Visits all the surfaces of a part. This includes all surfaces created by solid features, but not datum surfaces.

- **ProSolidQuiltVisit()**—Visits all the quilts in a part or an assembly.
- **ProSolidAxisVisit()**—Visits all the axes in a part or an assembly.
- **ProSolidCsysVisit()**—Visits all the coordinate system datums in a part or an assembly.

The function **ProSolidFeatstatusGet()** retrieves a list of the integer identifiers and statuses of all the features in a specified solid in the order in which they are regenerated. The integer identifier of a feature is the value of the *id* field in the *ProFeature* object and also the INTERNAL ID seen in Pro/ENGINEER.

The function **ProSolidFeatstatusSet()** enables you to set the regeneration order and statuses of the features in the solid.

The function **ProSolidFeatstatusWithoptionsSet()** assigns the regeneration order and status bit flags for the specified features in a solid based on the bitmask containing one or more regeneration control bit flags of the type `PRO_REGEN_*` defined in *ProSolid.h*. Refer to the “Regenerating a Solid” section for more information on the bit flags.

The function **ProSolidFeatstatusflagsGet()** retrieves the array of integer identifiers of the features in a specified solid and the corresponding array of bitmasks representing one or more feature status bit flags of the type `PRO_FEAT_STAT_*` defined in *ProFeature.h*. Refer to the “Core: Features” chapter for more information on the feature status bit flags.

The function **ProSolidFailedFeatsList()** retrieves the list of identifiers of failed features in a specified solid.

Note: From Pro/ENGINEER Wildfire 5.0 onward, the function **ProSolidFailedFeatsList()** has been deprecated. Use the function **ProSolidFailedfeaturesList()** instead. Pass `NULL` for the input arguments *co_failed_ids* and *co_x_failed_ids* while using **ProSolidFailedfeaturesList()** in the Resolve mode.

The function **ProSolidFailedfeaturesList()** retrieves the list of identifiers of all or any of the failed features, children of failed features, children of external failed features, or both the features and their children.

Displaying a Solid

Function Introduced:

- **ProSolidDisplay()**

The function **ProSolidDisplay()** displays a solid in the current Pro/ENGINEER window. This does not make the object current from the point of Pro/ENGINEER.

Example 1: Loading and Displaying a Solid

This example shows how to use the functions **ProObjectwindowCreate()** and **ProSolidDisplay()**.

```
int UserLoadPart()
{
    ProFamilyName    name;
    ProMdl            part;
    ProFileName      msgfil;
    ProError          err;
    int               status;

    ProStringToWstring (msgfil, "msg_ug6.txt");
    /*-----*\
    Get the name of the part from the user.
    \*-----*/
    ProMessageDisplay (msgfil, "USER Enter the part name [QUIT] : ");
    err = ProMessageStringRead (PRO_FAMILY_NAME_SIZE, name);
    if (err != PRO_TK_NO_ERROR)
        return (0);
    /*-----*\
    Retrieve the part from disk.
    \*-----*/
    err = ProMdlRetrieve (name, (ProMdlType)PRO_PART, &part);
    /*-----*\
    Check that the part was retrieved.
    \*-----*/
    if (err != PRO_TK_NO_ERROR)
    {
        ProMessageDisplay (msgfil, "USER Failed to retrieve part
        %0w.prt",name);
        return (0);
    }
    /*-----*\
    Open a window and display the part in it.
    \*-----*/
    ProObjectwindowCreate (name, PRO_PART, NULL);
    ProSolidDisplay (part);
}
```

Regenerating a Solid

Function Introduced:

- **ProSolidRegenerate()**
- **ProSolidRegenerationIsNoresolveMode()**
- **ProSolidRegenerationstatusGet()**

The function **ProSolidRegenerate()** regenerates the specified solid. One of the inputs to the function is a bitmask that specifies how the regeneration must be performed. The bitmask may contain the following flags:

- **PRO_REGEN_NO_FLAGS**—Equivalent to passing no flags.
- **PRO_REGEN_CAN_FIX**—Allows the user to interactively fix the model using the user interface, if regeneration fails. This bit flag needs to be set only in case of interactive applications. If this option is not included, the user interface does not update even if regeneration is successful. Use **ProWindowRepaint()** and **ProTreetoolRefresh()** to perform the update if needed. Also, this bit flag must be set only in the Resolve mode. Otherwise, **ProSolidRegenerate()** returns **PRO_TK_BAD_CONTEXT**.
- **PRO_REGEN_ALLOW_CONFIRM**—Allows the user to interactively select the option of retaining failed features and children of failed features via a pop-up dialog box, if regeneration fails. This bit flag must be set only in the No-Resolve mode. Otherwise, **ProSolidRegenerate()** returns **PRO_TK_BAD_CONTEXT**.
- **PRO_REGEN_UNDO_IF_FAIL**—Allows the user to undo the failed regeneration and restore the previous status. This flag needs to be set only in the No-Resolve mode. Otherwise, **ProSolidRegenerate()** returns **PRO_TK_BAD_CONTEXT**. The result obtained may be different from the one attained by using the **Restore** option in the Resolve mode. **Restore** in the Resolve mode can be used immediately after the first failure. But undo in the No-Resolve mode due to this bit flag happens only after all the features are regenerated or failed. In some cases, the undo may not happen at all.

Note: The bit flags **PRO_REGEN_ALLOW_CONFIRM** and **PRO_REGEN_UNDO_IF_FAIL** are not compatible with each other. Setting both of them together will result in **PRO_TK_BAD_CONTEXT**.

- **PRO_REGEN_SKIP_DISALLOW_SYS_RECOVER**—Skips the preparation for failure recovery. If this option is used, Undo Changes is possible if a failure occurs. This option is used only in conjunction with **PRO_REGEN_CAN_FIX**.
- **PRO_REGEN_UPDATE_INSTS**—Updates instances of the solid in memory. This may slow down the regeneration process.
- **PRO_REGEN_RGN_BCK_USING_DISK**—Stores the backup model on the disk. This is useful only if **PRO_REGEN_CAN_FIX** is set.
- **PRO_REGEN_FORCE_REGEN**—Forces the solid to fully regenerate. This will regenerate every feature in the solid. If not set, Pro/ENGINEER uses its internal algorithm to determine which features to regenerate.
- **PRO_REGEN_UPDATE_ASSEMBLY_ONLY**—Updates assembly and sub-assembly placements and regenerates assembly features and intersected parts. If the affected assembly is retrieved as a simplified representation, this flag will update the locations of the components. If the flag is not set, the component locations are not updated by default when the simplified representation is retrieved.

Note: This flag cannot be used with **PRO_REGEN_FORCE_REGEN**.

- **PRO_REGEN_RESUME_EXCL_COMPS**—Enables Pro/ENGINEER to resume available excluded components of the simplified representation during regeneration. This can result in a more accurate update of the simplified representation.

Note: Component models which are not in session at the time of the call to **ProSolidRegenerate()** will not be retrieved due to this option.

- **PRO_REGEN_NO_RESOLVE_MODE**—Specifies the No-Resolve mode introduced in Pro/ENGINEER Wildfire 5.0. This is the default mode in Pro/ENGINEER Wildfire 5.0. In this mode, if a model and feature regeneration fails, failed features and children of failed features are created and regeneration of other features continues.

- **PRO_REGEN_RESOLVE_MODE**—Specifies the Resolve mode. In this mode, you can continue with the Pro/ENGINEER Wildfire 4.0 behavior, wherein if a model and feature regeneration fails, the failure needs to be resolved before regeneration can be resumed. You can also switch to the Resolve mode by setting the configuration option `regen_failure_handling` to `resolve_mode` in the Pro/ENGINEER session.

Note: Setting the configuration option to switch to Resolve mode ensures the old behavior as long as you do not retrieve the models saved under the No-Resolve mode. To consistently preserve the old behavior, use Resolve mode from the beginning and throughout your Pro/ENGINEER session. Temporarily setting the bit flag **PRO_REGEN_RESOLVE_MODE** in the relevant functions does not ensure the old behavior.

The function **ProSolidRegenerationIsNoresolvemode()** identifies if the regeneration mode in the active Pro/ENGINEER session is the No-Resolve mode. Set the *ProBoolean* argument *is_no_resolve* to `PRO_B_TRUE` to set the No-Resolve mode.

The function **ProSolidRegenerationstatusGet()** returns the regeneration status of the solid model. This status is similar to the regeneration status indicator (Green/Yellow/Red) in the Pro/ENGINEER User Interface. The regeneration status can take one of the following values:

- `PRO_SOLID_REGENERATED`
- `PRO_SOLID_NEEDS_REGENERATION`
- `PRO_SOLID_FAILED_REGENERATION`

Note: Models with certain contents, such as circular references or assembly analysis features, will never return a fully “regenerated” status. Thus, this status should not provide an absolute restriction. If the flag remains in the “`PRO_SOLID_NEEDS_REGENERATION`” status through two successful regenerations, the model could be considered up-to-date.

Example 2: Combining Regeneration Flags

This example shows how to use the function **ProSolidRegenerate()**.

```
#include <ProToolkit.h>
#include <ProSolid.h>
```

```

/*=====*\
FUNCTION: UserSolidFullRegenerate()
PURPOSE: Fully regenerates the indicated model -
prompt for fix model on failure
/*=====*/
ProError UserSolidFullRegenerate (ProSolid solid, ProBoolean allow_fix)
{
ProError status;
int regeneration_flags = PRO_REGEN_FORCE_REGEN;

/*-----*\
Add the "Can fix" flag to the regeneration options
/*-----*/
if (allow_fix)
regeneration_flags |= PRO_REGEN_CAN_FIX;
status = ProSolidRegenerate (solid, regeneration_flags);
return PRO_TK_NO_ERROR;
}

```

Evaluating Mathematical Expressions for a Solid

Functions Introduced:

- **ProMathExpressionEvaluate()**

The function **ProMathExpressionEvaluate()** evaluates the given mathematical expression in the context of a given solid. The expression may include parameters, dimensions, embedded functions, or predefined constants. This function returns a pointer to the calculated result and a pointer to the unit of the calculated result.

Solid Outline

Functions Introduced:

- **ProSolidOutlineGet()**
- **ProSolidOutlineCompute()**

The function **ProSolidOutlineGet()** provides you with the maximum and minimum values of X, Y, and Z occupied by the contents of the solid, with respect to the default, solid coordinate system.

The function **ProSolidOutlineCompute()** calculates the outline of the solid with respect to any orientation, defined by a transformation matrix. (For more information, see the Core: Coordinate Systems and Transformations chapter.) The function enables you to exclude from the calculation items of any or all of the following types:

- Datum plane
- Datum point
- Datum axes
- Datum coordinate system
- Facets

Example 3: Computing the Outline of a Solid

This example computes the outline of a solid with respect to a selected coordinate system, and converts the result back to solid coordinates.

```

/*=====*\
FUNCTION: UserOutlineCompute()
PURPOSE:  Display the extents of a solid in space defined by the
          CSYS.
\*=====*/
int UserOutlineCompute()
{
    int                sel_count;
    ProError           status;
    ProFileName        msgfile;
    ProSelection       *psels = NULL;
    ProModelitem       csys_feat;
    ProGeomitemdata    *geom_data = NULL;
    ProCsysdata        *p_csys = NULL;
    ProMdl             solid;
    Pro3dPnt           outline[2];
    ProMatrix          transf, itranf;
    ProSolidOutlExclTypes excludes[] = {PRO_OUTL_EXC_DATUM_PLANE,
                                         PRO_OUTL_EXC_DATUM_POINT,
                                         PRO_OUTL_EXC_DATUM_CSYS};

/*-----*\
    Request a coordinate system as a reference point and
    orientation to report outline points.
\*-----*/
    ProStringToWstring (msgfile, "msg_ug7.txt");
    status = ProMessageDisplay (msgfile, "UG7 Select CSYS as
        reference point");
    if ((ProSelect ("csys", 1, NULL, NULL, NULL, NULL, &psels,
        &sel_count) != PRO_TK_NO_ERROR) || (sel_count < 1))
    {
        status = ProMessageDisplay (msgfile, "UG7 Invalid Selection!");
        return ((int)PRO_TK_GENERAL_ERROR);
    }

/*-----*\
    Retrieve the coordinate system data.
\*-----*/

```

```

status = ProSelectionModelitemGet (psels[0], &csys_feat);
status = ProGeomitemdataGet (&csys_feat, &geom_data);
if (geom_data->obj_type != PRO_CSYS)
{
    status = ProMessageDisplay (msgfile,
        "UG7 Invalid Feature Selected as reference point");
    return ((int)PRO_TK_GENERAL_ERROR);
}

p_csys = geom_data->data.p_csys_data;
ProUtilVectorsToTransf (p_csys->x_vector, p_csys->y_vector,
    p_csys->z_vector, p_csys->origin, transf);
ProUtilMatrixInvert (transf, itranf);
status = ProMdlCurrentGet (&solid);
status = ProSolidOutlineCompute (solid, transf, excludes, 3,
    outline);
status = ProMessageDisplay (msgfile,
    "UG7 Outline points: (%0f %1f %2f) (%3f %4f %5f)",
    &outline[0][0], &outline[0][1], &outline[0][2],
    &outline[1][0], &outline[1][1], &outline[1][2]);
return ((int)status);
}

```

Solid Accuracy

Functions Introduced:

- **ProSolidAccuracyGet()**
- **ProSolidAccuracySet()**

Use the functions **ProSolidAccuracyGet()** and **ProSolidAccuracySet()** to retrieve and set the accuracy of a specified part or assembly, respectively.

Note: To set or retrieve the accuracy for an assembly you must traverse through all its parts in the assembly with these functions.

The input arguments for the function **ProSolidAccuracySet()** are as follows:

- *solid*—The part or assembly whose accuracy you want to set.
- *type*—The type of the accuracy. The valid values are:
 - PRO_ACCURACY_RELATIVE—Specifies the relative accuracy
 - PRO_ACCURACY_ABSOLUTE—Specifies the absolute accuracy

- *accuracy*—The value of the accuracy that you want to set. The unit used for the absolute accuracy of the dimension is based on the unit of the part or assembly.

Note: Regenerate the model using the function **ProSolidRegenerate()** after setting the accuracy using **ProSolidAccuracySet()**.

The function **ProSolidAccuracyGet()** returns the type and value of the accuracy. The accuracy may be relative or absolute.

Using these functions to set and retrieve part accuracy is similar to performing these functions in Pro/ENGINEER using **File > Properties > Model Properties**.

Derive the geometry epsilon for the required relative accuracy as follows:

$$geometry\_epsilon = model\_size \times relative\_part\_accuracy$$

Solid Units

Introduction to Unit of Measurement and System of Units

Each model has a basic system of unit to ensure that all material properties of that model are consistently measured and defined. All models are defined on the basis of system of units. A part can have only one system of unit.

Following are the types of quantities which govern the definition of unit of measurement:

- **Basic Quantities**—The basic units and dimensions of the system of units. For example, consider the Centimeter Gram Second (CGS) system of unit. The basic quantity for this system of unit is:
 - Length—cm
 - Mass—g
 - Force—dyne
 - Time—sec
 - Temperature—K
- **Derived Quantities**—The derived units are those that are derived from the basic quantities. For example, consider the Centimeter Gram Second (CGS) system of unit. The derived quantities for this system of unit are as follows:
 - Area—cm²

- Volume—cm³
- Velocity—cm/sec

Types of Systems of Units

Following are the types of system of units:

- Pre-defined system of unit—This system of unit is provided by default.
- Custom-defined system of unit—This system of unit is defined by the user only if the model does not contain standard metric or nonmetric units or if the material file contains units that cannot be derived from the predefined system of units or both.

In PRO/ENGINEER, the system of units are categorized as follows:

- Mass Length Time (MLT)

The following systems of units belong to this category:

- CGS —Centimeter Gram Second
- MKS—Meter Kilogram Second
- mmKS—millimeter Kilogram Second

- Force Length Time (FLT)

The following systems of units belong to this category:

- Pro/ENGINEER Default—Inch lbm Second. This is the default system followed by Pro/ENGINEER.
- FPS—Foot Pound Second
- IPS—Inch Pound Second
- mmNS—Millimeter Newton Second

Definitions

For Pro/TOOLKIT, a system of units is represented by the structure *ProUnitsystem*. This structure is defined as:

```
typedef struct {
    ProMdl  owner;
    ProName name;
}ProUnitsystem;
```

where the name is the user-visible name used in the Unit Manager dialog.

An individual unit is represented by the structure *ProUnititem*. This structure is defined as:

```
typedef struct {
    ProMdl  owner;
    ProName name;
}ProUnititem;
```

where the name is the user-visible abbreviation used in the Unit Manager dialog.

Note: The functions described in the following sections supersede the functions **prodb_get_model_units()** and **prodb_set_model_units()**.

Retrieving Systems of Units

Functions Introduced:

- **ProMdlUnitsystemsCollect()**
- **ProMdlPrincipalunitsystemGet()**

Use the function **ProMdlUnitsystemsCollect()** to retrieve the set of systems of units which are accessible to the model in the form of an array. The input arguments of the function are as follows:

- *mdl*—Specifies a handle to the model.

The function outputs a *ProArray* containing the set of systems of units for the specified model.

Note: The function retrieves both the pre-defined as well as the custom-defined system of unit.

Use the function **ProMdlPrincipalunitsystemGet()** to retrieve the principal system of units for the specified model.

Modifying Systems of Units

Functions Introduced:

- **ProUnitssystemRename()**
- **ProUnitssystemDelete()**

Use the function **ProUnitssystemRename()** to rename a custom-defined system of unit. The input parameters for this function are as follows:

- *system*—Specifies a handle to the system of unit.

- *new_name*—Specifies the new name for the system.

Use the function **ProUnitsystemDelete()** to delete a custom-defined system of unit. Specify a handle to the system of units to be deleted as the input parameter for this function.

Note: You can only delete a custom-defined system of units. You cannot delete a pre-defined system of units.

Accessing Systems of Units

Functions Introduced:

- **ProUnitsystemIsStandard()**
- **ProUnitsystemTypeGet()**
- **ProUnitsystemUnitGet()**

Use the function **ProUnitsystemIsStandard()** to check whether the system of unit is a Pro/ENGINEER standard system. Specify the name of the system of unit as the input parameter.

Use the function **ProUnitsystemTypeGet()** to retrieve the type of system of unit. Specify the name of the system of unit as the input argument. The output argument of this function is as follows:

- *type*—The type of system of unit. It can have the following values:
 - PRO_UNITSYSTEM_MLT—Mass Length Time
 - PRO_UNITSYSTEM_FLT—Force Length Time

For more information on the same refer to the section on *Types of Systems of Units* above.

Use the function **ProUnitsystemUnitGet()** to retrieve the unit of particular type for a specified system of unit.

Creating a New System of Units

Function Introduced:

- **ProMdlUnitsystemCreate()**

Use the function **ProMdlUnitsystemCreate()** to create a new system of unit or to create a copy of an existing system of unit. The function expects the following input parameters:

- *mdl*—Specifies a handle to the model.
- *type*—Specifies the new type of system of unit.

- *name*—Specifies the name of the new system of unit.
- *units*—Specifies the set of units for the new system of unit created.

It outputs the newly created system of unit.

Accessing Individual Units

Functions Introduced:

- **ProMdlUnitsCollect()**
- **ProUnitInit()**
- **ProUnitTypeGet()**
- **ProUnitConversionGet()**
- **ProUnitIsStandard()**
- **ProUnitExpressionGet()**
- **ProUnitInitByExpression()**

Use the function **ProMdlUnitsCollect()** to retrieve a set of units of a particular type that are available to the specified model.

The function **ProUnitInit()** retrieves the unit of a particular name for a specified model.

Note: The function is applicable only for basic units and not for derived ones.

Use the function **ProUnitTypeGet()** to retrieve the unit type of a particular unit.

Unit types can have any of the following values:

- PRO_UNITTYPE_LENGTH
- PRO_UNITTYPE_MASS
- PRO_UNITTYPE_FORCE
- PRO_UNITTYPE_TIME
- PRO_UNITTYPE_TEMPERATURE
- PRO_UNITTYPE_ANGLE

Use the function **ProUnitConversionGet()** to retrieve the conversion factor for a particular unit. The output arguments of this function are:

- *conversion*— Specifies the conversion factor for a unit in terms of scale of the unit and an offset value.

Example - Consider the formula to convert temperature from Centigrade to Fahrenheit

$$F = a + (C * b)$$

where

F is the temperature in Fahrenheit

C is the temperature in Centigrade

a = 32 (constant signifying the offset value)

b = 9/5 (ratio signifying the scale of the unit)

Note: Pro/ENGINEER scales the length dimensions of the model using the factor specified. If the scale is modified, the model is regenerated. When you scale the model, the model units are not changed. Imported geometry cannot be scaled.

- *ref_unit*— Specifies the reference unit for the conversion.

Use the function **ProUnitIsStandard()** to determine whether the unit is a standard unit as defined in Pro/ENGINEER.

Pro/ENGINEER uses named quantities to represent units other than the basic units (e.g. "lbs_stress_unit", which represents a quantity of stress in the default Pro/ENGINEER unit systems). Parameters and material properties which are assigned derived units will return the name in the ProUnititem structure, rather than the actual unit-based expression for the quantity.

Use the function **ProUnitExpressionGet()** to retrieve the unit-based expression for a given Pro/ENGINEER unit name.

Use the function **ProUnitInitByExpression()** to retrieve the ProUnititem given a unit-based expression.

Modifying Units

Functions Introduced:

- **ProUnitModify()**
- **ProUnitDelete()**
- **ProUnitRename()**

Use the function **ProUnitModify()** to modify a pre-defined unit. Modifying the units can invalidate your relations, as they are not scaled along with the model. The input parameters are:

- *unit*— Specifies the unit to be modified.
- *conversion*— Specifies the conversion factor for the unit.

- *ref_unit*— Specifies the reference unit.

Use the function **ProUnitDelete()** to delete a previously created unit.

Note: You cannot delete a pre-defined unit. If you delete a unit, you cannot undo the deletion.

Use the function **ProUnitRename()** to rename an existing unit.

Creation of a new Unit

Function Introduced:

- **ProUnitCreate()**

Use the function **ProUnitCreate()** to create a new basic unit given a reference unit and the required conversion factor.

Conversion of Models to a New Unit System

Function Introduced:

- **ProMdlPrincipalunitsystemSet()**

Use the function **ProMdlPrincipalunitsystemSet()** to change the principal system of units assigned to the solid model. The options available for the conversion are:

- **PRO_UNITCONVERT_SAME_DIMS**—Specifies to keep the dimension values despite the change in units.
- **PRO_UNITCONVERT_SAME_SIZE**—Specifies to scale the dimension values to keep the same size for the model.

Conversion of a system of units may result in regeneration failures due to the modification of dimensions, parameters, and relations. **ProMdlPrincipalunitsystemSet()** does not currently support a flag to undo the changes made by the unit system conversion, and will always bring the Fix Model interface to fix any regeneration failure. Therefore this function can be used safely only in interactive mode applications.

Note: The initial units for an assembly are those of its base component. If, however, the units of the base component have been changed, the assembly units do not automatically change. You must also modify the units of the assembly. You cannot change the units of an assembly containing assembly features that intersect a part.

Mass Properties

Function Introduced:

- **ProSolidMassPropertyGet()**

The function **ProSolidMassPropertyGet()** provides information about the distribution of mass in the part or assembly. It can provide the information relative to a coordinate system datum, which you name, or the default one if you provide NULL as the name. It returns the information in a structure called *ProMassProperty*, declared in the header file *ProSolid.h*.

The *ProMassProperty* structure contains the following fields (all doubles):

- *volume*—The volume.
- *surface_area*—The surface area.
- *density*—The density. The density value is 1.0, unless a material has been assigned.
- *mass*—The mass.
- *center_of_gravity[3]*—The center of gravity (COG).
- *coord_sys_inertia[3][3]*—The inertia matrix.
- *coord_sys_inertia_tensor[3][3]*—The inertia tensor.
- *cg_inertia_tensor[3][3]*—The inertia about the COG.
- *principal_moments[3]*—The principal moments of inertia (the eigenvalues of the COG inertia).
- *principal_axes[3][3]*—The principal axes (the eigenvectors of the COG inertia).

Solid Postfix Identifiers

Functions Introduced:

- **ProSolidToPostfixId()**
- **ProPostfixIdToSolid()**

The postfix identifier of a solid is the integer run-time identifier used in relations to make the names of its dimensions unique in the context of a parent assembly. Pro/ENGINEER automatically updates these values when they are used in relations. The function **ProSolidToPostfixId()** gives you the identifier for the solid in session. The **ProPostfixIdToSolid()** function provides the solid handle, given the identifier.

Part Objects

The object *ProPart* is an instance of *ProSolid*. It is an opaque handle that can be cast to a *ProSolid* or *ProMdl* so you can use any of the functions for those objects.

Density

- **ProPartDensitySet()**
- **ProPartDensityGet()**

The density of a part is used in many calculations inside of Pro/ENGINEER, including mass properties calculations and shrinkwrap export. The function **ProPartDensitySet()** sets the density of the part without requiring assignment of a material. The function **ProPartDensityGet()** returns the current density. The density value is 1.0 by default.

Core: Solids, Parts,
and Materials

Example 4: Writing the Mass of a Given Part to the Model Tree

This example demonstrates the use of **ProPartDensityGet()**, **ProSolidMassPropertyGet()** and several units related functions. It writes the mass of the given part to the model tree, along with the appropriate units for the mass value.

```
include <ProToolkit.h>
#include <ProMenuBar.h>
#include <ProMdl.h>
#include <ProPart.h>
#include <ProMdlUnits.h>
#include <ProSolid.h>
#include <ProMessage.h>
#include <ProUtil.h>
#include <ProWstring.h>
#include <ProSelbuffer.h>

#include <TestError.h>

/*-----*\
Application global/external data
\*-----*/

static
wchar_t MSGFIL[] =
{'u','t','i','l','i','t','i','e','s','.','t','x','t','\0'};

/*=====*\
FUNCTION: UserPartInfoMass
```

```

PURPOSE:  Write a part's mass value & units to the message window.
\*=====*/
int UserPartInfoMass (ProPart part)
{
    ProError status;
    double density;
    ProMassProperty massProps;
    ProUnitssystem unitSystem;
    ProUnitssystemType type;
    ProUnititem unit, forceUnit, timeUnit, lengthUnit;
    ProLine massUnitsLabel;

/*-----*\
Check for the density.  If it hasn't been set, inform the user
\*-----*/
status = ProPartDensityGet (part, &density);

if (status != PRO_TK_NO_ERROR)
{
    ProMessageDisplay ( MSGFIL, "USER Part has not been assigned density.");
    return PRO_TK_E_NOT_FOUND;
}

/*-----*\
Extract the mass properties using the default part coordinate system
\*-----*/
status = ProSolidMassPropertyGet (part, NULL, &massProps);

/*-----*\
Extract system of units
\*-----*/
status = ProMdlPrincipalunitssystemGet (part, &unitSystem);

ProUnitssystemTypeGet (&unitSystem, &type);

if (type == PRO_UNITSYSTEM_MLT)
{
    ProUnitssystemUnitGet (&unitSystem, PRO_UNITTYPE_MASS, &unit);

    ProWstringCopy (unit.name, massUnitsLabel, PRO_VALUE_UNUSED);
}
else
{
/*-----*\
If the system of units are Force/Length/Time, format the units into the
correct equivalent for mass.
\*-----*/
ProUnitssystemUnitGet (&unitSystem, PRO_UNITTYPE_FORCE, &forceUnit);
ProUnitssystemUnitGet (&unitSystem, PRO_UNITTYPE_TIME, &timeUnit);
ProUnitssystemUnitGet (&unitSystem, PRO_UNITTYPE_LENGTH, &lengthUnit);

```

```

ProMessageToBuffer (massUnitsLabel, MSGFIL,
"USER Force unit -> mass", &forceUnit.name, &timeUnit.name,
&lengthUnit.name);
}

/*-----*\
Show the mass to the user
\*-----*/
ProMessageDisplay ( MSGFIL, "USER Part mass", &massProps.mass,
&massUnitsLabel);

return PRO_TK_NO_ERROR;}

```

Material Objects

Pro/TOOLKIT enables you to programmatically access the material properties of parts. Using the Pro/TOOLKIT functions, you can do the following actions:

- Create or delete materials.
- Set the current material.
- Retrieve and set the material types and properties.
- Read and write to material files.

To enable access to materials, Pro/TOOLKIT uses the following objects:

- *ProMaterial*—A structure that contains a material name and the part (*ProSolid* object) to which it is assigned. This handle is used in older material functions.
- *ProMaterialItem*—Another name for a *ProModelitem*, it contains the material owner, ID, and type (PRO_RP_MATERIAL).

To convert between *ProMaterial* and *ProMaterialItem*, use **ProModelitemByNameInit()** to obtain the item from the material owner and name. To obtain a *ProMaterial* from a *ProMaterialItem*, use **ProModelitemNameGet()** to get the material name.

Accessing Material Data

Functions Introduced:

- **ProMaterialCreate()**
- **ProPartMaterialsGet()**
- **ProMaterialDelete()**
- **ProMaterialCurrentGet()**
- **ProMaterialCurrentSet()**

The function **ProMaterialCreate()** creates a new material with the name you specify, and sets the default values within an associated *ProMaterialdata* object. Your application must set the correct material properties in the fields of the *ProMaterialdata* structure.

The input arguments of this function are as follows:

- *part*—Specifies the part.
- *matl_name*—Specifies the material name.
- *p_matl_data*—This argument has been deprecated. Pass NULL to create an empty material item whose properties can be set by **ProMaterialPropertySet()**.

The function **ProPartMaterialsGet()** obtains an array containing material names that exist in a part database. Note that you must use **ProArrayAlloc()** to allocate memory for this array. To remove a specified material from the part's database, call the function **ProMaterialDelete()**.

The current material of a part determines the material properties that will be used in some computational analyses of that part. Although multiple materials can be stored in a part database, only one material can be current. The function **ProMaterialCurrentGet()** gets the handle for the current material of the specified part. To change the current material, call the function **ProMaterialCurrentSet()**.

Note:

- By default, while assigning a material to a sheetmetal part, the function **ProMaterialCurrentSet()** modifies the values of the sheetmetal properties such as Y factor and bend table according to the material file definition. This triggers a regeneration and a modification of the developed length calculations of the sheetmetal part. However, you can avoid this behavior by setting the value of the configuration option `material_update_smt_bend_table` to `never_replace`.
- The function **ProMaterialCurrentSet()** may change the model display, if the new material has a default appearance assigned to it.
- The function may also change the family table, if the parameter `PTC_MATERIAL_NAME` is a part of the family table.

Material Types and Properties

The enumerated type *ProMaterialPropertyType* contains the material types and material property types.

The material type is given by `PRO_MATPROP_TYPE` that takes the following values:

- `PRO_MATERIAL_TYPE_STRUCTURAL_ISOTROPIC`—Specifies a material with an infinite number of planes of material symmetry, making the structural material properties equal in all directions.
- `PRO_MATERIAL_TYPE_STRUCTURAL_ORTHOTROPIC`—Specifies a material with symmetry relative to three mutually perpendicular planes for structural material properties.
- `PRO_MATERIAL_TYPE_STRUCTURAL_TRANS_ISOTROPIC`—Specifies a material with rotational symmetry about an axis for structural material properties. These properties are equal for all directions in the plane of isotropy.
- `PRO_MATERIAL_TYPE_THERMAL_ISOTROPIC`—Specifies a material with an infinite number of planes of material symmetry, making the thermal material properties equal in all directions.
- `PRO_MATERIAL_TYPE_THERMAL_ORTHOTROPIC`—Specifies a material with symmetry relative to three mutually perpendicular planes for thermal material properties.

- `PRO_MATERIAL_TYPE_THERMAL_TRANS_ISOTROPIC`—Specifies a material with rotational symmetry about an axis for thermal material properties. These properties are equal for all directions in the plane of isotropy.

The material subtype is given by `PRO_MATPROP_SUB_TYPE` that takes the following values:

- `PRO_MATERIAL_SUB_TYPE_LINEAR`—Specifies the linear elastic material type. This is the default value.
- `PRO_MATERIAL_SUB_TYPE_HYPERELASTIC`—Specifies the hyperelastic (non-linear) material types, such as rubber, that exhibit instantaneous elastic response to large strains.
- `PRO_MATERIAL_SUB_TYPE_ELASTOPLASTIC`—Specifies the elastoplastic (non-linear) material types, such as metals, with the following isotropic hardening laws:
 - Perfect Plasticity—Given by the value `PRO_MATERIAL_HARDENING_PERFECT_PLASTICITY`.
 - Linear Hardening—Given by the value `PRO_MATERIAL_HARDENING_LINEAR_HARDENING`.
 - Power Law—Given by the value `PRO_MATERIAL_HARDENING_POWER_LAW`.
 - Exponential Law—Given by the value `PRO_MATERIAL_HARDENING_EXPONENTIAL_LAW`.

The above three subtypes are available only for `PRO_MATERIAL_TYPE_STRUCTURAL_ISOTROPIC` and `PRO_MATERIAL_TYPE_THERMAL_ISOTROPIC` material types.

Functions Introduced:

- **`ProMaterialPropertyGet()`**
- **`ProUnitExpressionGet()`**
- **`ProMaterialPropertySet()`**
- **`ProUnitInitByExpression()`**
- **`ProMaterialDescriptionGet()`**
- **`ProMaterialDescriptionSet()`**
- **`ProMaterialPropertyDelete()`**

The functions **ProMaterialDataGet()** and **ProMaterialDataSet()** have been deprecated, and do not support all of the available material properties. PTC recommends that for accessing material properties, you convert the *ProMaterial* type to a model item using **ProModelitemByNameInit()**, and use **ProMaterialPropertyGet()** and **ProMaterialPropertySet()** to access and modify the properties of that item, respectively.

The function **ProMaterialPropertyGet()** returns the value and the units for a material property.

Note: The name of the units returned can be the name of a Pro/ENGINEER unit, which may not be obviously understood by a user. Use **ProUnitExpressionGet()** to change this name to familiar units.

Use the function **ProMaterialPropertySet()** to create or modify a material property. It has the following input parameters:

- *p_material*—Specifies the material as defined by *ProMaterialItem*.
- *prop_type*—Specifies the material property type as defined by *ProMaterialPropertyType*.
- *p_value*—Specifies the material property value.
- *p_units*—Specifies the material property units.

Note: This function expects the Pro/ENGINEER unit name for some unit properties. To obtain this name, pass the user-visible units through **ProUnitByExpressionInit()**.

The following table displays the possible combinations of arguments for *p_value* and *p_units*:

p_value	p_units	Is the property already created in the material?	Result
Any value	Appropriate units for this property, or NULL, if the property is unitless	NO	Property is created with the given units and value.
Any value	NULL	NO	Property is created with the given value using the appropriate units from the owner model.

p_value	p_units	Is the property already created in the material?	Result
Any value	Current units for this property, or NULL, if the property is unitless	YES	Property value is changed to the new value.
Any value	NULL	YES	Property value is changed to the new value (which is interpreted as being in the units from the owner model)
The current value	New appropriate units	YES	Property units are changed but the value is interpreted as being for the new units.
NULL	New appropriate units	YES	Property units are changed and the current value is converted to the new units.

Note: When using **ProMaterialPropertySet()** to change the sheetmetal Y-factor or bend table assigned to the current material, pass the current material to **ProMaterialCurrentSet()** again to force Pro/ENGINEER to update the length calculations developed by sheetmetal.

Use the function **ProMaterialDescriptionGet()** to get the material description. This property is also accessible as the material property PRO_MATPROP_MATERIAL_DESCRIPTION.

Use the function **ProMaterialDescriptionSet()** to set the material description.

Use the function **ProMaterialPropertyDelete()** to remove a property from the material definition.

Material Input and Output

Functions Introduced:

- **ProMaterialfileWrite()**
- **ProMaterialfileRead()**

Material properties are frequently stored in text files accessible for repeated assignment to parts. Pro/TOOLKIT includes functions that write to and read these files.

The function **ProMaterialfileWrite()** writes the information contained in a *ProMaterial* object to a file with the specified name.

The format of this file is the new material file format which is consistent with the Pro/ENGINEER materials library.

The function **ProMaterialfileRead()** reads from a material file, the properties of the material with the specified name. The name of the file read can be either:

- *<name>.mtl*—Specifies a new material file format.
- *<name>.mat*—Specifies an old material file format (pre-Wildfire 3.0).

If the material is not already in the part database, **ProMaterialfileRead()** adds the material to the database after reading the material file. If the material is already in the database, the function replaces the material properties in the database with those contained in the material file.

Example 5: Working with Materials and Material Properties

The following example demonstrates how to work with materials and material properties.

```
#include <ProToolkit.h>
#include <ProParameter.h>
#include <ProMaterial.h>
#include <ProUtil.h>
#include <ProSolid.h>
#include <ProMdlUnits.h>

#include <ProSelbuffer.h>
#include <ProUICmd.h>
#include <ProMessage.h>
#include <ProModelitem.h>

/*=====*\
FUNCTION: UserApplyAndUpdateMaterial()
```

PURPOSE: Read a material from file, read some of its properties, and add a user-defined parameter to it. Then assign it to the model and calculate the model mass properties.

```
\*=====*/
ProError UserApplyAndUpdateMaterial (ProPart part)
{
    ProError status;
    ProFileName msg_file;
    ProName fopen_label;
    ProLine filter_string;
    ProPath* shortcut_paths;
    ProName* shortcut_names;
    ProPath mtl_file;
    ProPath mtl_file_path;
    ProPath current_dir;
    ProName mtl_name;
    ProMaterialItem material_item;
    ProParamvalue mtl_type, mtl_density;
    ProUnititem units;
    int mtl_type_value;
    ProParamvalue mtl_prop_value;
    ProName mtl_prop_name;
    ProParameter param;
    ProMaterial material;
    ProPath unit_expression;
    ProMassProperty mass_prop;

/*-----*\
    Set up the message file.
/*-----*/
    ProStringToWstring (msg_file, "msg_ugparam.txt");

/*-----*\
    Get the material file name from the user.
/*-----*/
    ProStringToWstring (fopen_label, "Select material file");
    ProStringToWstring (filter_string, "*.mtl");

    ProArrayAlloc (0, sizeof (ProPath), 1, (ProArray*)&shortcut_paths);
    ProArrayAlloc (0, sizeof (ProName), 1, (ProArray*)&shortcut_names);

    status = ProFileOpen (fopen_label, filter_string, shortcut_paths,
                          shortcut_names, NULL, NULL, mtl_file);

    ProArrayFree ((ProArray*)&shortcut_paths);
    ProArrayFree ((ProArray*)&shortcut_names);

    if (status != PRO_TK_NO_ERROR)
    {
        return (status);
    }
}
```

```

/*-----*\
    Change Pro/ENGINEER to the material file directory
\*-----*/
    ProFilenameParse (mtl_file, mtl_file_path, mtl_name, NULL, NULL);

    ProDirectoryCurrentGet (current_dir);

    ProDirectoryChange (mtl_file_path);

/*-----*\
    Read the material from the file.
\*-----*/
    status = ProMaterialfileRead (part, mtl_name);

    ProDirectoryChange (current_dir);

    if (status != PRO_TK_NO_ERROR)
    {
        ProMessageDisplay (msg_file, "UG MaterialFile Read Error",
                           mtl_name);
        ProDirectoryChange (current_dir);
        return (status);
    }

/*-----*\
    Get the material type and density.
\*-----*/
    status = ProModelitemByNameInit (part, PRO_RP_MATERIAL, mtl_name,
                                     &material_item);

    status = ProMaterialPropertyGet (&material_item, PRO_MATPROP_TYPE,
                                     &mtl_type, &units);

    status = ProMaterialPropertyGet (&material_item,
                                     PRO_MATPROP_MASS_DENSITY,
                                     &mtl_density, &units);

/*-----*\
    Convert the density units to a user-readable format.
\*-----*/
    status = ProUnitExpressionGet (&units, unit_expression);

/*-----*\
    Create a user-defined material property based on the type.
\*-----*/
    ProStringToWstring (mtl_prop_name, "STRUCTURAL_TYPE");

    mtl_type_value = mtl_type.value.i_val;

    mtl_prop_value.type = PRO_PARAM_STRING;
    if (mtl_type_value & PRO_MATERIAL_TYPE_STRUCTURAL_ISOTROPIC)

```

```

{
    ProStringToWstring (mtl_prop_value.value.s_val, "Isotropic");
}
else if (mtl_type_value & PRO_MATERIAL_TYPE_STRUCTURAL_ORTHOTROPIC)
{
    ProStringToWstring (mtl_prop_value.value.s_val, "Orthotropic");
}
else
    ProStringToWstring (mtl_prop_value.value.s_val,
                        "Transversely orthotropic");

    status = ProParameterCreate (&material_item, mtl_prop_name,
                                &mtl_prop_value, &param);
/*-----*\
    Assign the material, and compute the mass properties.
\*-----*/
    material.part = part;
    ProWstringCopy (mtl_name, material.matl_name, PRO_VALUE_UNUSED);

    status = ProMaterialCurrentSet (&material);

    status = ProSolidMassPropertyGet (part, NULL, &mass_prop);

    ProMessageDisplay (msg_file, "UG MaterialFile Mass Info", mtl_name,
                        &mtl_density.value.d_val, unit_expression,
                        &mass_prop.mass);

    return (0);
}

```

Example 6: Creating a Non-linear Material

The following example demonstrates how to create a non-linear material and assign it to a solid.

```

/*----- Pro/Toolkit Includes -----*/

#include <ProToolkit.h>
#include <ProMdl.h>
#include <ProMaterial.h>
#include <ProSolid.h>
#include <ProModelitem.h>

/*-----*\
Application global/external data
\*-----*/

/*=====*\
The structure contains the material properties required for non-linear
material definition. Other properties can be added from ProMaterial.h as
required.

```

```

\*=====*/

typedef struct matl_property_data {

    wchar_t* model;                /*PRO_MATPROP_MODEL,                String*/
    wchar_t* sub_type;             /*PRO_MATPROP_SUB_TYPE              String*/
    ProBool model_def_by_tests;    /*PRO_MATPROP_MODEL_DEF_BY_TESTS,   Boolean*/

    double model_coeff_mu;         /*PRO_MATPROP_MODEL_COEF_MU,        Double*/
    double model_coeff_lm;         /*PRO_MATPROP_MODEL_COEF_LM,        Double*/
    double model_coeff_c01;        /*PRO_MATPROP_MODEL_COEF_C01,       Double*/
    double model_coeff_c02;        /*PRO_MATPROP_MODEL_COEF_C02,       Double*/
    double model_coeff_c10;        /*PRO_MATPROP_MODEL_COEF_C10,       Double*/
    double model_coeff_c11;        /*PRO_MATPROP_MODEL_COEF_C11,       Double*/
    double model_coeff_c20;        /*PRO_MATPROP_MODEL_COEF_C20,       Double*/
    double model_coeff_c30;        /*PRO_MATPROP_MODEL_COEF_C30,       Double*/
    double model_coeff_d;          /*PRO_MATPROP_MODEL_COEF_D,         Double*/
    double model_coeff_d1;         /*PRO_MATPROP_MODEL_COEF_D1,        Double*/
    double model_coeff_d2;         /*PRO_MATPROP_MODEL_COEF_D2,        Double*/
    double model_coeff_d3;         /*PRO_MATPROP_MODEL_COEF_D3,        Double*/

    int type;                      /*PRO_MATPROP_TYPE                  Integer*/
    wchar_t* material_description; /*PRO_MATPROP_MATERIAL_DESCRIPTION  WString*/
    wchar_t* failure_criterion_type; /*PRO_MATPROP_FAILURE_CRITERION_TYPE WString*/
    wchar_t* fatigue_type;         /*PRO_MATPROP_FATIGUE_TYPE          WString*/

    double thermal_expansion_coefficient; /*PRO_MATPROP_THERMAL_EXPANSION_COEFFICIENT Double*/
    double specific_heat;          /*PRO_MATPROP_SPECIFIC_HEAT         Double*/

    double thermal_conductivity;    /*PRO_MATPROP_THERMAL_CONDUCTIVITY  Double*/
    double mass_density;            /*PRO_MATPROP_MASS_DENSITY           Double*/

    wchar_t* xhatch_file;          /*PRO_MATPROP_XHATCH_FILE           WString*/
    double tensile_yield_stress;    /*PRO_MATPROP_TENSILE_YIELD_STRESS  Double*/
}MatrlPropData;

/*----- Function Prototypes -----*/

ProError UsrMaterialItemIntPropertySet (ProMaterialItem* mat_item,
    ProMaterialPropertyType type,
    int i_val);

ProError UsrMaterialItemDoublePropertySet (ProMaterialItem* mat_item,
    ProMaterialPropertyType type,
    double d_val);

```

```

ProError UstrMaterialItemStringPropertySet (ProMaterialItem* mat_item,
    ProMaterialPropertyType type,
    wchar_t* s_val);

ProError UstrMaterialItemBoolPropertySet (ProMaterialItem* mat_item,
    ProMaterialPropertyType type,
    short l_val);

ProError UstrNonlinMaterialCreate (ProMdl Mdl, MatrlPropData data,
    ProName name);

/*=====*\
FUNCTION : UstrNonlinMaterialCreateWrapper
PURPOSE  : To set the new material properties.
\*=====*/
int UstrNonlinMaterialCreateWrapper()
{
    ProMdl mdl;
    MatrlPropData data;
    ProMaterialItem mat_item;
    ProName name;
    int status;

/*-----*\
    Set the new material name
\*-----*/
    ProStringToWstring (name, "matl_poly2");

    status = ProMdlCurrentGet (&mdl);
    if(status != PRO_TK_NO_ERROR)
        return 0;

/*-----*\
    Set the new material properties
\*-----*/
    data.type = PRO_MATERIAL_TYPE_STRUCTURAL_ISOTROPIC |
        PRO_MATERIAL_TYPE_THERMAL_ISOTROPIC;
    data.material_description = L"taken from spec";

    data.failure_criterion_type = PRO_MATERIAL_FAILURE_DISTORTION_ENERGY;
    data.fatigue_type = PRO_MATERIAL_FATIGUE_TYPE_NONE;
    data.sub_type = PRO_MATERIAL_SUB_TYPE_HYPERELASTIC;

    data.thermal_expansion_coefficient = 4.444000e-05;
    data.specific_heat = 1.444000e+06;
    data.thermal_conductivity = 1.182589e+01;
    data.mass_density = 4.335775e-02;
    data.xhatch_file = NULL;

    data.model_def_by_tests = PRO_B_FALSE;

```



```

data.model = PRO_MATERIAL_MODEL_POLYNOMIAL;

data.model_coeff_c10 = 8.809880e-01;
data.model_coeff_c20 = 1.116500e-02;
data.model_coeff_c01 = 3.780000e-04;
data.model_coeff_c02 = 1.000000e+00;
data.model_coeff_c11 = -1.620000e-04;

data.model_coeff_d1 = 0.00;
data.model_coeff_d2 = 0.00;

data.tensile_yield_stress = 2.519884e+06;

/*-----*\
  Call the material create function
\*-----*/
  UsrNonlinMaterialCreate(mdl, data, name);

  return 0;
}

/*=====*\
FUNCTION : UsrNonlinMaterialCreate()
PURPOSE  : To create the new non linear material.
\*=====*/
ProError UsrNonlinMaterialCreate(ProMdl Mdl, MatrlPropData data,
  ProName name)
{

  ProMaterialItem mat_item;
  ProMaterial material;

  ProMaterialCreate (Mdl, name,
    NULL, &material);

  ProModelitemByNameInit (Mdl, PRO_RP_MATERIAL, name,
    &mat_item);

/*-----*\
  Set material properties
\*-----*/

  UsrMaterialItemIntPropertySet (&mat_item,
    PRO_MATPROP_TYPE,
    data.type);

  UsrMaterialItemStringPropertySet (&mat_item,
    PRO_MATPROP_MATERIAL_DESCRIPTION,
    data.material_description);

```

```

UsrMaterialItemStringPropertySet (&mat_item,
    PRO_MATPROP_FAILURE_CRITERION_TYPE,
    data.failure_criterion_type);

UsrMaterialItemStringPropertySet (&mat_item,
    PRO_MATPROP_FATIGUE_TYPE,
    data.fatigue_type);

UsrMaterialItemStringPropertySet (&mat_item,
    PRO_MATPROP_SUB_TYPE,
    data.sub_type);

UsrMaterialItemDoublePropertySet (&mat_item,
    PRO_MATPROP_THERMAL_EXPANSION_COEFFICIENT,
    data.thermal_expansion_coefficient);

UsrMaterialItemDoublePropertySet (&mat_item,
    PRO_MATPROP_SPECIFIC_HEAT,
    data.specific_heat);

UsrMaterialItemDoublePropertySet (&mat_item,
    PRO_MATPROP_THERMAL_CONDUCTIVITY,
    data.thermal_conductivity);

UsrMaterialItemDoublePropertySet (&mat_item,
    PRO_MATPROP_MASS_DENSITY,
    data.mass_density);

UsrMaterialItemStringPropertySet (&mat_item,
    PRO_MATPROP_XHATCH_FILE,
    data.xhatch_file);

UsrMaterialItemBoolPropertySet (&mat_item,
    PRO_MATPROP_MODEL_DEF_BY_TESTS,
    data.model_def_by_tests);

UsrMaterialItemStringPropertySet (&mat_item,
    PRO_MATPROP_MODEL,
    data.model);

UsrMaterialItemDoublePropertySet (&mat_item,
    PRO_MATPROP_MODEL_COEF_C10,
    data.model_coeff_c10);

UsrMaterialItemDoublePropertySet (&mat_item,
    PRO_MATPROP_MODEL_COEF_C01,
    data.model_coeff_c01);

UsrMaterialItemDoublePropertySet (&mat_item,
    PRO_MATPROP_MODEL_COEF_C20,
    data.model_coeff_c20);

```

```

    UsrMaterialItemDoublePropertySet (&mat_item,
        PRO_MATPROP_MODEL_COEF_C02,
        data.model_coeff_c02);

    UsrMaterialItemDoublePropertySet (&mat_item,
        PRO_MATPROP_MODEL_COEF_C11,
        data.model_coeff_c11);

    UsrMaterialItemDoublePropertySet (&mat_item,
        PRO_MATPROP_MODEL_COEF_D1,
        data.model_coeff_d1);

    UsrMaterialItemDoublePropertySet (&mat_item,
        PRO_MATPROP_MODEL_COEF_D2,
        data.model_coeff_d2);

    UsrMaterialItemDoublePropertySet (&mat_item,
        PRO_MATPROP_TENSILE_YIELD_STRESS,
        data.tensile_yield_stress);

    return PRO_TK_NO_ERROR;
}

/*=====*\
FUNCTION : UsrMaterialItemBoolPropertySet()
PURPOSE  : To set material property of type ProBool.
\*=====*/

ProError UsrMaterialItemBoolPropertySet (ProMaterialItem* mat_item,
    ProMaterialPropertyType type,
    short l_val)
{
    ProParamvalue value;

    value.type = PRO_PARAM_BOOLEAN;
    value.value.l_val = l_val;

    ProMaterialPropertySet (mat_item, type, &value, NULL);

    return PRO_TK_NO_ERROR;
}

/*=====*\
FUNCTION : UsrMaterialItemIntPropertySet()
PURPOSE  : To set material property of type integer.
\*=====*/

ProError UsrMaterialItemIntPropertySet (ProMaterialItem* mat_item,
    ProMaterialPropertyType type,

```

```

        int i_val)
{
    ProParamvalue value;

    value.type = PRO_PARAM_INTEGER;
    value.value.i_val = i_val;

    ProMaterialPropertySet (mat_item, type, &value, NULL);

    return PRO_TK_NO_ERROR;
}

/*=====*\
FUNCTION : UsrMaterialItemDoublePropertySet()
PURPOSE  : To set material property of type double.
\*=====*/

ProError UsrMaterialItemDoublePropertySet (ProMaterialItem* mat_item,
    ProMaterialPropertyType type,
    double d_val)
{
    ProParamvalue value;
    ProUnititem units;
    ProParamvalue old_val;
    int err;

    value.type = PRO_PARAM_DOUBLE;
    value.value.d_val = d_val;

    err = ProMaterialPropertyGet (mat_item, type, &old_val, &units);

    if (err == PRO_TK_E_NOT_FOUND)
        ProMaterialPropertySet (mat_item, type, &value, NULL);
    else
        ProMaterialPropertySet (mat_item, type, &value, &units);

    return PRO_TK_NO_ERROR;
}

/*=====*\
FUNCTION : UsrMaterialItemStringPropertySet()
PURPOSE  : To set material property of type string
\*=====*/

ProError UsrMaterialItemStringPropertySet (ProMaterialItem* mat_item,
    ProMaterialPropertyType type,
    wchar_t* s_val)
{
    ProParamvalue value;

```

```
value.type = PRO_PARAM_STRING;  
ProWstringCopy (s_val, value.value.s_val, PRO_VALUE_UNUSED);  
  
ProMaterialPropertySet (mat_item, type, &value, NULL);  
  
return PRO_TK_NO_ERROR;  
}
```


4

Core: Features

This chapter describes the Pro/TOOLKIT functions that deal with features as a whole and the way they relate to each other.

Access to the geometry objects created by features is described in the Core: 3D Geometry chapter.

Access to the internal structure of a feature is described in the Element Trees: Principles of Feature Creation chapter.

Topic	Page
Feature Objects	4 - 2
Visiting Features	4 - 2
Feature Inquiry	4 - 3
Feature Geometry	4 - 6
Manipulating Features	4 - 7
Manipulating Features based on Regeneration Flags	4 - 12
Feature Dimensions	4 - 13
Manipulating Patterns	4 - 13
Creating Local Groups	4 - 19
Read Access to Groups	4 - 20
Updating or Replacing UDFs	4 - 23
Placing UDFs	4 - 24
The UDF Input Data Structure—ProUdfdata	4 - 26
Reading UDF Properties	4 - 34
Notification on UDF Library Creation	4 - 37

Feature Objects

Function Introduced:

- **ProFeatureInit()**

Features are represented by the object *ProFeature*, which is declared as a DHandle, or data handle. It shares the same declaration as *ProModelitem* and *ProGeomitem*, and therefore contains the type and integer identifier as fields in the structure.

Like *ProGeomitem*, *ProFeature* is an instance of *ProModelitem*. *ProFeature* objects are contained in *ProSolid* objects, and contain *ProGeomitem* objects.

You can create a new *ProFeature* handle using the function **ProFeatureInit()**.

Visiting Features

Function Introduced:

- **ProSolidFeatVisit()**

The function **ProSolidFeatVisit()** enables you to visit all the features in a part or assembly. It visits not only those features visible to the user, but also features used internally for construction purposes. To skip over such internal, “invisible” functions, call **ProFeatureVisibilityGet()**.

Note that the function **ProSolidFeatstatusGet()** (described in detail in the “Core: Solids, Parts, and Materials” chapter) provides an array of integer identifiers for all the features in a solid, thereby offering an alternate way of finding all the features.

Feature Inquiry

Functions Introduced:

- **ProFeatureTypeGet()**
- **ProFeatureSubtypeGet()**
- **ProFeatureTypenameGet()**
- **ProFeatureStatusGet()**
- **ProFeatureStatusflagsGet()**
- **ProFeatureIsIncomplete()**
- **ProFeatureIsNcseq()**
- **ProFeatureSolidGet()**
- **ProFeatureChildrenGet()**
- **ProFeatureParentsGet()**
- **ProFeatureSelectionGet()**
- **ProFeatureHasGeomchks()**
- **ProFeatureIsReadOnly()**
- **ProFeatureIsEmbedded()**
- **ProInsertModelsActive()**
- **ProFeatureCopyinfoGet()**

As described earlier, the function **ProSolidFeatVisit()** finds all the features belonging to a part or an assembly. The feature inquiry functions provide more information about a particular feature.

The function **ProFeatureTypeGet()** provides the type of the feature. This feature type uses the data type *ProFeattype*, which is really an integer that takes defined values such as the following:

- PRO_FEAT_FIRST_FEAT
- PRO_FEAT_HOLE
- PRO_FEAT_SHAFT
- PRO_FEAT_ROUND

See the include file *ProFeatType.h* for the list of defined values.

The function **ProFeatureTypeGetNameGet()** returns the name of the feature type. Given a ProFeature pointer to a specific feature, this function returns the name of the feature type, for example, CHAMFER, DATUM, COORDINATE SYSTEM, and so on. Arguments to this function must not be null.

The function **ProFeatureSubtypeGet()** provides the subtype (such as sheet metal) of a specified feature. Note that not all features support subtypes. This is like viewing valid model subtypes by opening the Pro/ENGINEER **Model Tree Setup>Column Display>View** menu and then selecting Feat Subtype as an additional display column.

The function **ProFeatureStatusGet()** classifies the feature according to the following status values:

- PRO_FEAT_ACTIVE—An ordinary feature.
- PRO_FEAT_SUPPRESSED—A suppressed feature.
- PRO_FEAT_FAMTAB_SUPPRESSED—A feature suppressed due to the family table settings.
- PRO_FEAT_SIMP_REP_SUPPRESSED—A feature suppressed due to the simplified representation.
- PRO_FEAT_PROG_SUPPRESSED—A feature suppressed due to Pro/PROGRAM.
- PRO_FEAT_INACTIVE—A feature that is not suppressed, but is not currently in use for reasons other than the ones identified above.
- PRO_FEAT_UNREGENERATED—A feature that has not yet been regenerated. This is due to a regeneration failure or if the status is obtained during the regeneration process.
- PRO_FEAT_INVALID—The feature status could not be retrieved.

The function **ProFeatureStatusflagsGet()** retrieves the bitmask containing one or more of the following feature status bit flags for a specified feature:

- PRO_FEAT_STAT_INVALID—Specifies an invalid feature.
- PRO_FEAT_STAT_INACTIVE—Specifies an inactive feature. If the bit flag is set to 0, then it means an active feature.
- PRO_FEAT_STAT_ACTIVE—Specifies an active feature.
- PRO_FEAT_STAT_FAMTAB_SUPPRESSED—Specifies a feature suppressed due to the family table settings.

- **PRO_FEAT_STAT_SIMP_REP_SUPPRESSED**—Specifies a feature suppressed due to the simplified representation.
- **PRO_FEAT_STAT_PROG_SUPPRESSED**—Specifies a feature suppressed due to Pro/PROGRAM.
- **PRO_FEAT_STAT_SUPPRESSED**—Specifies a suppressed feature.
- **PRO_FEAT_STAT_UNREGENERATED**—Specifies an active feature that has not yet been regenerated. This is due to a regeneration failure or if the status is obtained during the regeneration process.
- **PRO_FEAT_STAT_FAILED**—Specifies a failed feature.
- **PRO_FEAT_STAT_CHILD_OF_FAILED**—Specifies a child of a failed feature.
- **PRO_FEAT_STAT_CHILD_OF_EXT_FAILED**—Specifies a child of an external failed feature.

The function **ProFeatureIsIncomplete()** tells you whether a specified feature is incomplete. An incomplete feature is one that has been created by using **ProFeatureCreate()** from a Pro/TOOLKIT application, but which does not yet contain all the necessary feature elements to allow regeneration.

The function **ProFeatureIsNcseq()** determines whether a feature is an NC sequence.

The **ProFeatureSolidGet()** function provides the identifier of the solid that owns the specified feature.

The **ProFeatureChildrenGet()** and **ProFeatureParentsGet()** functions get the children and parents of the specified feature. For these functions, the parent of a feature means a feature it *directly* depends on, and a child is a feature that *directly* depends on it. This differs from the Pro/ENGINEER command **Info**, **Feat Info**, which also shows indirect dependencies.

The function **ProFeatureSelectionGet()** is used for features that were created in a part as a result of a feature in a parent assembly. For example, if you create a hole in Assembly mode, then select a part to be intersected by that hole, the geometry of the hole is visible to Pro/TOOLKIT as belonging to the part, even if the original feature is specified as being visible at the assembly level. This geometry—a list of the surfaces forming the hole—belongs to a feature in the part whose type is **PRO_FEAT_ASSEM_CUT**. The function **ProFeatureSelectionGet()**, when applied to that part feature, identifies the assembly, and the path down through it to the part in question, which contains the original feature.

During regeneration, Pro/ENGINEER performs geometry checking to prevent regeneration errors. The geometry check process identifies features that could cause problems if the part or assembly is modified, but which do not cause regeneration failure in the model in its present state. The **ProFeatureHasGeomchks()** function outputs a variable of type *ProBoolean* that indicates whether a particular feature, identified as an input argument to the function, has geometry checks.

The function **ProFeatureIsReadonly()** provides information about the read status of the specified feature. Its first argument is a pointer to the feature's (*ProFeature*) handle. If the feature is read only, the function outputs a *ProBoolean* with the value PRO_B_TRUE; otherwise, the value is PRO_B_FALSE.

The feature **ProFeatureIsEmbedded()** identifies whether the feature is an embedded datum. Embedded features are visible in the model tree, but cannot be used as reference parents for features other than the feature into which they are embedded.

To determine whether insert mode is active in a specified solid, use the function **ProInsertModeIsActive()**. If activated, features are inserted into the feature list after the feature specified when **ProFeatureInsertModeActivate()** was called. New features continue to be inserted until you call the function **ProInsertModeCancel()**. See the section Manipulating Features for more information about insert mode.

The function **ProFeatureCopyinfoGet()** returns information about a copied feature. The information includes the type of copy operation, dependency, source feature, and additional features copied in the same operation. This function supersedes the Pro_copy_info structure returned by the Pro/Develop function **prodb_feature_info()**.

Feature Geometry

Functions Introduced:

- **ProFeatureGeomitemVisit()**
- **ProGeomitemFeatureGet()**

For information about feature geometry, see the Core: 3D Geometry chapter.

Manipulating Features

Functions Introduced:

- **ProFeatureDelete()**
- **ProFeatureSuppress()**
- **ProFeatureResume()**
- **ProFeatureRedefine()**
- **ProFeatureInsertModeActivate()**
- **ProInsertModeCancel()**
- **ProFeatureReorder()**
- **ProFeatureNumberGet()**
- **ProFeatureReadonlySet()**
- **ProFeatureReadonlyUnset()**

Core: Features

The functions **ProFeatureDelete()** and **ProFeatureSuppress()** act like the Pro/ENGINEER commands **RMB>Delete** and **RMB>Suppress**, except they do not repaint the window. You can process many features in a single call using an input of type *ProFeatlist*. Each of these functions takes an array of options as the input that indicate whether to also delete or suppress features dependent on those being acted on directly. The options used while deleting or suppressing features are as follows:

- **PRO_FEAT_DELETE_NO_OPTS**—Delete or suppress the features without deleting or suppressing their dependent children features. This may result in regeneration failures. Use the option **PRO_FEAT_DELETE_FIX**, or one of the **CLIP** options to fix these failures.
- **PRO_FEAT_DELETE_CLIP**—Delete or suppress the features along with their dependent children features.
- **PRO_FEAT_DELETE_FIX**—Delete or suppress the features without deleting or suppressing their dependent children features. The fix model user interface will be prompted in case of a regeneration failure. This option must be used only in the Resolve mode. Otherwise, the function returns **PRO_TK_BAD_CONTEXT**.
- **PRO_FEAT_DELETE_RELATION_DELETE**—Delete relations with obsolete dimensions.
- **PRO_FEAT_DELETE_RELATION_COMMENT**—Change relations with obsolete dimensions into comments.

- **PRO_FEAT_DELETE_CLIP_ALL**—Delete or suppress the features along with all the following features.
- **PRO_FEAT_DELETE_INDIV_GP_MEMBERS**—Individually delete or suppress the features out of the groups to which they belong. If this option is not included, the entire group of features is deleted or suppressed. This option can be included only if the option **PRO_FEAT_DELETE_CLIP** is also included.
- **PRO_FEAT_DELETE_CLIP_INDIV_GP_MEMBERS**—Individually delete or suppress the children of features out of the group to which they belong. If this option is not included, the entire group containing the features and their children is deleted or suppressed. This option can be included only if **PRO_FEAT_DELETE_INDIV_GP_MEMBERS** is also included.
- **PRO_FEAT_DELETE_KEEP_EMBED_DATUMS**—Retain the embedded datums stored in a feature while deleting the feature using **ProFeatureDelete()**. If this option is not included, the embedded datums will be deleted along with the parent feature.

The function **ProFeatureRedefine()** is equivalent to the Pro/ENGINEER command **Feature>Redefine**. Additionally, it can redefine an existing feature with the new element tree. The data passed in through the new element tree replaces the existing data in the feature.

Pro/TOOLKIT provides access to the Pro/ENGINEER feature insert mode functionality with the **ProFeatureInsertModeActivate()** and **ProInsertModeCancel()** functions. The function **ProFeatureInsertModeActivate()** takes a single argument—the handle to the feature after which new features are to be inserted. This feature becomes the last feature in the feature regeneration list. All features that had appeared after that feature are temporarily suppressed. New features are added after the (new) last feature. Feature insertion continues until insert mode is terminated with a call to **ProInsertModeCancel()**. Its first argument is a handle to the solid, and the second is a *ProBoolean* that enables you to specify whether suppressed features are to be resumed.

Example 1: Inserting a Feature

The following example code shows how to use insert mode.

```
int InsertModeTest()
{
    static wchar_t  msgfil[80];
    ProMdl          solid;
    ProBoolean      insert_active;
```

```

int          status, n_sel;
ProSelection *sel_feat_array;
ProModelitem f_modelitem;

ProStringToWstring (msgfil, "insert_mode.txt");
/*-----*\
Get the feature handle for the new "last" feature.
\*-----*/
status = ProMessageDisplay (msgfil, "USER Select feature to be
    new 'last' feature");
status = ProSelect("feature", 1, NULL, NULL, NULL, NULL,
    &sel_feat_array, &n_sel);
/*-----*\
Activate insert mode.
\*-----*/
status = ProSelectionModelitemGet (sel_feat_array[0], &f_modelitem);

status = ProFeatureInsertModeActivate (&f_modelitem);
/*-----*\
Verify that insert mode is active.
\*-----*/
status = ProMdlCurrentGet (&solid);

status = ProInsertModeIsActive (solid, &insert_active);

if (insert_active == PRO_B_TRUE)
    ProMessageDisplay (msgfil, "USER Feature Insert Mode is active");
else
    ProMessageDisplay (msgfil, "USER Feature Insert Mode is NOT
        active");
/*-----*\
Add the axis feature after the "last" feature
\*-----*/
status = UserAxisRefs();
/*-----*\
Deactivate insert mode.
\*-----*/
status = ProInsertModeCancel (solid, PRO_B_TRUE);
return (status);
}

```

The function **ProFeatureReadonlySet()** assigns a read-only status to model features. Its only argument is a *ProFeature* handle that specifies the last feature in the feature list to be designated as read only. All preceding features are read only; all features following this feature have standard access. The function **ProFeatureReadonlyUnset()** removes the read-only status from all features in the specified solid.

The function **ProFeatureReorder()** enables you to change the position of one or more features in the feature regeneration sequence. Its input arguments are as follows:

- *ProSolid solid*—The handle to the solid owner of the features.
- *int *feat_ids*—An array of feature identifiers that specifies the features to be reordered. The array should contain features that formed a contiguous sublist within the original feature regeneration list. If reordering a group, all the features in the group including the Group Header feature must be included in this array.
- *int n_feats*—The number of features to reorder.
- *int new_feat_num*—An integer that indicates the feature before which the reordered features are to appear. This integer is not the feature identifier, but rather the regeneration sequence number of the feature. You obtain this number by calling the function **ProFeatureNumberGet()**.

Use the function **ProSolidFeatstatusGet()** to get the current sequence and statuses. You must use care when you change the sequence of features. Unless you have advance knowledge of the relationship between the features you are reordering, you should use the functions **ProFeatureParentsGet()** and **ProFeatureChildrenGet()** before changing the feature order to ensure that no feature is reordered to be before its parent features.

Example 2: Changing the Regeneration Sequence of a Solid

This example shows how to reorder features.

```
int FeatureReorderTest()
{
    static wchar_t    msgfil[80];
    ProMdl            model;
    ProModelitem      modelitem;
    int               status, n_feats_sel, sel_feat_ids[5], n_feat,
                    feat_num, i;
    ProSelection      *sel_features, *reorder_before_feat;
    ProFeature        first_feat;

    ProStringToWstring (msgfil, "feature.txt");
    /*-----*\
    Prompt the user to select the features to be reordered.
    \*-----*/
    status = ProMessageDisplay (msgfil, "USER Select feature(s) to
        reorder (max # of selections = 5)");

    status = ProSelect ("feature", 5, NULL, NULL, NULL, NULL,
```



```

        &sel_features, &n_feats_sel);
if (status != PRO_TK_NO_ERROR)
{
    ProMessageDisplay (msgfil, "USER No selections made; reorder
        terminated");
    return (0);
}
/*-----*\
    Create the array of feature identifiers for the selected
    features.
\*-----*/
for (i = 0; i < n_feats_sel; i++)
{
    status = ProSelectionModelitemGet (sel_features[i], &modelitem);
    sel_feat_ids[i] = modelitem.id;
}
/*-----*\
    Prompt for the feature before which the reordered features will
    be regenerated.
\*-----*/
status = ProMessageDisplay (msgfil, "USER Select feature before
    which to reorder");
status = ProSelect ("feature", 1, NULL, NULL, NULL, NULL,
    &reorder_before_feat, &n_feat);
if (status != PRO_TK_NO_ERROR)
{
    ProMessageDisplay (msgfil, "USER No selection made; reorder
        terminated");
    return (0);
}
/*-----*\
    Get the feature number (in the regeneration sequence) to be
    assigned to the first feature in the reorder feature array.
\*-----*/
status = ProSelectionModelitemGet (*reorder_before_feat,
    &modelitem);

status = ProFeatureNumberGet (&modelitem, &feat_num);
/*-----*\
    Reorder the features.
\*-----*/
status = ProMdlCurrentGet (&model) != PRO_TK_NO_ERROR;
status = ProFeatureReorder ((ProSolid)model, sel_feat_ids,
    n_feats_sel, feat_num);
return (0);
}

```

Manipulating Features based on Regeneration Flags

From Pro/ENGINEER Wildfire 5.0 onward, the following functions described in the earlier section have been deprecated:

- **ProFeatureCreate()**
- **ProFeatureDelete()**
- **ProFeatureSuppress()**
- **ProFeatureResume()**
- **ProFeatureRedefine()**
- **ProFeatureReorder()**
- **ProFeatureInsertModeActivate()**
- **ProInsertModeCancel()**

Instead, you can use the functions described below with the input argument *flags* set to PRO_REGEN_NO_FLAGS for the behavior that is equivalent of the one provided by the deprecated functions.

Functions Introduced:

- **ProFeatureWithoptionsCreate()**
- **ProFeatureWithoptionsDelete()**
- **ProFeatureWithoptionsSuppress()**
- **ProFeatureWithoptionsResume()**
- **ProFeatureWithoptionsRedefine()**
- **ProFeatureWithoptionsReorder()**
- **ProFeatureInsertmodeWithoptionsActivate()**
- **ProInsertmodeWithoptionsCancel()**

The above functions enable you to create, delete, or manipulate the specified features in a solid, based on the bitmask (given by the input argument *flags*) containing one or more regeneration control bit flags of the type PRO_REGEN_* defined in *ProSolid.h*. Refer to the “Regenerating a Solid” section in the “Core: Solids, Parts, and Materials” chapter for more information on the bit flags.

Feature Dimensions

Function Introduced:

- **ProFeatureDimensionVisit()**

The function **ProFeatureDimensionVisit()** visits all the dimensions which belong to the feature.

For more information about dimensions, refer to section Dimensions from the chapter Annotations: Annotation Features and Annotations.

Manipulating Patterns

In the Pro/ENGINEER Wildfire release, the following changes are implemented in patterns.

Patterns as Features

Patterns are treated as features in Pro/ENGINEER Wildfire. Patterns are assigned a header feature of the type PRO_E_PATTERN_HEAD.

The Pattern feature in Pro/ENGINEER Wildfire effects the previous releases of Pro/ENGINEER as follows:

- Models containing patterns automatically get one extra feature of type PRO_FEAT_PATTERN_HEAD in the regeneration list. This changes the feature numbers of all the subsequent features, including those in the pattern.
- The pattern access functions such as **proptn_get_pattern** and **ProPatternLeaderGet** are unaffected by the addition of the pattern header feature. The pattern leader is still the first geometric feature contained in the pattern.

The new function **ProPatternHeaderGet()** returns the header feature.

Fill Patterns

Pro/ENGINEER Wildfire uses a new Fill type of pattern. The function **proptn_get_pattern** allows a new pattern type PRO_PTYPE_FILL for fill patterns. It includes one direction containing all pattern members.

Functions Introduced:

- **ProFeaturePatternStatusGet()**
- **ProFeatureGrppatternStatusGet()**
- **ProFeaturePatternGet()**
- **ProPatternDelete()**
- **ProPatternLeaderGet()**
- **ProPatternHeaderGet()**
- **proptn_get_pattern()**

The function **ProFeaturePatternStatusGet()** classifies the feature according to its possible role in a feature pattern. The possible values are as follows:

- **PRO_PATTERN_NONE**—The feature is not in a pattern.
- **PRO_PATTERN_LEADER**—The feature is the leader of a pattern.
- **PRO_PATTERN_MEMBER**—The feature is a member of a pattern.

The function **ProFeatureGrppatternStatusGet()** does the equivalent for group patterns. The possible values are as follows:

- **PRO_GRP_PATTERN_NONE**—The feature is not in a group pattern.
- **PRO_GRP_PATTERN_LEADER**—The feature is in a group that is the leader of a group pattern.
- **PRO_GRP_PATTERN_MEMBER**—The feature is in a group that is a group pattern member.

The function **ProFeaturePatternGet()** obtains the *ProPattern* handle for the pattern containing the specified feature. (The *ProPattern* handle is described in detail in the chapter Element Trees: Patterns.)

To delete a pattern, pass the corresponding *ProPattern* handle to the function **ProPatternDelete()**.

To obtain the leader feature for a given pattern, pass a *ProPattern* object to the function **ProPatternLeaderGet()**.

To obtain the header feature for a given pattern, pass a *ProPattern* object to the function **ProPatternHeaderGet()**.

To access pattern information use the pattern element tree described in the chapter Element Trees: Patterns. You can access element tree information using the functions **ProElement***(), described in the chapter Element Trees: Principles of Feature Creation.

The following sections describe the data structures *Pro_Pattern*, *Pro_pattern_dir*, and *Pro_pattern_dim*.

Pro_Pattern Structure

The *Pro_Pattern* structure is defined as follows:

```
typedef struct pro_pattern
{
    Pro_pattern_type      type;
    Pro_pattern_option    option;
    int                   ref_pattern_leader;
    Pro_pattern_dir       dir1;
    Pro_pattern_dir       dir2;
    int                   **member_ids;
} Pro_Pattern;

typedef enum pro_pattern_type
{
    PRO_PTYPE_DIMENSION,
    PRO_PTYPE_REFERENCE,
    PRO_PTYPE_TABLE,
    PRO_PTYPE_FILL
} Pro_pattern_type;

typedef enum pro_pattern_option
{
    PRO_POPT_VARYING,
    PRO_POPT_GENERAL,
    PRO_POPT_IDENTICAL
} Pro_pattern_option;
```

The data structure fields are as follows:

- *type*—The pattern type.
- *option*—The pattern option.
- *ref_pattern_leader*—For a reference pattern, this is the feature identifier of the leader of the referenced pattern.
- *dir1, dir2*—The two directions of the pattern. If the pattern is only one-dimensional, *dir2.n_members* are 0, not 1. Table-driven (pattern type PRO_PTYPE_TABLE) and fill patterns (pattern type PRO_PTYPE_FILL) are always one-dimensional.
- *member_ids*—A two-dimensional array of the identifiers of the features in the pattern. The leader of the pattern is *member_ids[0][0]*. The first direction is indexed by the second array index, so member *n* of a one-dimensional pattern is *member_ids[0][n]*. This is true for all pattern types.

Pro_pattern_dir Structure

The *Pro_pattern_dir* structure is defined as follows:

```
typedef struct pro_pattern_dir
{
    int                n_members;
    int                pattern_par;
    int                n_dims;
    Pro_pattern_dim    *pattern_dims;
} Pro_pattern_dir;
```

The data structure fields are as follows:

- *n_members*—The number of members in this direction, including the leading member. If this is 1 or 0, the other fields are not set.
- *pattern_par*—The identifier of the pattern parameter that defines the number of members in this direction. This is valid for dimension-driven patterns only (pattern type PRO_PTYPE_DIMENSION).
- *n_dims*—The number of entries in the *pattern_dims* array.

- *pattern_dims*—An array that contains the identifiers of the dimensions that vary between members of the pattern.

Pro_pattern_dim Structure

The *Pro_pattern_dim* structure is defined as follows:

```
typedef struct pro_pattern_dim
{
    int    original_dim_id;
    int    increment_dim_id;
    int    absolute_dim_id;
} Pro_pattern_dim;
```

The *Pro_pattern_dim* fields are as follows:

- *original_dim_id*—The identifier of the original dimension—that is, the dimension of the leading feature modified in the members. This is valid for dimension-driven patterns only.
- *increment_dim_id*—The identifier of the dimension that contains the increment in the original dimension applied to each successive pattern member. This is valid for dimension-driven patterns only.
- *absolute_dim_id*—For a table-driven pattern, this is the identifier of an absolute dimension corresponding to an entry in the pattern table.

Table-Driven Patterns

Functions Introduced:

- `proptntbl_get_lead_pat_dims()`
- `proptntbl_get_all_tbl_names()`
- `proptntbl_get_active_table()`
- `proptntbl_get_inst_indices()`
- `proptntbl_get_inst_dim_value()`
- `proptntbl_add_inst_to_table()`
- `proptntbl_remove_instance()`
- `proptntbl_set_inst_dim_value()`
- `proptntbl_set_dimval_driven()`
- `proptntbl_set_active_table()`
- `proptntbl_rename_table()`
- `proptntbl_delete_table()`

These functions enable you to read and manipulate the tables for a table-driven pattern. The pattern can be an ordinary feature pattern or a group pattern.

Each table-driven pattern can have more than one table associated with it, but only one is active at any time. The active table is the one whose values are used in regenerating the geometry. All the tables for a given pattern share the same column names. The column names are the names of the dimensions in the pattern leader that are modified in the other pattern members. The tables can have quite different lists of pattern members. The tables are identified in Pro/TOOLKIT by the names that are seen in Pro/ENGINEER, and are represented as wide strings.

The input to the pattern table functions that identifies the pattern is the integer identifier of the leading feature in the pattern. You can find the leader of the pattern from one of the members using the function **ProPatternLeaderGet()**.

A given feature can be the leader of a feature pattern, and at the same time can be the first member in a group that is the leader of a group pattern. In this case, the feature is used to identify both patterns. Therefore, each of the pattern table functions has an input argument *group_flag*, which you set to 1 for a group pattern, and 0 for a feature pattern.

The function **proptntbl_get_all_tbl_names()** provides an array that contains the names of the tables associated with the specified pattern.

The functions **proptntbl_get_active_table()** and **proptntbl_set_active_table()** get and set the name of the currently active table. If you set a different table, use **pro_regenerate_object()** to modify the model geometry to reflect the change.

The function **proptntbl_get_lead_pat_dims()** provides an array of integer identifiers of the dimensions that form the column headers in the pattern tables. The column headers are the dimensions on the pattern leader that are modified to form the other pattern members. All the tables for a given pattern share the same columns, so this function does not need a table name as input.

The function **proptntbl_get_inst_indices()** provides an array of pattern member indices—in effect, these are the row names of the pattern. The pattern member indices can be different for each table, so the function requires a table name as input.

The function **proptntbl_get_inst_dim_value()** provides the current value of a specified dimension for a specified instance in a specified table. The function returns 1 if the value is actually set by the value for the pattern leader (that is, it appears as an asterisk in the pattern table when viewed in Pro/ENGINEER).

The functions **proptntbl_set_inst_dim_value()** and **proptntbl_set_dimval_driven()** enable you to set the value of a pattern table cell to be the specified value, or to be driven by the value of the leader (that is, to be shown as an asterisk in Pro/ENGINEER).

Creating Local Groups

Function Introduced:

- **ProLocalGroupCreate()**

Local groups offer a way to collect several features together as if they were one feature. This functionality is particularly useful when you are creating patterns.

The function **ProLocalGroupCreate()** groups together features specified by an array of feature identifiers. The output of **ProLocalGroupCreate()** is the object *ProGroup*, which is a typedef of a structure similar to *ProFeature*.

The feature identifiers passed to **ProLocalGroupCreate()** must correspond to features that possess consecutive regeneration numbers. That is, the feature identifiers can have any values, but the corresponding features must occupy a contiguous portion of the regeneration list. (To see the regeneration number of a feature, add the column Feat # to the model tree.)

If there are features whose regeneration numbers lie between those belonging to the features to be grouped, Pro/ENGINEER asks the user whether these unspecified features are to be included in the group. If the user responds with No, the group is not created.

After you create a local group, you may want to refresh the model tree to see the changes. To refresh the model tree, call **ProTreetoolRefresh()**.

Read Access to Groups

Groups in Pro/ENGINEER represent sets of contiguous features that act as a single feature for purposes of some operations. While the individual features can be affected by most operations individually, certain operations apply to the entire group:

- Suppress
- Delete
- Layer operations
- Patterning operations

For more information about local groups, see the *Part Modeling User's Guide*.

User-Defined Features (UDFs) are groups of features that can be stored in a file. When a UDF is placed in a new model, the features created are automatically assigned to a group.

A local group is a set of features that have been explicitly assigned to a group, for purposes of ease of modification or patterning.

Note: All the functions in this section work for both UDFs and local groups.

Each instance of a group is identified in Pro/TOOLKIT as a ProGroup structure. This structure is the same as a ProFeature data handle:

```
typedef struct pro_model_item {  
    ProMdl  owner;
```

```

        int      id;

        ProType type;

    } ProGroup;

```

The integer id in this case is the id of the *group header feature*, which is the first feature in the group. All groups, including those in models created before release 200i², are assigned with a group header feature upon retrieval.

The consequences of the group header feature for users of previous versions of Pro/TOOLKIT is as follows:

- Models that contain groups automatically get one extra feature in the regeneration list, of type PRO_FEAT_GROUP_HEAD. This changes the feature numbers of all subsequent features, including those in the group.
- Each group automatically contains one new feature in the arrays returned by Pro/TOOLKIT.
- Each group automatically gets a different leader feature (the group head feature is the leader). The leader is the first feature in the arrays returned by Pro/TOOLKIT.
- Each group pattern contains, of course, a series of groups, and each group in the pattern is similarly altered.

Finding Groups

Functions Introduced:

- **ProSolidGroupVisit()**
- **ProSolidGroupsCollect()**
- **ProFeatureGroupStatusGet()**
- **ProFeatureGroupGet()**

The function **ProSolidGroupVisit()** allows you to visit the groups in the solid model. The function **ProSolidGroupsCollect()** returns an array of the group structures.

The function **ProFeatureGroupStatusGet()** tells you if the specified feature is in a group.

The function **ProFeatureGroupGet()** returns the ProGroup that includes the feature.

Group Information

Functions Introduced

- **ProUdfNameGet()**
- **ProGroupsTabledriven()**
- **ProGroupFeatureVisit()**
- **ProGroupFeaturesCollect()**
- **ProUdfDimensionVisit()**
- **ProUdfDimensionsCollect()**
- **ProUdfDimensionNameGet()**

The function **ProUdfNameGet()** returns the name of the group. For a local group, this is the name assigned upon creation. For a UDF-created group, this is the name of the UDF file. If the UDF is an instance in a UDF family table, the function also returns the instance name.

The function **ProGroupFeatureVisit()** traverses the members of the feature group. The function **ProGroupFeaturesCollect()** returns an array containing the feature handles.

The function **ProUdfDimensionVisit()** traverses the variable dimensions used in the creation of the UDF (this is only applicable to UDF-created groups). The function **ProUdfDimensionsCollect()** returns an array of the variable dimensions. The variable dimensions are the dimensions that Pro/ENGINEER prompts for when you create the UDF.

The function **ProUdfDimensionNameGet()** returns the original dimension symbol for the variable dimension in the UDF. This symbol is different from the symbol assigned to the dimension in the group.

Creating Groups

Functions Introduced:

- **ProLocalGroupCreate()**

The function **ProLocalGroupCreate()** creates a group out of a set of specified features. The features must represent a contiguous set of features in the solid model. (Refer also to Creating Local Groups).

Deleting Groups

Functions Introduced:

- **ProGroupUngroup()**
- **ProGroupUngroupPreAction()**
- **ProGroupUngroupPostAction()**

The function **ProGroupUngroup()** removes the indicated group and deletes the group header feature.

The function prototype **ProGroupUngroupPreAction()** should be used for a notification corresponding to the *ProNotifyType* PRO_GROUP_UNGROUP_PRE. This callback will be called just before the user ungroups an existing local group or UDF group in the user interface. If the application returns an error from this callback, ungroup activity will be prevented (thus providing a means by which UDFs or other groups may be effectively locked). If the ungroup is being cancelled by the application, the application is required to provide an informational message to the user explaining this action.

The function prototype **ProGroupUngroupPostAction()** should be used for a notification corresponding to the *ProNotifyType* PRO_GROUP_UNGROUP_POST. This prototype provides information about the group that was just ungrouped:

- *solid*—The solid model that owns the group.
- *group_id*—The former group feature id.
- *udf_name*—The name of the UDF the group was created from.
- *feature_list*—The feature ids that were members of the group.

Updating or Replacing UDFs

This section lists operations, which you can perform on UDFs.

Functions Introduced:

- **ProUdfUpdate()**
- **ProUdfReplace()**

The function **ProUdfUpdate()** updates a dependent UDF to the latest version of the .gph file. The function should be able to locate the .gph file from within the session or by the search path. Only dependent UDFs are updated from their original definitions.

Use the function **ProUdfReplace()** to replace a UDF placement with a similar UDF provided that the two UDF's must use the same reference types. The input to the function can include data that would be used to respond to prompts shown during an interactive replacement (for items like scale, dimension display and orientation prompts).

Placing UDFs

Function Introduced:

- **ProUdfCreate()**

The function **ProUdfCreate()** is used to create a new group by retrieving and applying the contents of an existing UDF file. It is equivalent to the Pro/ENGINEER command **Insert>User Defined Feature** available from the top menu bar.

To understand this function explanation, you must have a good knowledge and understanding of the use of UDFs in Pro/ENGINEER. PTC recommends that you read about UDFs in the *Part Modeling User's Guide*, and practice defining and using UDFs in Pro/ENGINEER before you attempt to use this function.

When you create a UDF interactively, Pro/ENGINEER prompts you for the information it needs to fix the properties of the resulting features. When you create a UDF from Pro/TOOLKIT, you can provide some or all of this information programmatically by assembling the data structure that is the input to the function **ProUdfCreate()**.

During the call to **ProUdfCreate()**, Pro/ENGINEER prompts you for the following:

- Any information the UDF needs that you did not provide in the input data structure
- Correct information to replace erroneous information

Such prompts are a useful way of diagnosing errors when you develop your application. This also means that, in addition to creating UDFs fully programmatically to provide automatic synthesis of model geometry, you can also use **ProUdfCreate()** to create UDFs semi-interactively. This can simplify the interactions needed to place a complex UDF, making it easier for the user and less prone to error.

Creating a UDF may require the following types of information:

- **Name**—The name of the UDF library to create, and the instance name, if applicable.
- **Name or path**—the name (or full path) of the UDF to create, and the instance name, if applicable.
- **Dependency**—Whether the UDF is independent of the UDF definition, or is modified by changes made to it.
- **Scale**—How to scale the UDF relative to the placement model.
- **Variable parameters**—The new values of the variable parameters allowed to be changed during UDF placement.
- **Variable annotations**—The new values of the variable gtol values, surface finish values and dimension tolerances allowed to be changed during UDF placement.
- **Variable dimensions**—The new values of the variable dimensions and pattern parameters; those whose values can be modified each time the UDF is created.
- **Dimension display**—Whether to show or blank non-variable dimensions created within the UDF group.
- **References**—The geometrical elements (surfaces, edges, datum planes, and so on) that the UDF needs to relate the features it contains to the existing model features. The elements correspond to the picks that Pro/ENGINEER prompts you for when you create the UDF interactively (using the prompts defined when the UDF was set up). You cannot select an embedded datum as the UDF reference.
- **Part intersections**—If the UDF is being created in an assembly and contains features that modify existing solid geometry, you need to define which parts in the assembly are to be affected (or "intersected"), and at which level in the assembly each such intersection is to be visible.
- **Orientations**—If a UDF contains a feature whose direction is defined with respect to a datum plane (for example, a hole feature that uses a datum plane at its placement plane), Pro/ENGINEER needs to know in which direction the new feature is to point (that is, on which side of the datum plane it should lie). When you create such a UDF interactively, Pro/ENGINEER prompts you for this orientation with a flip arrow.

- **Quadrants**—If a UDF contains a linearly placed feature that references two datum planes to define its location (in the new model), Pro/ENGINEER prompts you to pick the location of the new feature. This decides on which side of each datum plane the feature must lie. This choice is referred to as the "quadrant," because there are four combinations of possibilities for each linearly placed feature.
- **External symbols**—The parameter or dimension to use in place of a missing external symbol from a note callout or relation.
- **Copied model names**—If a UDF creates components in an assembly, this argument specifies the names of the new copied components that the placement creates.

The function **ProUdfCreate()** takes the following arguments:

- *solid*—The solid model (part or assembly) on which to place the UDF.
- *data*—The UDF creation data, described below.
- *asm_reference*—An external reference assembly for calculating intersections and external references.
- *options*—An array of option flags.
- *n_options*—The size of the options array.

The UDF Input Data Structure—ProUdfdata

Most of the input needed by the function **ProUdfCreate()** is contained in the single ProUdfdata structure. This structure can be assembled using the ProUdfdata functions.

The options in the data structure correspond closely to the prompts Pro/ENGINEER gives you when you create a UDF interactively. PTC strongly recommends that before you write the Pro/TOOLKIT code to fill the structure, you experiment with creating the UDF interactively using Pro/ENGINEER, noting what prompts it gives you, and use this as a guide to the information you need to provide.

Functions Introduced:

- **ProUdfdataAlloc()**
- **ProUdfdataFree()**
- **ProUdfdataNameSet()**
- **ProUdfdataPathSet()**
- **ProUdfdataInstancenameSet()**
- **ProUdfdataDependencySet()**
- **ProUdfdataScaleSet()**
- **ProUdfdataDimdisplaySet()**
- **ProUdfdataOrientationAdd()**
- **ProUdfdataQuadrantAdd()**

The function **ProUdfdataAlloc()** allocates memory for the ProUdfdata structure. The function **ProUdfdataFree()** frees the data structure memory.

The function **ProUdfdataNameSet()** allows you to set the name of the UDF (the root of the file name) to create and, optionally, the instance in the UDF family table to use.

Use the function **ProUdfdataPathSet()** to set the path of the UDF file. **ProUdfCreate()** will use the input from this path, if set, otherwise the data from **ProUdfdataNameSet()** is used.

Use function **ProUdfdataInstancenameSet()** to assign the instance to be used when placing this UDF.

The function **ProUdfdataDependencySet()** specified the dependency of the UDF. The choices correspond to the choices available when you create the UDF interactively. The default for this option, if not explicitly specified, is to create the group independent of the UDF definition.

The function **ProUdfdataScaleSet()** specifies how to modify the dimensions of the UDF with respect to the placement model. The choices correspond to the options presented when you create the UDF interactively. A value for a user-defined scale can also be specified by this function. The default for this option, if not explicitly specified, is to use the same size for the UDF, regardless of the units of the placement model.

The function **ProUdfdataDimdisplaySet()** specifies how to present the non-variable dimensions in the created group. These values correspond to the options presented in Pro/ENGINEER when placing the UDF interactively. The default for this option, if not explicitly specified, is to display the dimensions normally (allowing modification).

The function **ProUdfdataOrientationAdd()** adds to an array of orientation choices. These orientation options answer the Pro/ENGINEER prompts that propose a flip arrow (presented, for example, when using datum planes as a reference). There should be one orientation answer presented for each prompt in Pro/ENGINEER, and the order of the options should correspond to the order as presented in Pro/ENGINEER. If an orientation option is not provided, the value “no flip” is applied.

The function **ProUdfdataQuadrantAdd()** adds to an array of 3-dimensional points that correspond to the picks answering the Pro/ENGINEER prompts for the feature positions. The quadrant is requested when placing a hole or a shaft with respect to two datum planes if the UDF references were also datum planes. The order of quadrants specified should correspond to the order in which Pro/ENGINEER prompts for them when the UDF is created interactively.

Variable Parameters and Annotations

The data structure for both variable parameters and annotations is *ProUdfvarparam*.

Functions Introduced:

- **ProUdfvarparamAlloc()**
- **ProUdfdataVarparamAdd()**
- **ProUdfvarparamValueSet()**
- **ProUdfvarparamFree()**

The function **ProUdfvarparamAlloc()** allocates a UDF variable parameter or annotation structure which describes a variable parameter or annotation. The input arguments of this function are:

- *name*—Specifies the parameter name. If a variable annotation, this must be one of the standard annotation parameter names:
 - PTC_GTOL_PRIMARY_TOL—gtol value
 - PTC_ROUGHNESS_HEIGHT—surface finish value

- PTC_DIM_TOL_VALUE—dimension symmetric tolerance value
- PTC_DIM_UPPER_TOL_VALUE—upper dimension tolerance
- PTC_DIM_LOWER_TOL_VALUE—lower dimension tolerance
- *item*—Specifies the owner item. This item must have type and id filled out. (The owner field is ignored by Pro/ENGINEER). The following types are allowed here:
 - PRO_FEATURE
 - PRO_ANNOTATION_ELEM

Use the function **ProUdfdataVarparamAdd()** to add information about a variable parameter assignment to the UDF data.

The function **ProUdfvarparamValueSet()** assigns the value to be used for a variable parameter or annotation value when the UDF is placed. Note: you still must add the variable parameter to the UDF data using **ProUdfdataVarparamAdd()**.

Use the function **ProUdfvarparamFree()** to free the UDF variant parameter handle.

Variable Dimensions and Pattern Parameters

The data structure for variable dimensions and pattern parameters is *ProUdfvardim*.

Functions Introduced:

- **ProUdfvardimAlloc()**
- **ProUdfdataUdfvardimAdd()**
- **ProUdfvardimValueSet()**
- **ProUdfvardimFree()**

The function **ProUdfvardimAlloc()** sets the values used to determine the variant dimension value. This function requires the following inputs:

- *dim_name*—The symbol that the dimension or pattern parameter had when the UDF was originally defined; not the prompt that the UDF uses when interactively created. To make the name easy to remember, modify the symbols of all the dimensions that you want to select to be variable before you define the UDF that you plan to create with Pro/TOOLKIT.

If you do not remember the name, find it by creating the UDF interactively in a test model and then using the Pro/TOOLKIT functions **ProUdfDimensionVisit()** and **ProUdfDimensionNameGet()** on the resulting UDF.

If you get the name wrong, **ProUdfCreate()** does not recognize the dimension and prompts the user for the value.

- *value*—The new value of the dimension or pattern parameter.
- *type*—This enumerated type takes one of the following values:
 - PROUDFVARTYPE_DIM—For a dimension.
 - PROUDFVARTYPE_IPAR—For a pattern parameter.

The function **ProUdfdataUdfvardimAdd()** adds a variant dimension data structure to the UDF creation data.

The function **ProUdfvardimValueSet()** assigns the value to be used for a variable dimension value when the UDF is placed.

Note: The variant dimension must be added to the UDF data structure using **ProUdfdataUdfvardimAdd()** in order for it to be used during placement.

Use the function **ProUdfvardimFree()** to free the UDF variant dimension handle.

UDF References

Functions Introduced:

- **ProUdfreferenceAlloc()**
- **ProUdfdataReferenceAdd()**
- **ProUdfreferenceFree()**

The function **ProUdfreferenceAlloc()** creates a new reference data structure. The data that must be provided to allocate the structure is:

- *prompt*—The prompt defined for this reference when the UDF was originally set up. It indicates which reference this structure is providing.

- *ref_item*—A ProSelection object representing the geometry to use as the reference. You can allocate an embedded datum as the UDF reference. If the reference is external, the selection component path should represent the path to the owning model relative to the external reference assembly specified in the call to **ProUdfCreate()**. If this reference item refers to an annotation reference, you can pass NULL to make the placed annotation incomplete.
- *external*—PRO_B_TRUE if the reference is external, and PRO_B_FALSE if it is internal.
 - Internal—The referenced element belongs directly to the model that contains the UDF. For an assembly, this means that the element belongs to the top level assembly.
 - External—The referenced element belongs to an assembly member other than the placement member.

The function **ProUdfdataReferenceAdd()** adds the reference structure to the ProUdfdata structure.

Use the function **ProUdfreferenceFree()** to free the UDF reference handle.

Assembly Intersections

The data structure used for assembly intersections is *ProUdfintersection*.

Functions Introduced:

- **ProUdfintersectionAlloc()**
- **ProUdfdataIntersectionAdd()**
- **ProUdfintersectionFree()**

The function **ProUdfintersectionAlloc()** sets the values used to determine how a UDF placed in the context of an assembly intersects the members of the assembly. This function requires the following inputs:

- *intersect_part*—The component path from either the placement assembly or the external reference assembly down to the intersected component. The external reference assembly is provided by the asm_reference argument to **ProUdfCreate()**.

- *visibility*—The depth of the component path into the assembly where the intersected UDF is visible. If visibility is equal to the length of the component path, the feature is visible in the part that it intersects and all assemblies and subassemblies. If visibility is 0, the feature is only visible in the top level assembly.
- *instance_names*—An array of names for the new instances of parts created to represent the intersection geometry.

The function **ProUdfdataIntersectionAdd()** adds the intersection structure to the ProUdfdata structure.

Use the function **ProUdfintersectionFree()** to free the UDF intersection handle.

External Symbol: Parameters

The data structure for external symbol parameters is *ProUdfextparam*.

Functions Introduced:

- **ProUdfextparamAlloc()**
- **ProUdfdataExtparamAdd()**
- **ProUdfextparamFree()**

The function **ProUdfextparamAlloc()** allocates and sets a *ProUdfextparam* structure, which describes an external symbol referencing a parameter. The input arguments of this function are:

- *prompt*—The prompt for the external parameter symbol.
- *parameter*—The parameter to used to resolve this external symbol in the placement model.

Use the function **ProUdfdataExtparamAdd()** to add information about an external symbol parameter to the UDF data. Use the function **ProUdfextparamFree()** to free the UDF external parameter handle.

External Symbol: Dimensions

The data structure for external symbol dimensions is *ProUdfextdim*.

Functions Introduced:

- **ProUdfextdimAlloc()**
- **ProUdfdataExtdimAdd()**
- **ProUdfextdimFree()**

Use the function **ProUdfextdimAlloc()** to allocate and set a structure which describes an external dimension symbol required by the UDF. The input arguments of this function are:

- *prompt*—Specifies the prompt used for this external symbol.
- *dimension*—Specifies the dimension handle to be used to resolve the external symbol in the placement model

Use the function **ProUdfdataExtdimAdd()** to add information about a required external dimension symbol to the UDF data. Use the function **ProUdfextdimFree()** to free the UDF dimension external symbol handle.

Copied Model Names

The data structure used for specifying new component model names is *ProUdfmdlNames*.

Functions Introduced:

- **ProUdfmdlNamesAlloc()**
- **ProUdfmdlNamesSet()**

The function **ProUdfmdlNamesAlloc()** sets the values used to determine the names of new components created by the UDF placement. This function requires the following inputs:

- *old_name*—The old name of the component.
- *new_name*—The new name of the component to be created.

The function **ProUdfmdlNamesSet()** adds the model names structure to the *ProUdfdata* structure.

Reading UDF Properties

The functions in this section provide the ability to read the options for placement directly from a UDF file (a .gph file) in order for an application to decide at runtime the inputs it will use for placing a given UDF. The following functions operate on *ProUdfdata*. These functions are capable of reading properties from the UDF file so long as the UDF name or path has already been set by **ProUdfdataNameSet()** or **ProUdfdataPathSet()**.

Some of the data retrieved by the functions in this section uses the same data types as the corresponding *ProUdfdata* set functions used for placing the UDF (as listed in the earlier section). However, data that you read out of the *ProUdfdata* is not related to data that you are using to place the UDF. You must explicitly pass each piece of data to the *ProUdfdata* functions if you want the UDF to be placed with this information.

Variable Dimensions

Functions Introduced:

- **ProUdfdataVardimsGet()**
- **ProUdfvardimNameGet()**
- **ProUdfvardimPromptGet()**
- **ProUdfvardimDefaultvalueGet()**

Use the function **ProUdfdataVardimsGet()** to obtain an array of available variant dimensions that may be set when placing this UDF. You can use the function **ProUdfvardimProarrayFree()** to free this *ProArray* of variant dimensions.

Note: The handles obtained when the function **ProUdfdataVardimsGet()** is called are not automatically assigned to the UDF for placement. In order to place the UDF with a user-defined variant dimension value, you must use **ProUdfdataVardimAdd()**.

Use the function **ProUdfvardimNameGet()** to obtain the symbol of the variant dimension. This symbol of the dimension in the reference model should be used in **ProUdfvardimAlloc()**.

Use the function **ProUdfvardimPromptGet()** to obtain the prompt of the variant dimension.

Use the function **ProUdfvardimDefaultvalueGet()** to obtain the default value for the variant dimension.

Variable Parameters

Functions Introduced:

- **ProUdfdataVarparamsGet()**
- **ProUdfvarparamOwnerGet()**
- **ProUdfvarparamNameGet()**
- **ProUdfvarparamDefaultvalueGet()**
- **ProUdfvarparamProarrayFree()**

Use the function **ProUdfdataVarparamsGet()** to obtain an array of available variant parameters and/or annotation values that can optionally be set when placing this UDF. You can use the function **ProUdfvarparamProarrayFree()** to free this ProArray of variant items.

Note: The handles obtained when the function **ProUdfdataVarparamsGet()** is called are not automatically assigned to the UDF for placement. In order to place the UDF with a user-defined variant parameter or annotation value, you must use **ProUdfdataVarparamAdd()**.

Use the function **ProUdfvarparamOwnerGet()** to obtain the feature or annotation element that owns this variant parameter or annotation value.

Use the function **ProUdfvarparamNameGet()** to obtain the name or the symbol of the variant parameter or annotation value.

Use the function **ProUdfvarparamDefaultvalueGet()** to obtain the default value for the variant parameter or annotation value.

UDF References

Functions Introduced:

- **ProUdfdataRequiredreferencesGet()**
- **ProUdfrequiredrefPromptGet()**
- **ProUdfrequiredrefTypeGet()**
- **ProUdfrequiredrefIsannotationref()**
- **ProUdfrequiredrefFree()**
- **ProUdfrequiredrefProarrayFree()**

Use the function **ProUdfdataRequiredreferencesGet()** to obtain a list of the references required to be set for UDF placement. In order to use this function, the UDF data must have its name or path set, and Pro/ENGINEER must be able to successfully find the .gph file based on this information.

Use the function **ProUdfrequiredrefPromptGet()** to obtain the reference prompt for a UDF reference.

Use the function **ProUdfrequiredrefTypeGet()** to obtain the type of item that should be supplied for a UDF reference.

Use the function **ProUdfrequiredrefIsannotationref()** to determine if the reference is an annotation reference and is allowed to be left incomplete.

You can use the function **ProUdfrequiredrefFree()** to free a required reference handle for a UDF. Use the function **ProUdfrequiredrefProarrayFree()** to free a *ProArray* of handles to the required references of a UDF.

External Symbols

Functions Introduced:

- **ProUdfdataExternalsymbolsGet()**
- **ProUdfextsymbolTypeGet()**
- **ProUdfextsymbolPromptGet()**
- **ProUdfextsymbolParametertypeGet()**

- **ProUdfextsymbolFree()**
- **ProUdfextsymbolProarrayFree()**

Use the function **ProUdfdataExternalsymbolsGet()** to obtain an array of external symbols required by this UDF. You can free a UDF external symbol handle using the function **ProUdfextsymbolFree()** and use the function **ProUdfextsymbolProarrayFree()** to free an array of external symbol handles.

Use the function **ProUdfextsymbolTypeGet()** to obtain the type of external symbol required (dimension or parameter).

Use the function **ProUdfextsymbolPromptGet()** to obtain the prompt for this external symbol.

Use the function **ProUdfextsymbolParametertypeGet()** used to obtain the expected parameter type for an external symbol, if the type is PRO_UDFEXTSYMBOL_PARAM.

Instance Names

Function Introduced:

- **ProUdfdataInstancenamesGet()**

Use the function **ProUdfdataInstancenamesGet()** to obtain an array of the instance names that may be used when placing this UDF. You can free this *ProArray* of instances using the function **ProArrayFree()**.

Notification on UDF Library Creation

Pro/TOOLKIT provides the ability to be notified whenever a new UDF library is created or when one is modified. You can use this notification to store additional information about the UDF library file, e.g. the names and values of parameters used in the UDF.

Functions Introduced:

- **ProUdfLibraryCompletePostAction()**

Use the function prototype **ProUdfLibraryCompletePostAction()** for a notification corresponding to the *ProNotifyType* PRO_UDF_LIB_COMPLETE_POST. This function provides the name of the newly created or modified UDF library file, and a list of all the features included in the UDF.

Note: If you modify a UDF library, which is independent and contains no reference model then no features will be included in the input to the notification.

5

Core: 3D Geometry

This chapter deals with the objects and actions used to extract the geometry of a Pro/ENGINEER solid. Because the geometry objects are closely related to each other and have a number of generic types of action in common, this chapter is organized not by object, but by types of action needed in Pro/TOOLKIT.

Some of the objects and actions in this chapter also apply to assemblies. See the Assembly: Basic Assembly Access chapter for information on objects and actions specific to assemblies.

Topic	Page
Geometry Objects	5 - 2
Visiting Geometry Objects	5 - 3
Tessellation	5 - 15
Evaluating Geometry	5 - 18
Geometry Equations	5 - 21
Ray Tracing	5 - 30
Editing Datum Points	5 - 31
Measurement	5 - 32
Geometry as NURBS	5 - 34
Interference	5 - 35
Faceted Geometry	5 - 42

Geometry Objects

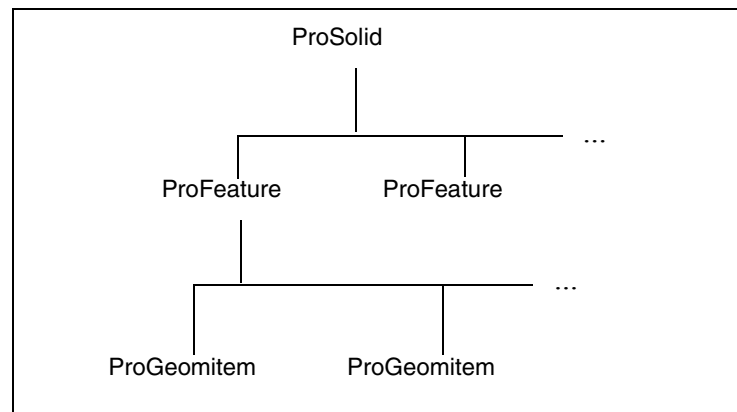
The generic object for geometry is called *ProGeomitem*, or “geometry item”. It is a DHandle that shares the declaration of *ProModelitem*. Its own instances are the specific types of geometrical item familiar to users of Pro/ENGINEER. Each of these is declared as an OHandle, or opaque handle.

The *ProGeomitem* types are as follows:

- *ProSurface*—Surface, datum surface, or datum plane
- *ProEdge*—Edge
- *ProCurve*—Datum curve
- *ProCompcurv*—Composite datum curve
- *ProQuilt*—Quilt
- *ProAxis*—Axis
- *ProPoint*—Datum point
- *ProCsys*—Datum coordinate system

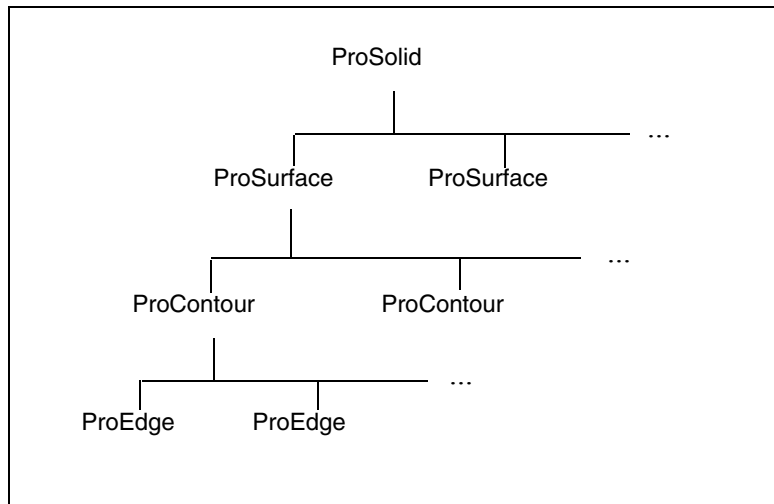
Every *ProGeomitem* is contained in a feature, and each feature is contained in a solid, as shown in the following figure.

Figure 5-1: ProGeomItem in Feature In Solid



Some geometrical items in a part are also contained in another hierarchy, which shows how they connect together geometrically, rather than to which features they belong. This type of hierarchy is shown in the following figure.

Figure 5-2: Hierarchy of Geometrical Items in a Part



The Pro/TOOLKIT object *ProContour* is also an OHandle, but has no corresponding integer identifier, and therefore is not an instance of *ProGeomitem*.

There are a number of actions applicable to many of these types, whose corresponding functions begin with “*ProGeomitem*”. These include functions such as **ProGeomitemdataGet()**, for which there are also specific functions for the subtypes (where appropriate), and some functions for generic measurement operations. These functions are described under the context of their action type.

To read and modify the name of a *ProGeomitem*, use the functions **ProModelitemNameGet()** and **ProModelitemNameSet()**, described in the chapter Core: Models and Model Items.

Visiting Geometry Objects

Visiting geometry objects means acquiring the object handles to all the geometry objects in a solid model, either in the form of a *ProGeomitem*, or in the form of the various specific opaque handles.

The term “solid” is used in Pro/TOOLKIT to distinguish models that contain three-dimensional geometry—parts and assemblies—from other model types, such as drawings. However, to the Pro/ENGINEER user, the term “solid” is used in parts and assemblies to distinguish features that represent the geometry of the design object from features used in construction only—the various types of “datum.” Within this chapter, therefore, the terms “solid geometry” and “datums” are used in that sense.

The most general way to visit geometrical items is through their features. The section Visiting Feature Geometry describes this in detail, and includes an illustration of the hierarchy used.

You can also traverse solid geometry items through the hierarchy of surfaces, contours, and edges in a part. This is described in the section Visiting Solid Geometry.

The following sections describe the traversal of the various datums. Some of these datums have their own visit functions, whereas others are visited through the feature hierarchy.

Note: Although the Pro/ENGINEER user can create solid features in Assembly mode, the geometrical items that result from them are stored only within the component parts whose geometry is modified—not in the assembly features themselves. Therefore, although traversal of datums is applicable to assemblies exactly as to parts, no solid geometry items are found in assemblies.

Datum planes, datum surfaces, and solid surfaces are all represented by the *ProSurface* object because they share the same types of mathematical description.

Visiting Feature Geometry

Functions Introduced:

- **ProSolidFeatVisit()**
- **ProFeatureStatusGet()**
- **ProFeatureTypeGet()**
- **ProFeatureVisibilityGet()**
- **ProFeatureGeomitemVisit()**
- **ProGeomitemIsInactive()**
- **ProGeomitemdataGet()**

All geometry in Pro/ENGINEER is created as a result of features, so each geometry object in Pro/TOOLKIT belongs to a feature. Therefore, the most general way to traverse geometry of all types is to traverse the features, then traverse the geometry each one contains.

The function **ProSolidFeatVisit()** visits every feature in a solid. The function **ProFeatureTypeGet()** reports the type of a feature in terms of the enumerated type *ProFeattype* (described in the include file *ProFeattype.h*).

Note that **ProSolidFeatVisit()** is designed partly for internal use within Pro/ENGINEER. It visits not only the features seen by the Pro/ENGINEER users, but also the features created internally to help in the construction of geometry. These internal features are rarely of interest to Pro/TOOLKIT users. To distinguish the visible features from the internal, or invisible, features, call the function **ProFeatureVisibilityGet()**.

The function **ProFeatureStatusGet()** reports whether a feature is suppressed or inactive for some reason—only active features contain active geometry.

The function **ProFeatureGeomitemVisit()** visits the geometry items within a feature. It can visit all the geometry items, or one of these specific types:

SURFACE, PRO_EDGE, or PRO_CURVE. Like **ProSolidFeatVisit()**, this function visits not only the visible items, but also items used internally to aid in regeneration. Use the function **ProGeomitemIsInactive()** to skip over the internal, or inactive, geometry items. For features with solid geometry, **ProFeatureGeomitemVisit()** visits not only the surfaces, but also the edges. Contrast this with the visit functions specific to those items, described in the next section, that show the hierarchical relationships between surfaces, contours, and edges.

Active geometry objects for datums will usually be found in features created for them, and therefore have the corresponding type. For example, a *ProGeomitem* object of type PRO_CSYS is usually contained in a feature of type PRO_FEAT_CSYS. However, this is not always true; a geomitem of type PRO_AXIS can exist in a feature of type PRO_FEAT_HOLE, for example. A feature of type PRO_FEAT_MERGE, which may arise from a **Mirror** operation in Part mode, or from a **Merge** in Assembly mode, contains geometry objects corresponding to all those in the referenced features, whatever their type. In general, it is best to make no assumptions about what kinds of feature in which you should look for datums.

Remember to distinguish the feature object from the geometry object it contains, even when they have a one-to-one relationship. For example, a feature of type PRO_FEAT_DATUM_AXIS contains a single geometry item of type PRO_AXIS, and each of these can be represented as a *ProModelitem* object. However, they are still distinct items with their own identifiers and types.

To extract the type and shape of each geometry item, use the function **ProGeomitemdataGet()**, described in detail in the section Geometry Equations.

Note: Some of the following sections about traversing specific geometry items introduce new functions specific to those types. PTC recommends that you use the more specific functions rather than the general method described in this section, because they are easier to use and usually have better performance.

All the functions in this section specific to features are described in detail in the chapter Core: Features.

Visiting Solid Geometry

Functions Introduced:

- **ProSolidSurfaceVisit()**
- **ProSurfaceContourVisit()**
- **ProContourEdgeVisit()**
- **ProContourTraversalGet()**
- **ProContainingContourFind()**
- **ProEdgeDirGet()**
- **ProEdgeNeighborsGet()**
- **ProEdgeVertexdataGet()**

In a Pro/ENGINEER solid, each surface contains a list of contours, and each contour contains a list of edges. The edges in a contour form a closed loop, and are ordered such that following the edges keeps the surface on the right. External contours go clockwise, and internal contours go counterclockwise.

The functions **ProSolidSurfaceVisit()**, **ProSurfaceContourVisit()**, and **ProContourEdgeVisit()** traverse all the objects in this three-level hierarchy. If you visit all the surfaces, the contours of each surface, and the edges of each contour, the resulting code visits each surface and contour one time, and each edge twice. This is true because each edge forms the intersection between two surfaces, and is therefore listed in one contour of each of the two surfaces.

The function **ProContourTraversalGet()** tells you whether the specified contour is internal or external. The function **ProContainingContourFind()** finds the innermost contour that closes the specified contour. If the specified contour is internal, the returned contour will be external, and vice versa. If the specified contour is the outermost contour for the surface, **ProContainingContourFind()** outputs NULL.

Each contour has a natural direction in terms of the order in which **ProContourEdgeVisit()** visits its edges. Each edge also has its own direction, in terms of its parameterization—the parameter, t , moves from 0 to 1 along the edge. The function **ProEdgeDirGet()** tells you whether an edge is parameterized along or against the direction of the specified contour. Note that each edge belongs to two contours, and will be in the same direction as one contour, and in the opposite direction of the other.

The function **ProEdgeNeighborsGet()** returns the two surfaces that intersect at the specified edge, and which edges on each of those surfaces is the next one following the specified edge when traversing its contour.

The function **ProEdgeVertexdataGet()** returns the list of surfaces and edges that meet at the specified vertex.

Note: The functions in this section visit active geometry items only, so you do not need to call the function **ProGeomitemIsInactive()**.

Example 1: Finding the Surfaces Penetrated by a Hole

This example uses the techniques of feature traversal and solid geometry traversal to find the surfaces that neighbor the surfaces of the selected hole.

```
typedef struct uappdata
{
    int    feature_id;
    ProMdl solid;
} UappData;
```

```

/*=====*\
FUNCTION: UserHoleSrfDisp()
PURPOSE: Display the surfaces adjacent to the selected hole.
\*=====*/
int UserHoleSrfDisp()
{
    int            n, sel_count, feat_id;
    ProFeattype     feat_type;
    ProError        status;
    ProFeature       feat;
    ProFileName      msgfile;
    ProSelection     *psels = NULL;
    ProMdl           p_solid;
    ProModelitem     p_surf, p_edge, p_mdl_item;
    UappData         appdata;
    ProError UserEdgeNeighborsEval();

/*-----*\
    Prompt the user to select a hole.
\*-----*/
    ProStringToWstring (msgfile, "msg Ug9.txt");
    ProMessageDisplay (msgfile, "UG9 Select Hole Feature:");
    if ((ProSelect ("feature", 1, NULL, NULL, NULL, NULL, &psels,
        &sel_count) != PRO_TK_NO_ERROR) || (sel_count < 1))
        return ((int) PRO_TK_GENERAL_ERROR);

    status = ProSelectionModelitemGet (psels[0], &p_mdl_item);

/*-----*\
    Verify the selection of a hole feature.
\*-----*/
    status = ProFeatureTypeGet (&p_mdl_item, &feat_type);
    if (feat_type != PRO_FEAT_HOLE)
    {
        ProMessageDisplay (msgfile, "UG9 Feature is not a hole");
        return ((int) PRO_TK_GENERAL_ERROR);
    }

/*-----*\
    Use the UappData structure (cast to ProAppData type)
    to pass data between the visit function and here.
\*-----*/
    appdata.feature_id = p_mdl_item.id;
    status = ProMdlCurrentGet (&(appdata.solid));
    status = ProFeatureGeomitemVisit (&p_mdl_item, PRO_EDGE,
        UserEdgeNeighborsEval, NULL, (ProAppData)&appdata);
    return ((int) PRO_TK_NO_ERROR);
}

/*=====*\
FUNCTION: UserEdgeNeighborsEval()
PURPOSE: Evaluate the surfaces adjacent to the edges by
        comparing the feature identifier to the feature
        identifier of the hole.
\*=====*/

```

```

ProError UserEdgeNeighborsEval (
    ProGeomitem    *p_edge,    /* input from the visit function */
    ProError        status,    /* output from filter function */
    ProAppData      p_inp      /* user application data - input in
                                this case */
)
{
    int            n;
    UappData       *ud = (UappData *)p_inp;
    ProEdge        this_edge, next1, next2;
    ProSurface      surf[2];
    ProGeomitem     geom_item;
    ProSelection    selection;
    ProFeature      feat;

/*-----*\
    Look at both surfaces attached to this edge and highlight
    whichever does not belong to this feature.
\*-----*/
    status = ProGeomitemToEdge (p_edge, &this_edge);
    status = ProEdgeNeighborsGet (this_edge, &next1, &next2, &surf[0],
                                &surf[1]);
    for (n = 0; n < 2; n++)
    {
        status = ProSurfaceToGeomitem (ud->solid, surf[n], &geom_item);
        status = ProGeomitemFeatureGet (&geom_item, &feat);
        if (feat.id != ud->feature_id)
        {
            status = ProSelectionAlloc (NULL, &geom_item, &selection);
            status = ProSelectionHighlight (selection,
                                           PRO_COLOR_HIGHLIGHT);
            status = ProSelectionFree (&selection);
        }
    }
    return ((int)PRO_TK_NO_ERROR);
}

```

Visiting Axis Datums

Functions Introduced:

- **ProSolidAxisVisit()**
- **ProAxisIdGet()**
- **ProAxisInit()**
- **ProGeomitemFeatureGet()**
- **ProAxisSurfaceGet()**

An axis is represented by the object *ProAxis*, which is declared as an opaque handle. The function **ProSolidAxisVisit()** visits all the axes in a part or assembly. An axis created explicitly using the Pro/ENGINEER command **Feature, Create, Datum, Axis** will be contained in a feature of type PRO_FEAT_DATUM_AXIS, but axes can also exist in features of other types, such as PRO_FEAT_HOLE.

To find the feature that an axis belongs to, describe the axis in terms of a *ProGeomitem* object using the functions **ProAxisIdGet()** and **ProModelitemInit()**, then call **ProGeomitemFeatureGet()**.

You could also traverse axes using the functions **ProSolidFeatVisit()** and **ProFeatureGeomitemVisit()**, described in the section Visiting Feature Geometry. As input to **ProFeatureGeomitemVisit()**, use the type PRO_AXIS. You would have to visit features of any type that could contain axes.

Always remember to use the function **ProGeomitemIsInactive()** to skip over axes used internally only.

The function **ProAxisSurfaceGet()** provides the *ProSurface* object that identifies the surface used to define the axis position.

Visiting Coordinate System Datums

Functions Introduced:

- **ProSolidCsysVisit()**
- **ProCsysIdGet()**
- **ProCsysInit()**

A coordinate system datum is represented by the object *ProCsys*, which is declared as an opaque handle. The function **ProSolidCsysVisit()** visits all the coordinate system datums in a part or an assembly.

You could also traverse the coordinate system datums using the **ProSolidFeatVisit()** and **ProFeatureGeomitemVisit()** functions, described in the section Visiting Feature Geometry. The coordinate system datums are usually found in features of type PRO_FEAT_CSYS, although they can appear in others, and have the geomitem type PRO_CSYS.

Always remember to use the function **ProGeomitemIsInactive()** to skip over coordinate system datums used internally only.

The function **ProCsysIdGet()** provides the persistent integer identifier of the coordinate system, which is used if you need to make a *ProGeomitem* representation of the coordinate system. The function **ProCsysInit()** creates a *ProCsys* object from the integer identifier.

Visiting Datum Planes

Functions Introduced:

- **ProSurfaceInit()**
- **ProSurfaceIdGet()**

A datum plane is represented by the object *ProSurface*, which is declared as an opaque handle and is also used to represent solid surfaces and datum surfaces.

To visit all the datum planes, use the functions **ProSolidFeatVisit()** and **ProFeatureGeomitemVisit()**, described in the section Visiting Feature Geometry. The datum planes are contained in features of type PRO_FEAT_DATUM, each of which contains a single active *ProGeomitem* object whose *type* field is PRO_SURFACE.

Always remember to use the function **ProGeomitemIsInactive()** to skip over datum planes used internally only. (Active datum planes occur in features of type PRO_FEAT_DATUM only; datum planes created on-the-fly during creation of other features are inactive.)

To convert the *ProGeomitem* to a *ProSurface*, use the *id* field in the *ProGeomitem* as input to the function **ProSurfaceInit()**.

The function **ProSurfaceIdGet()** gives the integer identifier of a *ProSurface*, so you can convert back to a *ProGeomitem* using the function **ProModelitemInit()**.

Note: Although a datum plane has a nominal outline used to visualize the datum in the Pro/ENGINEER display, this is not part of the geometry because a datum plane is an infinite, unbounded plane. Therefore, if you try to use the function **ProSurfaceContourVisit()** on a datum plane, it will not find any contours.

Visiting Quilts and Datum Surfaces

Functions Introduced:

- **ProSolidQuiltVisit()**
- **ProQuiltSurfaceVisit()**
- **ProQuiltIdGet()**
- **ProQuiltInit()**

A datum surface is represented by the object *ProSurface*, which is declared as an opaque handle and is also used to represent solid surfaces and datum planes.

From the viewpoint of Pro/TOOLKIT, every datum surface belongs to a quilt, even if no explicit quilt feature has been made in Pro/ENGINEER. If the user creates a single datum surface, it belongs to an internal quilt created for that purpose.

A quilt is represented by the object *ProQuilt*, which is declared as an opaque handle.

To visit datum surfaces, you must therefore first visit all the quilts, using **ProSolidQuiltVisit()**, then visit each surface in the quilt using **ProQuiltSurfaceVisit()**.

The function **ProSolidQuiltVisit()** takes the filter function **ProQuiltFilterAction()** and the visit function **ProQuiltVisitAction()** as its input arguments. The function **ProQuiltFilterAction()** is a generic action function for filtering quilts from a solid model. It returns the filter status of the quilts. This status is used as an input argument by the visit action function **ProQuiltVisitAction()**.

The function **ProQuiltSurfaceVisit()** takes the filter function **ProQuiltSurfaceFilterAction()** and the visit function **ProQuiltSurfaceVisitAction()** as its input arguments. The function **ProQuiltSurfaceFilterAction()** is a generic action function for filtering datum surfaces in a quilt. It returns the filter status of the datum surfaces. This status is used as an input argument by the visit function **ProQuiltSurfaceVisitAction()**.

Always remember to use the function **ProGeomitemIsInactive()** to skip over quilts and datum surfaces used internally only.

To convert a *ProQuilt* object to a *ProGeomitem*, use the functions **ProQuiltIdGet()** and **ProModelitemInit()**.

To create a *ProQuilt* object from the integer identifier, use **ProQuiltInit()**.

To find the contours and edges of a datum surface, use the visit functions **ProSurfaceContourVisit()** and **ProContourEdgeVisit()**, described in the section Visiting Solid Geometry.

Visiting Datum Curves

Functions Introduced:

- **ProCurveIdGet()**
- **ProCurveInit()**
- **ProCurvePersistentColorGet()**
- **ProCurvePersistentColorSet()**
- **ProCurvePersistentLineStyleGet()**
- **ProCurvePersistentLineStyleSet()**

A datum curve is represented by the object *ProCurve*, which is declared as an opaque handle.

To visit all the datum curves, use the functions **ProSolidFeatVisit()** and **ProFeatureGeomitemVisit()**, described in the section Visiting Feature Geometry. The datum curves are contained in features of many different types, each of which contains one or more active *ProGeomitem* objects whose *type* field is PRO_CURVE.

Always remember to use the function **ProGeomitemIsInactive()** to skip over datum curves used internally only.

To convert a *ProCurve* object to a *ProGeomitem*, use the functions **ProCurveIdGet()** and **ProModelitemInit()**.

To create a *ProCurve* object from the integer identifier, use **ProCurveInit()**.

Use the functions **ProCurvePersistentColorGet()** and **ProCurvePersistentColorSet()** to obtain and set the color of a specified curve. In order to view the color changes, use the function **ProDisplistInvalidate()** on the owner model.

Use the functions **ProCurvePersistentLineStyleGet()** and **ProCurvePersistentLineStyleSet()** to obtain and set the linestyle of a specified curve. In order to view the linestyle changes, use the function **ProDisplistInvalidate()** on the owner model.

Visiting Composite Datum Curves

Function Introduced:

- **ProCurveCompVisit()**

A composite datum curve is also represented by the object *ProCurve*. To distinguish a composite curve from an ordinary curve when dealing with a *ProCurve* object, use the function **ProCurveTypeGet()**. This function outputs the value `PRO_ENT_CMP_CRV` for a composite curve.

To visit all the composite datum curves, use the functions **ProSolidFeatVisit()** and **ProFeatureGeomitemVisit()**, described in the section Visiting Feature Geometry.

The composite curves are contained in features of many different types, each of which contains one or more active *ProGeomitem* objects whose *type* field is `PRO_CURVE`.

To visit the datum curves in a composite curve, use the function **ProCurveCompVisit()**.

Remember that each curve in a composite may be a composite itself, so you may need to make recursive calls. However, you can find all non-composite curves, including those contained in composites, using the method described in the previous section. It is therefore unnecessary to traverse all the composite curves to find all the non-composite curves.

Visiting Datum Points

Functions Introduced:

- **ProPointIdGet()**
- **ProPointInit()**

A datum point is represented by the object *ProPoint*, which is declared as an opaque handle.

To visit all the datum points, use the functions **ProSolidFeatVisit()** and **ProFeatureGeomitemVisit()**, described in the section Visiting Feature Geometry. The datum points are usually contained in features of type `PRO_FEAT_DATUM_POINT`, although they can also occur in others, such as `PRO_FEAT_MERGE`. Datum points are represented by geometry items of type `PRO_POINT`.

Always remember to use the function **ProGeomitemIsInactive()** to skip over datum points used internally only.

To convert a *ProPoint* object to a *ProGeomitem*, use the functions **ProPointIdGet()** and **ProModelitemInit()**.

To create a *ProPoint* object from the integer identifier, use **ProPointInit()**.

Tessellation

You can calculate tessellation for different types of Pro/ENGINEER geometry. The tessellation is made up of small lines (for edges and curves), or triangles (for surfaces and solid models).

Curve and Edge Tessellation

Functions Introduced:

- **ProEdgeTessellationGet()**
- **ProCurveTessellationGet()**

The function **ProEdgeTessellationGet()** enables you to invoke the algorithm that generates a sequence of lines from an arbitrary curved edge. This function provides the following outputs:

- An array of the XYZ coordinates of the vertices between the tessellations
- The two surfaces that neighbor the edge (as also provided by **ProEdgeNeighborsGet()**)
- An array of *uv* pairs for the tessellation vertices in the first neighboring surface
- An array of *uv* pairs for the second neighboring surface
- The number of tessellation vertices

The function **ProCurveTessellationGet()** retrieves the curve tessellation for a datum curve. It returns the number of tessellation points and a list of them.

Surface Tessellation

Functions Introduced:

- **ProSurfaceTessellationGet()**
- **ProTessellationFree()**
- **ProTessellationVerticesGet()**
- **ProTessellationFacetsGet()**
- **ProTessellationNormalsGet()**
- **ProTessellationParamsGet()**
- **ProSurfacetessellationinputAlloc()**
- **ProSurfacetessellationinputFree()**
- **ProSurfacetessellationinputChordheightSet()**
- **ProSurfacetessellationinputAnglecontrolSet()**
- **ProSurfacetessellationinputUvprojectionSet()**

The function **ProSurfaceTessellationGet()** calculates the tessellation data given by the *ProTessellation* object for a specified surface. Use the function **ProTessellationFree()** to release the memory used by this data object.

The function **ProTessellationVerticesGet()** obtains the vertices for the tessellation for a specified surface.

The function **ProTessellationFacetsGet()** obtains the indices indicating the vertices used for each facet of the tessellated item.

The function **ProTessellationNormalsGet()** obtains the normal vectors for each of the tessellation vertices.

The function **ProTessellationParamsGet()** obtains the UV parameters for each of the tessellation vertices.

The function **ProSurfacetessellationinputAlloc()** allocates the *ProSurfaceTessellationInput* data object containing the options for surface tessellation. Use the function **ProSurfacetessellationinputFree()** to release the memory allocated to this data object.

The function **ProSurfacetessellationinputChordheightSet()** assigns the chord height used for surface tessellation.

The function **ProSurfacetessellationinputAnglecontrolSet()** assigns the value of the angle control used for surface tessellation.

The function **ProSurfaceTessellationInputUvprojectionSet()** assigns the parameters used to calculate the UV projection for the texture mapping to the tessellation inputs. The types of UV projection are given by the enumerated type *ProSurfaceTessellationProjection*, and are as follows:

- **PRO_SRFTESS_DEFAULT_PROJECTION**—Provides the UV parameters for the tessellation points that map to a plane whose U and V extents are [0,1] each. This is the default projection.
- **PRO_SRFTESS_PLANAR_PROJECTION**—Projects the UV parameters using a planar transform, where $u=x$, $v=y$, and z is ignored.
- **PRO_SRFTESS_CYLINDRICAL_PROJECTION**—Projects the UV parameters using a cylindrical transform, where $x=r*\cos(\theta)$, $y=r*\sin(\theta)$, $u=\theta$, $v=z$, and r is ignored.
- **PRO_SRFTESS_SPHERICAL_PROJECTION**—Projects the UV parameters onto a sphere, where $x=r*\cos(\theta)*\sin(\phi)$, $y=r*\sin(\theta)*\sin(\phi)$, $z=r*\cos(\phi)$, $u=\theta$, $v=\phi$, and r is ignored.
- **PRO_SRFTESS_NO_PROJECTION**—Provides the unmodified UV parameters for the tessellation points. This is similar to using the function **ProSurfaceParamEval()**.

Note:

- If the function **ProSurfaceTessellationInputUvprojectionSet()** is not used, the output tessellation will not contain any UV parameters and the function **ProTessellationParamsGet()** will not return any values.
- Specify the unmodified UV parameters obtained using **PRO_SRFTESS_NO_PROJECTION** as the input u and v values for the functions **ProSurfaceXyzdataEval()**, **ProSurfaceUvpntVerify()**, **ProSurfaceDiameterEval()**, and **ProSurfacePrincipalCrvtEval()**.

Part and Assembly Tessellation

Functions Introduced:

- **ProPartTessellate()**
- **ProPartTessellationFree()**

The function **ProPartTessellate()** tessellates all surfaces of the specified part or assembly in one operation. On parts, **ProPartTessellate()** acts on all surfaces. On assemblies, this function acts only on surfaces that belong to the assembly, that is, it does not tessellate surfaces of the assembly components. **ProPartTessellate()** returns an array of *ProSurfaceTessellationData* data objects. Use the function **ProPartTessellationFree()** to release the memory assigned to these data objects.

Evaluating Geometry

The geometry of each edge or curve in Pro/ENGINEER is described as a set of three parametric equations that represent the values of X, Y, and Z as functions of the independent parameter, t . For a surface, the three equations are functions of the two independent parameters u and v .

The Pro/TOOLKIT functions described in this section provide the ability to evaluate the parametric edge and surface functions—that is, find the values and derivatives of X, Y and Z for the specified values of t , or u and v . They also provide for reverse evaluation.

Evaluating Surfaces, Edges, and Curves

Functions Introduced:

- **ProSurfaceXyzdataEval()**
- **ProEdgeXyzdataEval()**
- **ProCurveXyzdataEval()**
- **ProEdgeUvdataEval()**
- **ProSurfaceUvpntVerify()**
- **ProContourUvpntVerify()**

The function **ProSurfaceXyzdataEval()** evaluates the parametric equations for a surface at a point specified by its u and v values. The inputs to the function are the *ProSurface* object and the u and v values. The u and v values are obtained by specifying the projection type as PRO_SRFTESS_NO_PROJECTION for the function **ProSurfacetessellationinputUvprojectionSet()**.

The function outputs are as follows:

- The X, Y, and Z coordinates of the point, with respect to the model coordinates

- The first partial derivatives of X, Y, and Z, with respect to u and v
- The second partial derivatives of X, Y, and Z, with respect to u and v
- A unit vector in the direction of the *outward* normal to the surface at that point

The function **ProEdgeXYZdataEval()** performs a similar role for an edge. Its inputs are the *ProEdge* object and the value of t at the required point. The function outputs are as follows:

- The X, Y, and Z coordinates of the point, with respect to the model coordinates
- The first partial derivatives of X, Y, and Z, with respect to t
- The second partial derivatives of X, Y, and Z, with respect to t
- A unit vector in the direction of the edge

You must allocate a memory location for each of the output arguments of these two functions. Pass a NULL pointer if you do not want to use an output argument. You cannot pass a null for both the output arguments.

The function **ProCurveXYZdataEval()** is equivalent to **ProEdgeXYZdataEval()**, but works for datum curves.

The **ProEdgeUvdataEval()** function relates the geometry of a point on an edge to the surfaces that meet at that point.

The function **ProSurfaceUvpntVerify()** verifies whether a surface point, specified by its u and v values, lies inside, outside, or very close to the boundary of the surface. The u and v values are obtained by specifying the projection type as PRO_SRFTESS_NO_PROJECTION for the function **ProSurfaceTessellationInputUvprojectionSet()**.

Function **ProContourUvpntVerify()** does the same for points on a given contour.

Inverse Evaluation and Minimum Distances

Functions Introduced:

- **ProSurfaceParamEval()**
- **ProEdgeParamEval()**
- **ProCurveParamEval()**

These functions provide the parameters of a point on a surface, edge, or datum curve nearest to the specified XYZ coordinate point.

You can use the function **ProEdgeParamEval()** only for the points that are either on the edge or very close to the edge.

The function **ProSurfaceParamEval()** returns the closest approximation to the unmodified u and v values obtained by specifying the projection type as **PRO_SRFTESS_NO_PROJECTION** for the function **ProSurfacetessellationinputUvprojectionSet()**.

Geometry at Points

Functions Introduced:

- **ProGeometryAtPointFind()**
- **ProPoint3dOnsurfaceFind()**
- **ProPoint3dIntoleranceFind()**
- **ProSolidProjectPoint()**

The function **ProGeometryAtPointFind()** locates the geometry items that lie on a given point. This function supports solid geometry only.

The function **ProPoint3dOnsurfaceFind()** determines if the distance between the specified point and the specified surface is less than the Pro/ENGINEER model accuracy as set in the current Pro/ENGINEER session. Accuracy can also be set with function **ProSolidAccuracySet()**. This function is applicable to solid and datum surfaces.

The function **ProPoint3dIntoleranceFind()** determines if two specified points are co-incident, that is, if the distance between the two points is within the Pro/ENGINEER tolerance set in **ProSolidToleranceGet()**.

The function **ProSolidProjectPoint()** projects a point along the shortest possible line normal to a surface, finds the point where that line hits the solid, and returns that point. Note that this function works on parts only.

Geometry Equations

Functions Introduced:

- **ProGeomitemdataGet()**
- **ProGeomitemdataFree()**

The parametric equations that describe surfaces, edges, and datum curves in Pro/ENGINEER are documented in the Geometry Representations appendix. (Datum curves are geometrically equivalent to edges, but because they play a different role in Pro/ENGINEER, they need a parallel set of functions to access them. The word curve is used as a generic word for the shape of either an edge or a datum curve.)

To know the form of a particular geometry item, you need to know not only which type of equation is being used, but also the values of the various coefficients and constants used in that equation for that item. Geometry Representations documents the equations using the same names for these coefficients and constants used to store them in the Pro/ENGINEER data structures. The functions in this section enable you to get copies of the data structures containing those coefficients and constants. Therefore, you can perform your own evaluations.

The data structures for *ProSurfacedata* are defined in the include file *ProSurfacedata.h*, and those for *ProCurvedata* are defined in *ProCurvedata.h*.

The function **ProGeomitemdataGet()** allocates and fills a data structure that describes the geometry of the item. The structure *ProGeomitemdata* is declared in the file *ProGeomitemdata.h*, and looks like this:

```
typedef struct geom_item_data_struct
{
    ProType          obj_type;
    union
    {
        ProCurvedata    *p_curve_data;
        ProSurfacedata   *p_surface_data;
        ProCsysdata      *p_csys_data;
    } data;
} ProGeomitemdata;
```

The *type* field has the same value as the *type* field in the *ProGeomitem* object.

The three fields in the union contain data structures for the geometry of curves (including solid edges and axes), surfaces, and coordinate system datums. These three data structures are described in detail in the sections Geometry of Solid Edges, Geometry of Surfaces, and Geometry of Coordinate System Datums, respectively.

The memory for the data structure is allocated by the function, but is never freed. To free the memory when you have finished with it, call **ProGeomitemdataFree()**.

Geometry of Solid Edges

Functions Introduced:

- **ProEdgeTypeGet()**
- **ProEdgeDataGet()**
- **ProEdgedataMemoryFree()**
- **ProEdgedataFree()**

Function **ProEdgeTypeGet()** provides the equation used to describe the edge. Function **ProEdgeDataGet()** returns the data structure associated with the specified edge.

Use function **ProEdgedataMemoryFree()** to free the top-level memory associated with the edge data structure. Function **ProEdgedataFree()** frees the underlying memory of the data structure.

Follow these steps to get the description of an edge:

1. Get the type of equation used to describe the edge using the function **ProEdgeTypeGet()**. The possible types for a solid edge are as follows:
 - **PRO_ENT_LINE**—A straight line
 - **PRO_ENT_ARC**—An arc
 - **PRO_ENT_ELLIPSE**—An ellipse
 - **PRO_ENT_SPLINE**—A nonuniform cubic spline
 - **PRO_ENT_B_SPLINE**—A nonuniform rational B-spline (NURBS)

Several other types of shape can be described by the *ProCurvedata* structure, but only these four types are valid for solid edges.

2. Get the data structure for the geometry using the function **ProEdgeDataGet()**. For an edge, the *type* field is set to PRO_EDGE, and the relevant field from the union is *p_curve_data*. The type for that field, *ProCurvedata*, is itself a union that contains a field for each type of edge equation. For example, if the edge type is PRO_ENT_ARC, the relevant field in the *ProCurvedata* structure is the one called *arc*, of type *ProArcdata*. Each such structure contains fields for the coefficients and constants in the relevant equations (described in the Geometry Representations appendix), and share the same names.
3. When you have read the information you need from the *ProGeomitemdata* structure, free the memory using **ProGeomitemdataFree()**.

Example 2: Extracting the Diameter of an Arc Edge

This example shows how to extract the geometry equation of a solid edge.

```

/*----- Pro/Toolkit Includes -----*/
#include <ProToolkit.h>
#include <ProSelection.h>
#include <ProGeomitem.h>
#include <ProUtil.h>
#include <ProMessage.h>
#include <ProEdge.h>

/*----- Application Includes -----*/
#include <TestError.h>

/*----- Function Prototypes -----*/
ProError UserArcDiaDisp();

/*=====*\
Function: UserArcDiaDisp()
Purpose: Display Diameter of Selected Arc
\*=====*/
ProError UserArcDiaDisp()
{
    int sel_count;
    double diameter;
    ProError status;
    ProModelitem p_mdl_item;
    ProFileName msgfile;
    ProGeomitemdata *geom_data=NULL;
    ProArcdata *p_arc=NULL;
    ProEdge edge;

```

```

    ProEnttype edge_type;
    ProSelection *psels=NULL;

/*-----*\
    Prompt user for selection of arc
\*-----*/
    ProStringToWstring(msgfile,"msg_uggeom.txt");
    status = ProMessageDisplay(msgfile,"USER Select Arc to Evaluate:");
    ERROR_CHECK("UserArcDiaDisp","ProMessageDisplay(Select Arc)",status);
    if((ProSelect("edge",1,NULL,NULL,NULL,NULL,&psels, &sel_count) !=
        PRO_TK_NO_ERROR) || (sel_count < 1))
        return((int)PRO_TK_GENERAL_ERROR);

    status = ProSelectionModelitemGet(psels[0],&p_mdl_item);
    ERROR_CHECK("UserArcDiaDisp","ProSelectionModelitemGet", status );

    status = ProGeomitemToEdge(&p_mdl_item,&edge);
    ERROR_CHECK("UserArcDiaDisp","ProGeomitemToEdge",status);

    status = ProEdgeTypeGet(edge,&edge_type);
    ERROR_CHECK("UserArcDiaDisp","ProEdgeTypeGet",status);
    if((edge_type != PRO_ENT_ARC) && (edge_type != PRO_ENT_CIRCLE))
    {
        status = ProMessageDisplay(msgfile,"USER Invalid Arc Selection");
        ERROR_CHECK("UserArcDiaDisp",
            "ProMessageDisplay(Invalid Arc)",status);
        return((int)PRO_TK_GENERAL_ERROR);
    }

    status = ProGeomitemdataGet(&p_mdl_item,&geom_data);
    ERROR_CHECK("UserArcDiaDisp","ProGeomitemdataGet",status);

    diameter = geom_data->data.p_curve_data->arc.radius*2.0;

    status = ProMessageDisplay(msgfile,
        "USER Arc Diameter = %0f",&diameter);
    ERROR_CHECK("UserArcDiaDisp",
        "ProMessageDisplay(Arc Diameter)",status);

    ProGeomitemdataFree(&geom_data);
    return((int)status);
}

```

Geometry of Surfaces

Functions Introduced:

- **ProSurfaceTypeGet()**
- **ProSurfaceDataGet()**
- **ProSurfacelsDatumPlane()**
- **ProSurfaceSameSrfsFind()**
- **ProSldsurfaceVolumesFind()**

The method for getting the description of surface geometry is analogous to that described in the previous section for solid edges. Function **ProSurfaceTypeGet()** provides the equation used to describe the surface. Function **ProSurfaceDataGet()** returns the data structure associated with the specified surface.

Use function **ProSurfacedataMemoryFree()** to free the top-level memory associated with the surface data structure. Function **ProSurfacedataFree()** frees the underlying memory of the data structure.

The possible types of surface are as follows:

- PRO_SRF_PLANE—A plane
- PRO_SRF_CYL—A cylinder
- PRO_SRF_CONE—A cone
- PRO_SRF_TORUS—A torus
- PRO_SRF_COONS—A Coons patch
- PRO_SRF_SPL—A spline surface
- PRO_SRF_FIL—A fillet surface
- PRO_SRF_RUL—A ruled surface
- PRO_SRF_REV—A surface of revolution
- PRO_SRF_TABCYL—A tabulated cylinder
- PRO_SRF_B_SPL—A nonuniform rational B-spline (NURBS)
- PRO_SRF_FOREIGN—A foreign surface (as created by the Pro/TOOLKIT function **prosrfs_bind_srf_class()**)
- PRO_SRF_CYL_SPL—A cylindrical spline surface

The relevant field in the *ProGeomitemdata* structure is *p_surface_data*, of type *ProSurfacedata*.

The structure *ProSurfacedata* contains information applicable to surfaces of all types, such as the maximum and minimum values of *u* and *v*, and of X, Y, and Z for the surface, and a union that contains a field for each type of surface geometry.

As with edges, these structures contain fields for the coefficients and constants in the relevant equations, described in the Geometry Representations appendix.

These functions are also applicable to datum surfaces, and to datum planes (in which the surface type will always be PRO_SRF_PLANE).

The function **ProSurfaceIsDatumPlane()** identifies if the given surface is a datum plane.

The function **ProSurfaceSameSrfsFind()** finds and returns an array of surfaces that are the same as the input surface. For example, in case of a cylinder, Pro/ENGINEER creates two half cylindrical surfaces. If you obtain one half, the other half is returned by this function.

The function **ProSldsurfaceVolumesFind()** analyzes and returns the number of connect volumes of a part and the surfaces that bound them.

Example 3: Getting the Angle of a Conical Surface

This example shows how to read the geometry equation of a surface.

```
/*=====*\
FUNCTION: UserConeAngDisp()
PURPOSE: Display the angle of the selected cone.

\*=====*/

int UserConeAngDisp()
{
    int          sel_count;
    double       angle;
    ProError     status;
    ProModelitem p_mdl_item;
    ProFileName  msgfile;
    ProGeomitemdata *geom_data = NULL;
    ProSurface   surface;
    ProSrftype   surface_type;
    ProSelection *psels = NULL;

/*-----*\

    Prompt the user to select a cone.
```

```

\*-----*/

ProStringToWstring (msgfile, "msg_ug7.txt");
status = ProMessageDisplay (msgfile,
    "UG7 Select Cone to Evaluate:");
if ((ProSelect ("surface", 1, NULL, NULL, NULL, NULL, &psels,
    &sel_count) != PRO_TK_NO_ERROR) || (sel_count < 1))
    return ((int)PRO_TK_GENERAL_ERROR);

status = ProSelectionModelitemGet (psels[0], &p_mdl_item);
status = ProGeomitemToSurface (&p_mdl_item, &surface);
status = ProSurfaceTypeGet (surface, &surface_type);
if (surface_type != PRO_SRF_CONE)
{
    ProMessageDisplay (msgfile,
        "UG7 Surface selected is not a Cone");
    return ((int)PRO_TK_GENERAL_ERROR);
}

status = ProGeomitemdataGet (&p_mdl_item, &geom_data);
angle = fabs(
    geom_data->data.p_surface_data->srf_shape.cone.alpha*180.0/PI);
status = ProMessageDisplay (msgfile,
    "UG7 Cone angle is %0f",&angle);
return ((int)status);
}

```

Geometry of Axes

Function Introduced:

- **ProAxisDataGet()**

An axis is treated in the same way as a solid edge. The function **ProAxisDataGet()** allocates and fills a *ProGeomitemdata* structure for a *ProAxis* object. The relevant field in the union is *p_curve_data*, but the type of the contained edge is always a line.

Geometry of Coordinate System Datums

Function Introduced:

- **ProCsysDataGet()**

The function **ProCsysDataGet()** provides a *ProGeomitemdata* structure in which the field *p_csys_data* is set. This is a pointer to a structure called *ProCsysdata*, declared in *ProCsysdata.h*, that contains the location of the origin, and the directions of the three axes, of the coordinate system datum.

Geometry of Datum Planes

Datum planes are treated exactly like surfaces, so you can use **ProSurfaceDataGet()**.

Their type is always PRO_SRF_PLANE.

Geometry of Quilts

Functions Introduced:

- **ProQuiltdataGet()**
- **ProQuiltdataSurfArrayGet()**
- **ProQuiltdataSurfArraySet()**
- **ProQuiltdataMemoryFree()**
- **ProQuiltdataFree()**
- **ProQuiltVolumeEval()**
- **ProQuiltIsBackupgeometry()**

A quilt is simply a container for datum surfaces, and does not have its own geometry independent of the surfaces it contains. To find the surfaces it contains, use **ProQuiltSurfaceVisit()** and analyze the geometry of each surface.

The function **ProQuiltdataGet()** retrieves information from the quilt data structure. The helper functions **ProQuiltdataSurfArrayGet()** and **ProQuiltdataSurfArraySet()** return or define, respectively, an array of pointers to the datum surfaces in the quilt data structure.

The function **ProQuiltdataMemoryFree()** releases the top-level memory associated with the quilt data structure. The function **ProQuiltdataFree()** releases the underlying memory of the data structure.

The function **ProQuiltVolumeEval()** calculates the volume of a closed quilt.

The function **ProQuiltIsBackupgeometry()** identifies if the specified quilt belongs to the invisible Copy Geometry backup feature. Its input argument is a pointer to the quilt's handle of the type *ProQuilt*. If the quilt belongs to the invisible Copy Geometry backup feature, the function returns a *ProBoolean* with the value PRO_B_TRUE; otherwise, the value is PRO_B_FALSE.

Geometry of Datum Surfaces

Because the system treats datum surfaces exactly like surfaces, you can use **ProSurfaceDataGet()**.

They can have any type of geometry.

Geometry of Datum Points

Functions Introduced:

- **ProPointCoordGet()**

The function **ProPointCoordGet()** provides the X, Y, and Z coordinates of the specified *ProPoint* object.

Geometry of Datum Curves

Functions Introduced:

- **ProCurveTypeGet()**
- **ProCurveDataGet()**
- **ProCurvedataMemoryFree()**
- **ProCurvedataFree()**

Datum curves use the same data structure as edges, with the same possible types of geometry. Because they are stored in a different location in the Pro/ENGINEER database, they need their own functions:

- **ProCurveTypeGet()** is analogous to **ProEdgeTypeGet()**.
- **ProCurveDataGet()** is analogous to **ProEdgeDataGet()**.

Use function **ProCurvedataMemoryFree()** to free the top-level memory associated with the curve data structure. Function **ProCurvedataFree()** frees the underlying memory of the data structure.

Geometry of Composite Curves

A composite curve does not have geometry of its own. Find the member curves using **ProCurveCompVisit()** and analyze their geometry in the usual way.

Ray Tracing

Functions Introduced:

- **ProSolidRayIntersectionCompute()**
- **ProSelectionDepthGet()**

The function **ProSolidRayIntersectionCompute()** finds the intersections between a ray and a solid.

The ray is defined in terms of a start location and direction vector. The intersections are described in terms of an array of *ProSelection* objects to show their context in an assembly.

The function finds intersections in both directions from the start point of the ray, and assigns each intersection a depth—the distance from the ray start point in the direction defined (intersections in the reverse direction have a negative depth). You can extract the depth of each intersection using the function **ProSelectionDepthGet()**. The intersections are ordered from the most negative depth to the most positive.

The function processes all solid surfaces and datum surfaces, but not datum planes. It also includes edges that lie within a certain critical distance, called the aperture radius, of the ray. Such an edge is shown as intersected, even if a neighboring surface is also intersected. This implies that several entries in the array may represent a single “piercing,” in geometrical terms.

The aperture radius is an optional input to the function, defined in terms of pixels. If you supply a value less than -1.0, the value is taken from the Pro/ENGINEER configuration file option “pick_aperture_radius.” If that option is not set, the function uses the default value of 7.0.

In an assembly, each component is processed separately, so if two coincident mating faces are hit, the function records two separate intersections.

Surfaces and edges that are not displayed because they are assigned to a blanked layer are not intersected.

This function is most often used in optical analysis, when calculating the path of a ray of light through an assembly whose parts represent lenses or mirrors. In this case, you want to find the closest intersecting surface in the positive direction, then calculate the normal to the surface at that point, in assembly coordinates.

Editing Datum Points

Functions introduced:

- **pro_dtmpntarr_add_pnt()**
- **pro_dtmpntarr_insert_pnt()**
- **pro_dtmpntarr_delete_pnt()**

These functions enable you to edit the datum points in a datum point array—that is, a single datum point feature that contains several datum points. These functions retain the Pro/DEVELOP style.

Note: These functions work on parametric datum point features only. If you use these functions on datum point features that were created using the **Without Dims** option, the features are not modified.

The function **pro_dtmpntarr_add_pnt()** adds a new datum point as the last item in the array.

The function **pro_dtmpntarr_insert_pnt()** adds a new datum point anywhere in the list.

The function **pro_dtmpntarr_delete_pnt()** deletes the specified datum point.

All the functions identify the datum point array by the integer feature identifier, and the point itself by the integer identifier.

To select a datum point using Pro/TOOLKIT, call **ProSelect()** with the option “point.” You can then obtain the feature identifier using **ProSelectionModelitemGet()**, **ProGeomitemFeatureGet()**, and then using the *id* field in the *ProFeature* object. The datum point identifier is the *id* field in the *ProModelitem* object.

To traverse the datum points in a feature, use the function **ProFeatureGeomitemVisit()**.

For those users who are familiar with Pro/DEVELOP:

To select a datum point using Pro/DEVELOP-style functions, call **pro_select()** with the option “point.” The feature identifier is the field *selected_id*, and the datum point identifier is *sel_elem_id*.

To traverse the datum points in a feature using Pro/DEVELOP-style functions, use the function **prodb_get_feature_dtm_points()**.

Measurement

Functions Introduced:

- **ProSurfaceAreaEval()**
- **ProContourAreaEval()**
- **ProSurfaceExtremesEval()**
- **ProSurfacePrincipalCrvtEval()**
- **ProEdgeLengthEval()**
- **ProCurveLengthEval()**
- **ProEdgeLengthT1T2Eval()**
- **ProCurveLengthT1T2Eval()**
- **ProEdgeParamByLengthEval()**
- **ProCurveParamByLengthEval()**
- **ProGeomitemDistanceEval()**
- **ProGeomitemAngleEval()**
- **ProSurfaceDiameterEval()**
- **ProGeomitemDiameterEval()**
- **ProContourBoundingBox3dCompute()**
- **ProContourBoundingBox2dCompute()**

The function **ProSurfaceAreaEval()** evaluates the surface areas of a specified surface. It is not valid for datum planes.

Function **ProContourAreaEval()** outputs the inside surface area of a specified outer contour. Note it takes into account internal voids.

The function **ProSurfaceExtremesEval()** finds the coordinates of the face edges at the extremes in the specified direction. The accuracy of the result is limited to the accuracy of edge tessellation.

Function **ProSurfacePrincipalCrvtEval()** outputs the curvatures and directions of the specified surface at a given UV point. The u and v values are obtained by specifying the projection type as PRO_SRFTESS_NO_PROJECTION for the function **ProSurfacetessellationinputUvprojectionSet()**.

The function **ProEdgeLengthEval()** evaluates the length of a solid edge, and **ProCurveLengthEval()** does the same for a datum curve.

Function **ProEdgeLengthT1T2Eval()** finds the length of a specified edge between two parameters. Use function **ProCurveLengthT1T2Eval()** to do the same for a curve.

ProEdgeParamByLengthEval() finds the parameter value of the point located a given length from the specified start parameter. Use function **ProCurveParamByLengthEval()** to do the same for a curve.

The function **ProGeomitemDistanceEval()** measures the distance between two geometry items. The geometry items are expressed as *ProSelection* objects, so you can specify any two objects in an assembly. Each object can be an axis, plane surface, or datum point.

The function **ProGeomitemAngleEval()** measures the angle between two geometry items expressed as *ProSelection* objects. Both objects must be straight, solid edges.

The function **ProSurfaceDiameterEval()** measures the diameter of a surface, expressed as a *ProSelection* object. The surface type must be one of the following:

- Cylinder—The cylinder radius
- Torus—The distance from the axis to the generating arc
- Cone—The distance of the point specified from the axis
- Surface of revolution—The distance of the point specified from the axis

The *u* and *v* values are obtained by specifying the projection type as PRO_SRFTESS_NO_PROJECTION for the function **ProSurfacetessellationinputUvprojectionSet()**.

Note: In the case of a sphere made by revolving an arc, the Pro/ENGINEER command **Info, Measure, Diameter** gives the real spherical diameter, whereas **ProGeomitemDiameterEval()** gives the distance of the specified point from the axis of revolution.

The functions **ProContourBoundingBox2dCompute()** and **ProContourBoundingBox3dCompute()** output a bounding box for the inside surface of the specified outer contour.

Note: Only the 3D function takes into account internal voids.

Geometry as NURBS

Functions Introduced:

- **ProSurfaceToNURBS()**
- **ProEdgeToNURBS()**
- **ProCurveToNURBS()**

A common reason for extracting the solid geometry of a Pro/ENGINEER model is to pass it to another MCAE tool for some kind of engineering analysis. Not all of the other MCAE tools share the rich variety of geometry equation types supported by Pro/ENGINEER, and therefore may not be able to import all the surface descriptions directly. Because many MCAE systems use nonuniform rational B-splines (NURBS) to model surfaces and edges, you frequently need to convert many or all of the Pro/ENGINEER surface descriptions to NURB splines.

The function **ProSurfaceToNURBS()** operates on a surface of any type. The function makes an accurate approximation of the shape of the surface using a NURBS, and outputs a pointer to the structure *ProSurfacedata*. This structure contains the surface type PTC_B_SPLSRF, which describes the form of the NURBS.

The function **ProEdgeToNURBS()** finds a one-dimensional NURBS that approximates a Pro/ENGINEER solid edge. The function outputs a pointer to the *ProCurvedata* union whose *b_spline* field contains the NURBS description.

The function **ProCurveToNURBS()** provides the same functionality as **ProEdgeToNURBS()**, but for a datum curve.

Both *ProSurfacedata* and *ProCurvedata* are declared in the Pro/TOOLKIT header file *ProGeomitem.h*.

Interference

Functions Introduced:

- **pro_dist_manifolds()**
- **pro_compute_clearance()**
- **pro_compute_interference()**
- **pro_compute_global_interference()**
- **pro_compute_volume()**
- **pro_display_interf_volume()**
- **pro_interference_volume_release()**

You can check whether there is interference by using NULL for the interference volume argument to the functions **pro_compute_interference()** and **pro_compute_global_interference()**.

Example 4: Interference Checking for Assemblies and Parts

This example shows how to use the interference functions to compute global interference within an assembly, and the interference between two parts.

```
/*----- Pro/Develop Includes -----*/
#include      "prodevelop.h"
#include      "user_wchar_t.h"
#include      "proincludes.h"
#include      "select3d.h"
#include      "prographics.h"
#include      "user_error.h"
#include      "pro_interf.h"
/*----- Pro/Toolkit Includes -----*/
#include <ProToolkit.h>
#include "ProMdl.h"
#include "ProUtil.h"
#include <PTApplsUnicodeUtils.h>
/*----- Function Prototypes -----*/
ProError user_surf_clear ();
ProError user_dist_manifolds ();
ProError user_part_interference ();
ProError user_global_interference ();
int user_dump_interferences ();
int user_dump_mdl();
/*----- External Data -----*/
extern char *user_gen_filename();
```

```

/*=====*\
FUNCTION   : user_surf_clear
PURPOSE    : Computes the clearance between two surfaces.
\*=====*/

ProError user_surf_clear ()
{
    int status;
    int      interf;
    double   distance, coord[2][3];
    Select3d *sel;
    ProFileName msg_fil;
    ProStringToWstring( msg_fil, "msg_uggeom.txt" );
    while (1)
    {
        status = ProMessageDisplay (msg_fil, "USER Select a face");
        ERROR_CHECK("user_surf_clear","ProMessageDisplay",status);
        if (pro_select ("face", 2, &sel, 0, 0) <= 1)
            return (PRO_TK_GENERAL_ERROR);
/*-----*\
Computes the clearance between the two selected surfaces.
\*-----*/
        status = pro_compute_clearance (sel, PRO_SURFACE_CLEARANCE,
                                         &distance, &interf, coord);
        ERROR_CHECK("user_surf_clear","pro_compute_clearance",
                    (status <=0));
        status = ProMessageDisplay (msg_fil, "USER Distance is
                                         %0(5.3)f", &distance);
        ERROR_CHECK("user_surf_clear","ProMessageDisplay",status);
        if (interf)
        {
            status = ProMessageDisplay (msg_fil, "USER Interference
                                         exists");
            ERROR_CHECK("user_surf_clear","ProMessageDisplay",status);
        }
    }
    return (PRO_TK_NO_ERROR);
}
/*=====*\
FUNCTION   : user_dist_manifolds
PURPOSE    : Computes the critical distance between two objects.
\*=====*/

ProError user_dist_manifolds ()
{
    int      status, old_color;
    Select3d *sel;
    double   uvpar[2][2], xyz[2][3], distance;
    ProFileName msg_fil;

    ProStringToWstring( msg_fil, "msg_uggeom.txt" );

```



```

status = ProMessageDisplay (msg_fil, "USER %0s", "Select
                             two entities");
ERROR_CHECK("user_dist_manifolds","ProMessageDisplay",status);
if (pro_select ("point,edge,curve,surface", 2, &sel, 0, 0) < 2)
return (PRO_TK_GENERAL_ERROR);
/*-----*\
Compute the distance between the two manifolds.
\*-----*/
status = pro_dist_manifolds (&sel[0], &sel[1], uvpar, xyz,
                             &distance);
ERROR_CHECK("user_dist_manifolds","pro_dist_manifolds",
            (status <=0));
if (status)
{
    status = ProMessageDisplay (msg_fil, "USER Distance is
                                     %0(5.3)f", &distance);
    ERROR_CHECK("user_dist_manifolds","ProMessageDisplay",status);
}
else
{
    status = ProMessageDisplay (msg_fil, "USER %0s", "No min/max
                                     distances found");
    ERROR_CHECK("user_dist_manifolds","ProMessageDisplay",status);
}
return (PRO_TK_NO_ERROR);
}
/*=====*\
FUNCTION : user_part_interference
PURPOSE  : Computes the interference between two parts.
\*=====*/

ProError user_part_interference ()
{
    int status;
    Select3d *sel;
    Prohandle *intf_surfs;
    int n_surfs, i;
    wchar_t w_str[5];
    char str[100], a_str[5];
    double volume;
    FILE *fp;
    char fname[PRODEV_NAME_SIZE+4];
    wchar_t w_fname[PRODEV_NAME_SIZE+4];
    Prohandle p_model;
    ProFileName msg_fil;

    p_model = pro_get_current_object();
    ERROR_CHECK("user_part_interference","pro_get_current_object",
                (p_model == NULL));
    if (p_model == NULL) return (PRO_TK_E_NOT_FOUND);
    ProStringToWstring( msg_fil, "msg_uggeom.txt" );

```

```

    status = ProMessageDisplay (msg_fil, "USER %0s",
                                "Select two parts");
    ERROR_CHECK("user_part_interference", "ProMessageDisplay", status);
    if (pro_select ("part", 2, &sel, 0, 0) <= 1)
        return (PRO_TK_GENERAL_ERROR);
/*-----*\
Compute the interference between the two parts.
\*-----*/
    status = pro_compute_interference (sel, 0, &intf_surfs,
                                        &n_surfs);
    ERROR_CHECK("user_part_interference", "pro_compute_interference",
                (status != PRODEV_NO_ERROR));
    if (status != PRODEV_NO_ERROR)
        return (PRO_TK_GENERAL_ERROR);
    if (n_surfs == 0)
    {
        status = ProMessageDisplay (msg_fil, "USER %0s",
                                    "No interference was detected.");
        ERROR_CHECK("user_part_interference", "ProMessageDisplay",
                    status);
        return (PRO_TK_NO_ERROR);
    }

    ProTKSprintf (str, "%d interference surfaces found. Display?
                    [Y]:", n_surfs);
    status = ProMessageDisplay (msg_fil, "USER %0s", str);
    ERROR_CHECK("user_part_interference", "ProMessageDisplay",
                status);
    if (!ProMessageStringRead (4, w_str))
    {
        ProWstringToString (a_str, w_str);
        if (a_str[0] != 'y' || a_str[0] != 'Y')
            return (PRO_TK_NO_ERROR);
    }
    user_gen_filename (p_model, ".intf", fname);
    if ((fp = PTApplsUnicodeFopen (fname, "w")) == NULL)
        return (PRO_TK_GENERAL_ERROR);
    ProTKFprintf (fp, "Interference in ");
    user_dump_mdl (fp, p_model);
    ProTKFprintf (fp, "\n\n");
/*-----*\
Compute the interference volume.
\*-----*/
    status = pro_compute_volume (intf_surfs, n_surfs, &volume);
    ERROR_CHECK("user_part_interference", "pro_compute_volume",
                (status <= 0));
    user_dump_interferences (fp, sel, volume);
/*-----*\
Highlight the interference regions.
\*-----*/
    status = pro_display_interf_volume (intf_surfs, n_surfs,

```

```

                                EDGE_HIGHLIGHT_COLOR);
ERROR_CHECK("user_part_interference", "pro_display_interf_volume",
            (status <= 0));
ProTKSprintf (str, "Total Interference Volume = %lf", volume);
status = ProMessageDisplay (msg_fil, "USER %0s", str);
ERROR_CHECK("user_part_interference", "ProMessageDisplay", status);
/*-----*/
Release the memory for the interference volume.
/*-----*/
status = pro_interference_volume_release (intf_surfs, n_surfs);
ERROR_CHECK("user_part_interference",
            "pro_interference_volume_release",
            (status <= 0));

fclose (fp);
ProStringToWstring (w_fname, fname);
status = ProInfoWindowDisplay (w_fname, NULL, NULL);
ERROR_CHECK("user_part_interference", "ProInfoWindowDisplay",
            status);
return (PRO_TK_NO_ERROR);
}
/*=====*/
FUNCTION : user_global_interference
PURPOSE  : Computes the global interference within an assembly.
/*=====*/

ProError user_global_interference ()
{
    int      status;
    Select3d      *sel;
    Prohandle     **intf_part_surfs;
    int           n_intf_parts, *n_part_surfs, i;
    Interference_parts *intf_parts;
    wchar_t       w_str[5];
    char          str[100], a_str[5];
    double        volume;
    FILE          *fp;
    char          fname[PRODEV_NAME_SIZE+4];
    wchar_t       w_fname[PRODEV_NAME_SIZE+4];
    ProFileName   msg_fil;
    Prohandle     p_model;
    ProStringToWstring( msg_fil, "msg_uggeom.txt" );
    p_model = pro_get_current_object();
    ERROR_CHECK("user_global_interference", "pro_get_current_object",
                (p_model == NULL));
    if (p_model == NULL) return (PRO_TK_E_NOT_FOUND);
/*-----*/
Compute the interference within the assembly.
/*-----*/
status = pro_compute_global_interference (p_model, 0,
                                         &n_intf_parts,
                                         &intf_parts, &intf_part_surfs, &n_part_surfs);

```

```

ERROR_CHECK("user_global_interference",
            "pro_compute_global_interference",
            (status <=0));

if (status != PRODEV_NO_ERROR)
    return (PRO_TK_GENERAL_ERROR);

if (n_intf_parts == 0)
{
    status = ProMessageDisplay (msg_fil, "USER %0s",
                                "No interference was detected.");
    ERROR_CHECK("user_global_interference",
                "ProMessageDisplay",status);
}

ProTKSprintf (str, "%d interfering part pairs found. Display
                  interference pair volumes? [Y]:", n_intf_parts);
ProMessageDisplay (msg_fil, "USER %0s", str);
if (!ProMessageStringRead (4, w_str))
{
    ProWstringToString (a_str, w_str);
    if (a_str[0] != 'y' || a_str[0] != 'Y')
        return (PRO_TK_NO_ERROR);
}
user_gen_filename (p_model, ".intf", fname);
if ((fp = PTApplsUnicodeFopen (fname, "w")) == NULL)
    return (PRO_TK_GENERAL_ERROR);
ProTKFprintf (fp, "Interference in ");
user_dump_mdl (fp, p_model);
ProTKFprintf (fp, "\n\n");
for (i = 0; i < n_intf_parts; i++)
{
/*-----*\
Compute the interference volumes.
\*-----*/
    status = pro_compute_volume (intf_part_surfs[i],
                                n_part_surfs[i], &volume);
    ERROR_CHECK("user_global_interference", "pro_compute_volume",
                (status <=0));

    user_dump_interferences (fp, (Select3d *)&(intf_parts[i].part1_sel),
                             volume);
/*-----*\
Highlight the interference regions.
\*-----*/
    status = pro_display_intf_volume (intf_part_surfs[i],
                                      n_part_surfs[i],
                                      EDGE_HIGHLITE_COLOR);
    ERROR_CHECK("user_global_interference",
                "pro_display_intf_volume",
                (status <=0));
}

```

```

ProTKSprintf (str, "Interference pair %d of
                %d: Volume = %0.2lf:
                Continue? [y]:", i+1, n_intf_parts, volume);
status = ProMessageDisplay (msg_fil, "USER %0s", str);
ERROR_CHECK("user_global_interference",
            "ProMessageDisplay", status);

if (!ProMessageStringRead (4, w_str))
{
    ProWstringToString (a_str, w_str);
    if (a_str[0] != 'y' || a_str[0] != 'Y')
    {
        fclose (fp);
        status = pro_interference_volume_release(
            intf_part_surfs[i], n_part_surfs[i]);
        return (PRO_TK_NO_ERROR);
    }
}

/*-----*\
Release the memory for the interference volume.
\*-----*/
status = pro_interference_volume_release (intf_part_surfs[i],
                                           n_part_surfs[i]);
}
fclose (fp);
ProStringToWstring (w_fname, fname);
status = ProInfoWindowDisplay (w_fname, NULL, NULL);
ERROR_CHECK("user_global_interference", "ProInfoWindowDisplay",
            status);
return (PRO_TK_NO_ERROR);
}

/*=====*\
FUNCTION   : user_dump_interferences
PURPOSE    : Writes the interference information into a file.
\*=====*/

int user_dump_interferences (fp, intf, volume)
FILE          *fp;
Select3d      intf[];
double        volume;
{
    int m;
    ProTKFprintf (fp, "Interfering Parts:\n");
    ProTKFprintf (fp, "\t ");
    user_dump_md1 (fp, intf[0].part_ptr);
    ProTKFprintf (fp, "\tmemb_id_tab[");
    for (m = 0; m < intf[0].memb_num; m++)
    {
        ProTKFprintf (fp, "%d", intf[0].memb_id_tab[m]);
        if (m < intf[0].memb_num-1)
            ProTKFprintf (fp, ",");
    }
}

```

```

    }
    ProTKFprintf (fp, "]\n");
    ProTKFprintf (fp, "\t ");
    user_dump_mdl (fp, intf[1].part_ptr);
    ProTKFprintf (fp, "\tmemb_id_tab[");
    for (m = 0; m < intf[1].memb_num; m++)
    {
        ProTKFprintf (fp, "%d", intf[1].memb_id_tab[m]);
        if (m < intf[1].memb_num-1)
            ProTKFprintf (fp, ",");
    }
    ProTKFprintf (fp, "]\n");
    ProTKFprintf (fp, "\tInterfering Volume: %0.2lf\n\n", volume);
    return(0);
}

int user_dump_mdl (fp, p_model)
FILE          *fp;
Prohandle     p_model;
{
    ProMdlData      info;
    char           name[80], type[10];
    if (p_model == NULL)
        return (0);
    ProMdlDataGet (p_model, &info);
    ProTKFprintf (fp, "%s.%s:", ProWstringToString (name, info.name),
        ProWstringToString (type, info.type));
    return (0);
}

```

Faceted Geometry

Pro/ENGINEER allows you to build a surface CAD model on top of faceted or triangular data. Each facet is uniquely identified by a face normal and three vertices.

A facet is represented by the opaque handle *ProFacet*, while a set of facets is represented by the DHandle *ProFacetSet*. *ProFacetSet* has the same structure as the object *ProModelitem* and is defined as follows:

```

typedef struct pro_model_item
{
    ProType  type;
    int      id;
    ProMdl  owner;
} ProFacetSet;

```

Visiting Facets and Facet Sets

The functions described below enable you to find all the facet sets in a model and visit the facets stored in those sets.

Functions Introduced:

- **ProSolidFacetsetVisit()**
- **ProFacetsetFacetVisit()**

The function **ProSolidFacetsetVisit()** visits each of the facet sets in a given model. This function takes the filter function **ProFacetsetFilterAction()** and the visit function **ProFacetsetVisitAction()** as its input arguments. The function **ProFacetsetFilterAction()** is a generic action function for filtering the facet sets from a model. It returns the filter status of the facet sets. This status is used as an input argument by the visit action function **ProFacetsetVisitAction()**.

The function **ProFacetsetFacetVisit()** visits the facets in the faceted geometry set. This function takes the filter function **ProFacetFilterAction()** and the visit function **ProFacetVisitAction()** as its input arguments. The function **ProFacetFilterAction()** is a generic action function for filtering a facet from the faceted geometry set. It returns the filter status of the facet. This status is used as the input argument by the visit action function **ProFacetVisitAction()**.

Accessing Facet Properties

The functions described below allow you to access the face normal and three vertices or corners of a facet. A facet vertex is given by the object *ProFacetVertex*.

Functions Introduced:

- **ProFacetVerticesGet()**
- **ProFacetverticesFree()**
- **ProFacetvertexPointGet()**
- **ProFacetNormalGet()**

The function **ProFacetVerticesGet()** obtains the vertices of a specified facet.

Use the function **ProFacetverticesFree()** to release the memory allocated for the vertices of a facet.

The function **ProFacetvertexPointGet()** returns the location of the facet vertex in the model coordinate system.

The function **ProFacetNormalGet()** returns the normal vector of a facet in the model coordinate system.

6

Core: Relations

This chapter describes how to access relations on all models and model items in Pro/ENGINEER using the functions provided in Pro/TOOLKIT.

Topic	Page
Relations	6 - 2
Adding a Customized Function to the Relations Dialog in Pro/ENGINEER	6 - 4

Relations

Functions Introduced:

- **ProModelitemToRelset()**
- **ProSolidRelsetVisit()**
- **ProRelsetToModelitem()**
- **ProRelsetRegenerate()**
- **ProRelsetCreate()**
- **ProRelsetDelete()**
- **ProRelsetRelationsGet()**
- **ProRelsetRelationsSet()**
- **ProRelationEval()**

The object *ProRelset* represents the entire set of relations on any model or model item. It is an opaque handle whose contents can be accessed only through the functions described in this section.

Pro/TOOLKIT can only access the relations of the models and model item types as listed in the table below:

Model Types	Description
PRO_PART	Part
PRO_ASSEMBLY	Assembly
PRO_DRAWING	Drawing
PRO_REPORT	Report
PRO_DIAGRAM	Diagram
PRO_DWGFORM	Format
PRO_UDF	User-defined feature
PRO_FEATURE	Feature
PRO_SURFACE	Surface
PRO_EDGE	Edge
PRO_WELD_PARAMS	Weld parameters
PRO_BND_TABLE	Bend table
PRO_EXTOBJ	External objects
PRO_PATREL_FIRST_DIR	Pattern direction 1
PRO_PATREL_SECOND_DIR	Pattern direction 2

Model Types	Description
PRO_RELOBJ_QUILT	Quilt
PRO_RELOBJ_CRV	Curve
PRO_RELOBJ_COMP_CRV	Compound curve
PRO_RELOBJ_ANNOT_ELEM	Annotation Element
PRO_RELOBJ_NC_STEP_OBJECT	NC Step Table Entry
PRO_RELOBJ_NC_STEP_MODEL	NC Step Table Model

The function **ProModelitemToRelset()** outputs a *ProRelset* object that contains the set of relations owned by the given model item. (Note that not all model items can have relations sets associated with them—only the types listed in the table.)

To get the relations of a feature pattern, the model item type should be either PRO_PATREL_FIRST_DIR or PRO_PATREL_SECOND_DIR, and the identifier should be that of the dimension on the pattern leader that drives the pattern in that direction. To find the identifiers of the pattern dimension, use the functions described in the section Manipulating Patterns.

The function **ProSolidRelsetVisit()** enables you to visit all the relation sets on every model item in a model. Like other visit functions, it calls a user-supplied action function for each relation set, although there is no filter function. If the user-supplied function returns any status other than PRO_TK_NO_ERROR, visiting will stop.

The function **ProRelsetToModelitem()** outputs the model item that is the owner of the specified *ProRelset*.

You can regenerate a relation set using the function **ProRelsetRegenerate()**. This function also determines whether the specified relation set is valid. If an error occurred, the function returns a status other than PRO_TK_NO_ERROR.

To create a new relation set for a model item, use the function **ProRelsetCreate()**. If a relation set already exists for that item, the function returns PRO_TK_E_FOUND.

To delete all the relations in a specified relation set, call the function **ProRelsetDelete()**.

The function **ProRelsetRelationsGet()** extracts the text of a set of relations described by a *ProRelset* object. This function takes two arguments: the *ProRelset* for the relation set and a preallocated expandable array. The elements of the expandable array are of type *ProLine* (wide strings).

The function **ProRelsetRelationsSet()** creates a *ProRelset* object from an expandable array of *ProLine* objects that describes the relations as text. For details of the syntax and use of relations, see the *Introduction to ProENGINEER*.

Note: Existing relations will be overwritten by a call to **ProRelsetRelationsSet()**.

The function **ProRelationEval()** evaluates a line of a relation set and outputs the resulting value in the form of a *ProParamvalue* structure. (See the chapter ‘Core: Parameters’ for a description of this data structure.) The use of special pattern relation symbols such as *memb_v* or *idx1* is not supported; instead, replace these symbols with the corresponding dimension value or number, and evaluate them individually.

Adding a Customized Function to the Relations Dialog in Pro/ENGINEER

Functions Introduced:

- **ProRelationFunctionRegister()**
- **ProRelationReadFunction()**
- **ProRelationWriteFunction()**
- **ProRelationArgscheckFunction()**

The function **ProRelationFunctionRegister()** registers a custom function that is visible to users in the Pro/ENGINEER relations dialog. To register a custom function you may supply the following:

- An array of expected arguments. The arguments are described by their type (double, integer, etc) and attributes indicating if the argument can be skipped when the user calls the relations function. These optional arguments must fall at the end of the argument list.

- A Boolean indicating whether or not to check argument types internally. If the Boolean is set not to check the argument types internally, Pro/ENGINEER does not need to know the contents of the arguments array. The custom function you create must handle all the user errors in this situation.
- An arguments check function, which can be used to verify the input arguments.
- A read function, which provides the value of the function when used in the right-hand side of a relation. For example

```
d12 = ptk_user_function (0.5, 5, true, inch)
```

- A write function, which receives the value when the function is used in the left-hand side of the relation. For example:

```
ptk_user_function (assigned_value) = 14.0;
```

- All the callback functions are optional and may be NULL.

Note: Pro/TOOLKIT registered relations are valid only when the function has been registered by the application. If the application is not running or not present, models that contain user-defined relations cannot evaluate these relations. In this situation, the relations are marked as errors, however, they can be commented until needed at a later time when the relation functions are reactivated.

The function type **ProRelationReadFunction()** is called when a custom relation function is used on the right-hand side of the relation. You should output the computed value of the custom relation function which is used to complete evaluation of the relation.

The function **ProRelationWriteFunction()**, is called when a custom function is used on the left hand side of a relation. The value of the right-hand side of the relation is provided as an input. You can use this type of function to initialize properties to be stored and used by your Pro/TOOLKIT application.

The function **ProRelationArgscheckFunction()** performs the argument check function for a custom relation function. Return PRO_TK_NO_ERROR to indicate that the arguments passed to the relation are valid. Any other error causes an error to be displayed in the Relations dialog.

The following example creates three externally defined functions to be available in the relations dialog:

ptk_set_a and ptk_set_b are used from the left-hand side of relations to initialize double values to stored "A" and "B" variables.

ptk_eval_ax_plus_b returns the computation of $Ax+B$, where x is an input to the function.

These functions are available only while the Pro/TOOLKIT application is running. Models can be saved with relations referencing these functions, but these models will have relation errors if retrieved while the application is not running.

```
#include <ProToolkit.h>
#include <ProRelSet.h>
#include <ProParamval.h>

#define PTK_SET_A "ptk_set_a"
#define PTK_SET_B "ptk_set_b"

static double stored_a;
static double stored_b;

/*=====*\
  FUNCTION: UserRelFPropertySet
  PURPOSE: "Write" function for the assignment of the parameters used
  for calculation.
\*=====*/
ProError UserRelFPropertySet (ProRelset* relset, ProMdl mdl,
char* ext_func_name, ProParamvalue* args,
ProParamvalue* input, ProAppData data)
{
  if (input->type != PRO_PARAM_DOUBLE)
    return (PRO_TK_GENERAL_ERROR);

  if (strcmp (ext_func_name, PTK_SET_A) == 0)
  {
    stored_a = input->value.d_val;
    return (PRO_TK_NO_ERROR);
  }

  if (strcmp (ext_func_name, PTK_SET_B) == 0)
  {
    stored_b = input->value.d_val;
    return (PRO_TK_NO_ERROR);
  }

  return (PRO_TK_GENERAL_ERROR);
}
```

```

/*=====*\
  FUNCTION: UserRelFGetAxPlusB
  PURPOSE:  "Read" function for the calculation of the values.
/*=====*\
ProError UserRelFGetAxPlusB (ProRelset* relset, ProMdl mdl,
char* ext_func_name,  ProParamvalue* args,
ProAppData data, ProParamvalue* result)
{

    double x = args [0].value.d_val;

    result->type = PRO_PARAM_DOUBLE;
    result->value.d_val = stored_a * x + stored_b;

    return (PRO_TK_NO_ERROR);
}

/*=====*\
  FUNCTION: UserCustomRelFunctionsDefine
  PURPOSE:  Define the functions that can be used by Pro/E users in
  relations.
/*=====*\
int UserCustomRelFunctionsDefine ()
{
    ProRelfuncArg* args_array;

    stored_a = 5.5;
    stored_b = 13.1;

/*-----*\
    Allocate the array of expected arguments and types for the get function.
/*-----*\
ProArrayAlloc (1, sizeof (ProRelfuncArg), 1, (ProArray*)&args_array);
args_array [0].type = PRO_PARAM_DOUBLE;
args_array [0].attributes = PRO_RELF_ATTR_NONE;

/*-----*\
    Register the set functions.  These functions should be used only on the
    LHS of relations, and they do not require or use any arguments.
/*-----*\
ProRelationFunctionRegister ("ptk_set_a", NULL,
NULL, UserRelFPropertySet, NULL,
PRO_B_TRUE, NULL);

ProRelationFunctionRegister ("ptk_set_b", NULL,
NULL, UserRelFPropertySet, NULL,
PRO_B_TRUE, NULL);

/*-----*\

```

```

Register the computation function. This function should be used only on
the RHS of relations. Type-checking is used to ensure that exactly one
double argument was passed.
\*-----*/
    ProRelationFunctionRegister ("ptk_eval_ax_plus_b", args_array,
        UserRelFGetAxPlusB, NULL, NULL,
        PRO_B_FALSE, NULL);

    return (PRO_TK_NO_ERROR);
}

#undef PTK_SET_A
#undef PTK_SET_B

```


7

Core: Parameters

This chapter describes the Pro/TOOLKIT functions that give access to parameters and geometric tolerances.

Topic	Page
Parameter Objects	7 - 2
Parameter Values	7 - 3
Accessing Parameters	7 - 4
Designating Parameters for Pro/INTRALINK or Windchill	
PDMLink	7 - 10
Restricted Parameters	7 - 10
Table-Restricted Parameters	7 - 11
Driven Parameters	7 - 13

Parameter Objects

The object *ProParameter* describes the contents and ownership of a parameter. *ProParameter* is a DHandle whose declaration is as follows:

```
typedef struct proparameter
{
    ProType      type;
    ProName      id;
    ProParamowner owner;
} ProParameter;

typedef struct proparamowner
{
    ProParamfrom  type;
    union
    {
        ProModelitem item;
        ProMdl      model;
    } who;
} ProParamowner;

typedef enum proparamfrom
{
    PRM_MODEL,
    PRM_ITEM
} ProParamfrom;
```

A structure called *ProParamvalue* is used to represent the value of a parameter. Its declaration is as follows:

```
typedef struct Pro_Param_Value
{
    ProParamvalueType  type;
    ProParamvalueValue value;
```

```

} ProParamvalue;

typedef enum param_value_types
{
    PRO_PARAM_DOUBLE,
    PRO_PARAM_STRING,
    PRO_PARAM_INTEGER,
    PRO_PARAM_BOOLEAN,
    PRO_PARAM_NOTE_ID,
    PRO_PARAM_VOID
} ProParamvalueType;

typedef union param_value_values
{
    double    d_val;
    int       i_val;
    short     l_val;
    ProLine   s_val;
} ProParamvalueValue;

```

Parameter Values

Functions introduced:

- **ProParamvalueSet()**
- **ProParamvalueValueGet()**
- **ProParamvalueTypeGet()**

These three functions are utilities to help you manipulate the *ProParamvalue* structure. They do not directly affect any parameter in Pro/ENGINEER.

The function **ProParamvalueSet()** sets the value type of a *ProParamvalue* structure, and writes a value of that type to the object.

The function **ProParamvalueTypeGet()** provides the type of a *ProParamvalue* object.

The function **ProParamvalueValueGet()** reads a value of the specified type from a *ProParamvalue* structure.

Accessing Parameters

Functions introduced:

- **ProParameterInit()**
- **ProParameterValueGet()**
- **ProParameterValueSet()**
- **ProParameterIsModified()**
- **ProParameterValueReset()**
- **ProParameterCreate()**
- **ProParameterDelete()**
- **ProParameterSelect()**
- **ProParameterTableExport()**
- **ProParameterVisit()**
- **ProParameterReorder()**
- **ProParameterToFamtableItem()**
- **ProParameterUnitsGet()**
- **ProParameterWithUnitsCreate()**
- **ProParameterScaledvalueGet()**
- **ProParameterScaledvalueSet()**
- **ProParameterDescriptionGet()**
- **ProParameterDescriptionSet()**

The function **ProParameterInit()** initializes a *ProParameter* object by defining its name and owner. The owner is expressed in terms of a *ProModelitem* object, and can be a Pro/ENGINEER model, feature, surface, or edge.

If the owner is a model, use **ProMdlToModelitem()** to create the *ProModelitem* object; in other cases, use **ProModelitemInit()**.

The function **ProParameterValueGet()** reads the value of a parameter specified by a *ProParameter* object into a *ProParamvalue* object provided by the application.

The function **ProParameterValueSet()** sets the value of a Pro/ENGINEER parameter identified by a *ProParameter* object to a value specified in a *ProParamvalue* structure.

The **ProParameterIsModified()** function returns a boolean value that indicates whether the value of the specified parameter has been modified since the last successful regeneration of the parameter owner.

The function **ProParameterValueReset()** sets the value of a parameter to the one it had at the end of the last regeneration.

The function **ProParameterCreate()** adds a new parameter to the Pro/ENGINEER database, and returns a valid *ProParameter* object for the new parameter. This function takes input arguments such as the *ProModelitem* object for the owner, the name, and the *ProParamvalue* structure for the value.

Note: Model items must have a name before you create a parameter for them. Geometric items such as surfaces, edges, curves, and quilts are not named by default; use **ProModelitemNameSet()** on these items before attempting to add parameters to them.

The function **ProParameterDelete()** deletes a parameter, specified by a *ProParameter* object, from the Pro/ENGINEER database.

The function **ProParameterSelect()** allows the user to select one or more parameters of a specified model or database item from the **Parameters** dialog box in Pro/ENGINEER. The top model from which the parameters will be selected must be displayed in the current window. The input argument *context* allows you to select parameters by context. It takes the following values:

- PRO_PARAMSELECT_ANY—Specifies any parameter.
- PRO_PARAMSELECT_MODEL—Specifies the parameters of the top-level model.
- PRO_PARAMSELECT_PART—Specifies the parameters of any part.
- PRO_PARAMSELECT_ASM—Specifies the parameters of any assembly.
- PRO_PARAMSELECT_FEATURE—Specifies the parameters of any feature.
- PRO_PARAMSELECT_EDGE—Specifies the parameters of any edge.

- **PRO_PARAMSELECT_SURFACE**—Specifies the parameters of any surface.
- **PRO_PARAMSELECT_QUILT**—Specifies the parameters of any quilt.
- **PRO_PARAMSELECT_CURVE**—Specifies the parameters of any curve.
- **PRO_PARAMSELECT_COMPOSITE_CURVE**—Specifies the parameters of any composite curve.
- **PRO_PARAMSELECT_INHERITED**—Specifies the parameters of any inheritance feature.
- **PRO_PARAMSELECT_SKELETON**—Specifies the parameters of any skeleton.
- **PRO_PARAMSELECT_COMPONENT**—Specifies the parameters of any component.
- **PRO_PARAMSELECT_ALLOW_SUBITEM_SELECTION**—Specifies the parameters of all the subitems of the top model.

Note: The signature of **ProParameterSelect()** has changed from Pro/ENGINEER Wildfire 2.0 onward.

The function **ProParameterTableExport()** exports a file containing information from a parameter table in Pro/ENGINEER in the CSV or TXT format. The output file contains the columns specified by the input argument *column_list* (bitmask of potential columns), if the output type is CSV. However, if the output type is TXT, then a default set of columns is exported.

The function **ProParameterVisit()** visits all the parameters on a specified database item.

Note:

- The parameters are returned in the order that they appear in the parameter dialog box for the database item.
- **ProParameterVisit()** does not visit mass property parameters.

The function **ProParameterReorder()** reorders the given parameter to come just after the indicated parameter.

The function **ProParameterToFamtableItem()** converts *ProParameter* objects to *ProFamtableItem* objects. You may need to call **ProParameterToFamtableItem()** after calling **ProParameterSelect()** that allows you to select parameters from a menu.

The function **ProParameterUnitsGet()** fetches the units assigned to a parameter.

The function **ProParameterWithUnitsCreate()** enables the creation of a new parameter with the assigned units.

The function **ProParameterScaledvalueGet()** retrieves the parameter value in terms of the units of the parameter, instead of the units of the owner model.

The function **ProParameterScaledvalueSet()** sets the parameter value in terms of the units provided, instead of using the units of the owner model.

The function **ProParameterDescriptionGet()** obtains the description of the parameter. The function **ProParameterDescriptionSet()** assigns the description of the parameter.

Notification Functions

The parameter notification functions support the parameters owned by an annotation element. These functions are call back functions and are accessible by calling the function **ProNotificationSet()**.

- **ProParameterCreatePreAction()**
- **ProParameterDeletePreAction()**
- **ProParameterModifyPreAction()**
- **ProParameterCreatePostAction()**
- **ProParameterModifyPostAction()**
- **ProParameterDeletePostAction()**

The notification function **ProParameterCreatePreAction()** is called before the parameter is created. This function is available by calling **ProNotificationSet()** with the value of the notify type as **PRO_PARAM_CREATE_PRE**.

The notification function **ProParameterDeletePreAction()** is called before the parameter is deleted. This function is available by calling **ProNotificationSet()** with the value of the notify type as **PRO_PARAM_DELETE_PRE**.

The notification function **ProParameterModifyPreAction()** is called before a parameter is modified. This function is available by calling **ProNotificationSet()** with the value of the notify type as **PRO_PARAM_MODIFY_PRE**.

You can use the **PreAction** functions to cancel any changes made to the parameters. If any value except PRO_TK_NO_ERROR is returned, then the change is not permitted. The application must provide appropriate messaging to the user to explain the reason for which the change was rejected.

Note: You are not permitted to cancel the parameter modification events when modifying multiple parameters as a group.

The notification function **ProParameterCreatePostAction()** is called after a parameter has been created. This function is available by calling **ProNotificationSet()** with the value of the notify type as PRO_PARAM_CREATE_POST.

The notification function **ProParameterModifyPostAction()** is called after a parameter has been modified. This function is available by calling **ProNotificationSet()** with the value of the notify type as PRO_PARAM_MODIFY_POST.

The notification function **ProParameterDeletePostAction()** is called after a parameter has been deleted. This function is available by calling **ProNotificationSet()** with the value of the notify type as PRO_PARAM_DELETE_POST.

Example 1: Labeling a Feature with a String Parameter

This example labels the selected feature with the specified string parameter. The program calls **ProParameterInit()** to find out whether the parameter already exists. If the parameter does not exist, the function calls **ProParameterCreate()**; if it exists, the function calls **ProParameterValueSet()**.

```
/*=====*\
FUNCTION: UserLabelFeature
PURPOSE: Label a feature with a string parameter.
\*=====*/
int UserLabelFeature()
{
    ProModelitem    feature;
    ProSelection    *sel;
    ProParameter    param;
    ProParamvalue   value;
    ProFileName     msgfil;
    ProName         name;
    ProLine         line;
    ProError        err;
    int             nsel;

    /*-----*\
        Set up the name of the message file.
```



```

\*-----*/
    ProStringToWstring (msgfil, "msg_ug12.txt");
/*-----*\
    Select a feature to label. If nothing is selected, exit.
\*-----*/
    err = ProMessageDisplay (msgfil, "USER Select a feature to
        label");

    err = ProSelect ("feature", 1, NULL, NULL, NULL, NULL, &sel,
        &nssel);

    if (nssel <= 0)
        return (0);
/*-----*\
    Get the model item from the selection.
\*-----*/
    err = ProSelectionModelitemGet (sel[0], &feature);
/*-----*\
    Get the name of the parameter.
\*-----*/
    err = ProMessageDisplay (msgfil, "USER %0s", "Enter the
        name of the parameter: ");
    err = ProMessageStringRead (PRO_NAME_SIZE, name);
/*-----*\
    Set the parameter type and value.
\*-----*/
    value.type = PRO_PARAM_STRING;

    err = ProMessageDisplay (msgfil, "USER %0s", "Enter the
        parameter string line: ");
    err = ProMessageStringRead (PRO_LINE_SIZE, value.value.s_val);
/*-----*\
    If the parameter exists, set its new value. Otherwise,
    create it.
\*-----*/
    err = ProParameterInit (&feature, name, &param);

    if (err == PRO_TK_E_NOT_FOUND)
    {
        err = ProParameterCreate (&feature, name, &value, &param);
    }
    else
    {
        err = ProParameterValueSet (&param, &value);
    }

    return (PRO_TK_NO_ERROR);
}

```

Designating Parameters for Pro/INTRALINK or Windchill PDMLink

Functions introduced:

- **ProParameterDesignationAdd()**
- **ProParameterDesignationVerify()**
- **ProParameterDesignationRemove()**

These functions control the designation of parameters for Pro/INTRALINK and Windchill PDMLink. A designated parameter will become visible within these management systems as an attribute on the owning model when it is next submitted.

The function **ProParameterDesignationAdd()** designates an existing parameter, referred to by its *ProParameter* object.

The function **ProParameterDesignationRemove()** removes the designation.

Note: PTC does not recommend undesignating or deleting a parameter that is already exposed to a Product Data Management system such as Windchill PDMLink or Pro/INTRALINK.

The function **ProParameterDesignationVerify()** tells you whether a specified model parameter is currently designated or not.

Restricted Parameters

Restricted parameters can have either:

- an enumeration which limits values only to those specified in advance; or
- a range which limits numerical values to fall within that range.

Functions introduced:

- **ProParameterIsEnumerated()**
- **ProParameterRangeGet()**

Use the function **ProParameterIsEnumerated()** to identify if a parameter is enumerated, and provides the values that may be assigned to it. If the parameter is enumerated then it gives the *ProArray* of values that are assigned to this parameter.

Use the function **ProParameterRangeGet()** to identify if a parameter's value is restricted to a certain range. If the parameter is restricted by a range then it gives the maximum and minimum value of the parameter.

Table-Restricted Parameters

A parameter table is made up of one or more parameter table sets. Each set represents one or more parameters with their assigned values or assigned ranges. A given parameter owner (model, feature, annotation element, geometry item) can only have one parameter table set applied that creates a given parameter. In Pro/TOOLKIT, a parameter table set is represented by the type *ProParamtableSet* and is made up of entries, represented by *ProParamtableEntry*. A single entry represents a parameter with an assigned value or range.

Functions introduced:

- **ProParameterTablesetGet()**
- **ProMdlParamtablesetsCollect()**
- **ProParamtablesetEntriesGet()**
- **ProParamtablesetTablepathGet()**
- **ProParamtablesetLabelGet()**
- **ProParamtablesetFree()**
- **ProParamtablesetProarrayFree()**
- **ProParamtableentryValueGet()**
- **ProParamtableentryRangeGet()**
- **ProParamtableentryNameGet()**
- **ProParamtableentryProarrayFree()**
- **ProParamtablesetApply()**
- **ProRelsetConstraintsGet()**

Use the function **ProParameterTablesetGet()** to obtain the governing parameter table set for this parameter, if it's a member of a set.

Use the function **ProMdlParamtablesetsCollect()** to obtain an array of all the table sets that are available for use in the given model. This includes all sets that are loaded from table files that are set up in this session, and any other sets that were previously stored in the model.

Use the function **ProParamtablesetFree()** to free the parameter table set and the function **ProParamtablesetProarrayFree()** to free an array of parameter table sets.

Use the function **ProParamtablesetEntriesGet()** to obtain the entries that are contained in a parameter table set.

Use the function **ProParamtablesetTablepathGet()** to obtain the name of the table that owns this parameter table set. If the set is loaded from a certain table file, this is the full path. If the set has been stored in the model directly, this is the table name.

Use the function **ProParamtablesetLabelGet()** to obtain the set label for a given parameter table set.

Use the function **ProParamtableentryNameGet()** to obtain the name for the parameter in this table set.

Use the function **ProParamtableentryValueGet()** to obtain the value for the parameter in this table set. If the parameter also has a range applied, this is the default value for the parameter. Use the function **ProParamtableentryRangeGet()** to obtain the permitted range for the parameter in this table set. The output arguments for this function are:

- *minimum*—The minimum value for this parameter as set by the parameter set.
- *maximum*—The maximum value for this parameter as set by the parameter set

You can use the function **ProParamtableentryProarrayFree()** to free an array of table entries.

The function **ProParamtablesetApply()** assigns this parameter set to the given parameter owner. Parameters used by the set are created or modified, as appropriate. The parameter values are set to the default values. This function does not regenerate the model and may fail if the parameter owner already has one or more of the set's required parameters defined which are not driven by this table.

The function **ProRelsetConstraintsGet()** obtains the constraints applied to a given relation set. Constraints may be assigned when one or more parameters of the set is governed by an external parameter file.

Driven Parameters

The functions described below provide access to the item (parameter or function) driving model parameters.

Functions Introduced:

- **ProParameterDriverGet()**
- **ProParameterDrivingparamGet()**
- **ProParameterDrivingparamSet()**
- **ProParameterDrivingFunctionGet()**
- **ProParameterDrivingFunctionSet()**

The function **ProParameterDriverGet()** obtains the type of operation driving a model parameter in the form of the *ProParameterDriver* object. The types of drivers are as follows:

- **PRO_PARAMDRIVER_PARAM**—Specifies the parameter driving the model parameter.
- **PRO_PARAMDRIVER_FUNCTION**—Specifies the function driving the model parameter.
- **PRO_PARAMDRIVER_RELATION**—Specifies the relation driving the model parameter.

The function **ProParameterDrivingparamGet()** obtains the driving parameter for a material parameter, if the driver type is **PRO_PARAMDRIVER_PARAM**.

The function **ProParameterDrivingparamSet()** sets the driver type for a material parameter to the value **PRO_PARAMDRIVER_PARAM**.

The function **ProParameterDrivingFunctionGet()** obtains the driving function for a material parameter, if the driver type is **PRO_PARAMDRIVER_FUNCTION**.

The function **ProParameterDrivingFunctionSet()** sets the driver type for a material parameter to the value **PRO_PARAMDRIVER_FUNCTION**.

8

Core: Coordinate Systems and Transformations

This chapter describes the various coordinate systems used by Pro/ENGINEER and Pro/TOOLKIT, and how to transform coordinates from one to another.

Topic	Page
Coordinate Systems	8 - 2
Coordinate System Transformations	8 - 4

Coordinate Systems

Pro/ENGINEER and Pro/TOOLKIT use the following coordinate systems:

- Solid coordinate system
- Screen coordinate system
- Window coordinate system
- Drawing coordinate system
- Drawing view coordinate system
- Assembly coordinate system
- Datum coordinate system
- Section coordinate system

The following sections describe each of these coordinate systems.

Solid Coordinate System

The solid coordinate system is the three-dimensional, Cartesian coordinate system used to describe the geometry of a Pro/ENGINEER solid model. In a part, the solid coordinate system describes the geometry of the surfaces and edges. In an assembly, the solid coordinate system also describes the locations and orientations of the assembly members.

You can visualize the solid coordinate system in Pro/ENGINEER by creating a coordinate system datum with the option **Default**. Distances measured in solid coordinates correspond to the values of dimensions as seen by the Pro/ENGINEER user.

Solid coordinates are used by Pro/TOOLKIT for all the functions that look at geometry, and most of the functions that draw three-dimensional graphics.

Screen Coordinate System

The screen coordinate system is a two-dimensional coordinate system that describes locations in a Pro/ENGINEER window. When the user zooms or pans the view, the screen coordinate system follows the display of the solid, so a particular point on the solid always maps to the same screen coordinate. The mapping changes only when the view orientation is changed.

Screen coordinates are a kind of nominal pixel count, so the bottom, left corner of the default window is at (0, 0) and the top, right corner is at (1000, 864).

Screen coordinates are used by some of the graphics functions, the mouse input functions, and all the functions that draw graphics or manipulate items on a drawing.

Window Coordinate System

The window coordinate system is similar to the screen coordinate system, except it is not affected by zoom and pan. When an object is first displayed in a window, or the option **View, Pan/Zoom, Reset** is used, the screen and window coordinates are the same.

Window coordinates are needed only if you need to take account of zoom and pan—for example, to find out whether a point on the solid is visible in the window, or to draw two-dimensional text in a particular window location, regardless of pan and zoom.

Drawing Coordinate System

The drawing coordinate system is a two-dimensional system that describes the location on a drawing relative to the bottom, left corner, and measured in drawing units. For example, on a U.S. letter-sized, landscape-format drawing sheet that uses inches, the top, right corner is (11, 8.5) in drawing coordinates.

The Pro/TOOLKIT functions that manipulate drawings generally use screen coordinates.

Drawing View Coordinate System

This drawing view coordinate system is used to describe the locations of entities in a drawing view.

Assembly Coordinate System

An assembly has its own coordinate system that describes the positions and orientations of the member parts and subassemblies, and the geometry of datum features created in the assembly.

When an assembly is retrieved into memory, each member is loaded too, and continues to use its own solid coordinate system to describe its geometry.

This is important when you are analyzing the geometry of a subassembly, and want to extract or display the results relative to the coordinate system of the parent assembly.

Datum Coordinate System

A coordinate system datum can be created anywhere in any part or assembly, and represents a user-defined coordinate system. It is often a requirement in a Pro/TOOLKIT application to describe geometry relative to such a datum.

Section Coordinate System

Every sketch has a coordinate system used to locate entities in that sketch. Sketches used in features will use a coordinate system different from that of the solid model.

Coordinate System Transformations

Functions Introduced:

- **ProPntTrfEval()**
- **ProVectorTrfEval()**

All coordinate systems are treated in Pro/TOOLKIT as if they were three-dimensional. Therefore, a point in any of the coordinate systems described is always represented in C by the following type:

```
typedef double ProPoint3d[3]
```

Vectors are distinguished for clarity by a different, though equivalent, declaration:

```
typedef double ProVector[3]
```

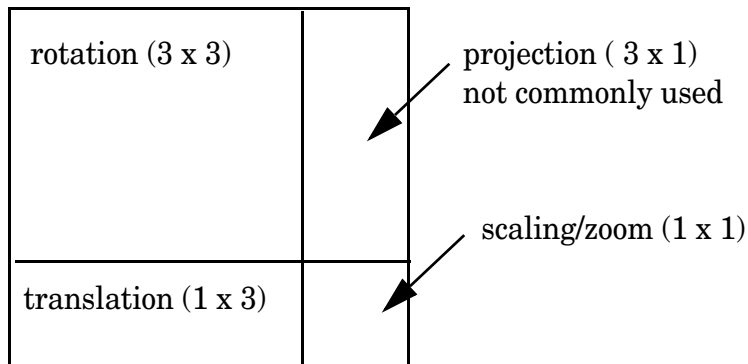
Screen, window and section coordinates contain a Z value whose positive direction is normal to the screen or the sketch. The value of Z is not generally important when specifying a screen location as an input to a function, but it is useful in other situations. For example, if the user selects a datum plane, you can find out which side is towards the user by calculating the normal to the plane, transforming to screen coordinates, then looking at the sign of the Z coordinate.

A transformation between two coordinate systems is represented by a 4x4 matrix, with the following type:

```
typedef double ProMatrix[4][4];
```

This combines the conventional 3x3 matrix that describes the relative orientation of the two systems, and the vector that describes the shift between them.

Figure 8-1: Transformation Matrix



Pro/TOOLKIT provides two utilities for performing coordinate transformations. The function **ProPntTrfEval()** transforms a three-dimensional point, and **ProVectorTrfEval()** transforms a three-dimensional vector.

The source code for other utilities that manipulate transformation matrices is located in the file
 <TK_LOADPOINT>/protk_appls/pt_examples/pt_utils/UtilMatrix.c

The following sections describe the functions needed to obtain the transformation matrix between two different coordinate systems in Pro/ENGINEER.

Transforming Solid to Screen Coordinates

Functions Introduced:

- **ProViewMatrixGet()**
- **ProViewMatrixSet()**

The view matrix describes the transformation from solid to screen coordinates. The function **ProViewMatrixGet()** provides the view matrix for the specified view. Example 1: Solid Coordinates to Screen Coordinates shows a function that transforms a point, using the **ProViewMatrixGet()** function, and an example user function.

The function **ProViewMatrixSet()** changes the orientation of the solid model on the screen.

Example 1: Solid Coordinates to Screen Coordinates

```
/*----- Pro/Toolkit Includes -----*/
#include <ProToolkit.h>
#include <ProAsmcomppath.h>
#include <ProObjects.h>
#include <ProView.h>

/*----- Application Includes -----*/
#include <TestError.h>

/*----- Function Prototypes -----*/
void UserSolidToScreen();

void UserSolidToScreen(
    ProSolid    solid,
    ProPoint3d  solid_point,
    ProPoint3d  screen_point)
{
    ProMatrix  view_matrix;
    ProError   status;

    status = ProViewMatrixGet(solid,NULL,view_matrix);
    ERROR_CHECK("UserSolidToScreen", "ProViewMatrixGet()", status);
    status = ProPntTrfEval(solid_point,view_matrix,screen_point);
    ERROR_CHECK("UserSolidToScreen", "ProPntTrfEval()", status);
}
```

Example 2: Transform from Solid Coordinates to Screen Coordinates

This example shows how to extract a selected point from the surface of a solid model, transform it from solid coordinates to screen coordinates, write the results into a file and display it.

```
/*----- Pro/Toolkit Includes -----*/
#include <ProToolkit.h>
#include <ProMdl.h>
#include <ProMessage.h>
#include <ProGraphic.h>
#include <ProWchar.h>
#include <ProUtil.h>
```

```
#include <PTApplsUnicodeUtils.h>
/*----- Pro/Develop Includes -----*/
#include <prodevelop.h>

/*----- Application Includes -----*/
#include <TestError.h>

/*----- Function Prototypes -----*/
ProError UserScreen2DrwExample();
ProError UserSolid2ScreenExample();

/*----- External Data -----*/
extern void UserScreenToDwg();
extern void UserSolidToScreen();
extern int UserMakeSelections();

ProError UserSolid2ScreenExample()
{
    ProError err;
    ProMdl model;
    ProMdlType type;
    ProSurface surface;
    ProModelitem p_mdl_item;
    ProPoint3d solid_point, screen_point;
    ProSolid solid;
    FILE *fp;
    ProPath path;

/*-----*\
    Get the current model and check if it is a solid
*\-----*/
    err = ProMdlCurrentGet( &model );
    ERROR_CHECK( "UserSolid2ScreenExample", "ProMdlCurrentGet", err );
    if ( err != PRO_TK_NO_ERROR ) return ( err );
    err = ProMdlTypeGet( model, &type );
    ERROR_CHECK( "UserSolid2ScreenExample", "ProMdlTypeGet", err );
    if ( (type != PRO_MDL_PART) && (type != PRO_MDL_ASSEMBLY) ) return (
err );
    solid = (ProSolid)model;

/*-----*\
    Let the user select a surface and extract the selected point
*\-----*/
    UserMakeSelections( &surface, &p_mdl_item, solid_point );

/*-----*\
    Transform from Solid to Screen Coordinates
*\-----*/
    UserSolidToScreen( solid, solid_point, screen_point );

/*-----*\
```

```

Write the results to a file and display it
\*-----*/
fp = PTApplsUnicodeFopen( "coord_trans.dat", "w" );
if ( fp != NULL ) {
    ProTKFprintf( fp, "You picked a point with these coordinates:\n");
    ProTKFprintf( fp, "\tX: %f\n", solid_point[0]);
    ProTKFprintf( fp, "\tY: %f\n", solid_point[1]);
    ProTKFprintf( fp, "\tZ: %f\n", solid_point[2]);
    ProTKFprintf( fp, "\n" );
    ProTKFprintf( fp, "Transformed to screen coordinates:\n");
    ProTKFprintf( fp, "\tX: %f\n", screen_point[0]);
    ProTKFprintf( fp, "\tY: %f\n", screen_point[1]);
    ProTKFprintf( fp, "\tZ: %f\n", screen_point[2]);
    fclose (fp);

    ProStringToWstring( path, "coord_trans.dat" );
    err = ProInfoWindowDisplay( path, NULL, NULL );
    ERROR_CHECK( "UserScreen2DrwExample", "ProInfoWindowDisplay", err );
}

return( PRO_TK_NO_ERROR );
}

```

Transforming Screen to Window Coordinates

Functions Introduced:

- **ProWindowCurrentMatrixGet()**
- **ProWindowPanZoomMatrixSet()**

Transformation from screen to window coordinates consists solely of a zoom factor and pan in X and Y.

The function **ProWindowCurrentMatrixGet()** gets the transformation matrix for the window. A pan and zoom transformation matrix consists of:

- The scale factor, running down the diagonal of the matrix. For example, to zoom in by a factor of 2, the value 2.0 will be down the diagonal in the elements (0,0), (1,1), and (2,2).
- The translation factor (pan) in the elements (3,0) - X and (3,1) - Y.
- The element at (3,3) should be 1.0.

The function **ProWindowPanZoomMatrixSet()** can change the pan and zoom of the window. The matrix should contain only the elements listed above, for function **ProWindowCurrentMatrixGet()**.

Transforming from Drawing View to Screen Coordinates in a Drawing

Function Introduced:

- **ProDrawingViewTransformGet()**

The function **ProDrawingViewTransformGet()** performs the transformation from drawing view coordinates (solid) to screen coordinates. It describes where a particular point on the solid will be in the drawing for a particular view of the solid.

Transforming from Screen to Drawing Coordinates in a Drawing

Function Introduced:

- **ProDrawingSheetTrfGet()**

The function **ProDrawingSheetTrfGet()** returns the matrix that transforms screen coordinates to drawing coordinates. The function performs this transformation for the first sheet.

Example 3: Screen Coordinates to Drawing Coordinates

```
/*----- Pro/Toolkit Includes -----*/
#include <ProToolkit.h>
#include <ProAsmcomppath.h>
#include <ProObjects.h>

/*----- Pro/Develop Includes -----*/
#include <prodev_types.h>

/*----- Function Prototypes -----*/
void UserScreenToDwg ();

void UserScreenToDwg (
    Prohandle drawing,
    ProPoint3d scn_point,
    ProPoint3d drw_point)
{
    wchar_t    w_size[10];
    ProMatrix  matrix;

    prodb_drawing_sheet_info(drawing, w_size, matrix);

    ProPntTrfEval(scn_point, matrix, drw_point);
}
```

Example 4: Transform from Screen Coordinates to Drawing Coordinates

This example allows you to select a point from the screen, transform it from screen coordinates to drawing coordinates, write the results into a file and display it.

```
/*----- Pro/Toolkit Includes -----*/
#include <ProToolkit.h>
#include <ProMdl.h>
#include <ProMessage.h>
#include <ProGraphic.h>
#include <ProWchar.h>
#include <ProUtil.h>
#include <PTApplsUnicodeUtils.h>
/*----- Pro/Develop Includes -----*/
#include <prodevelop.h>

/*----- Application Includes -----*/
#include <TestError.h>

/*----- Function Prototypes -----*/
ProError UserScreen2DrwExample();
ProError UserSolid2ScreenExample();

/*----- External Data -----*/
extern void UserScreenToDwg();
extern void UserSolidToScreen();
extern int UserMakeSelections();

ProError UserScreen2DrwExample()
{
    ProError err;
    ProMdl model;
    ProMdlType type;
    Prohandle drawing;
    ProFileName msgfil;
    ProPoint3d position, drw_point;
    ProMouseButton button_pressed;
    FILE *fp;
    ProPath path;

/*-----*\
    Get the current model and check if it is a drawing
\*-----*/
    err = ProMdlCurrentGet( &model );
    ERROR_CHECK( "UserScreen2DrwExample", "ProMdlCurrentGet", err );
    if ( err != PRO_TK_NO_ERROR ) return ( err );
    err = ProMdlTypeGet( model, &type );
    ERROR_CHECK( "UserScreen2DrwExample", "ProMdlTypeGet", err );
    if ( type != PRO_MDL_DRAWING ) return ( err );
```



```

        drawing = (Prohandle)model;

/*-----*\
    Prompt the user for a menu pick and get coordinates
\*-----*/
    ProStringToWstring( msgfil, "msg_ugfund.txt" );

    err = ProMessageDisplay( msgfil, "USER Pick point on the screen" );
    ERROR_CHECK( "UserScreen2DrwExample", "ProMessageDisplay", err );

    err = ProMousePickGet( PRO_ANY_BUTTON, &button_pressed, position );
    ERROR_CHECK( "UserScreen2DrwExample", "ProMousePickGet", err );

/*-----*\
    Transform the screen coordinates to drawing coordinates
\*-----*/
    UserScreenToDwg( drawing, position, drw_point );

/*-----*\
    Write the results to a file and display it
\*-----*/
    fp = PTApplsUnicodeFopen( "coord_trans.dat", "w" );
    if ( fp != NULL ) {
        ProTKFprintf( fp, "You picked a point with these coordinates:\n");
        ProTKFprintf( fp, "\tX: %f\n", position[0]);
        ProTKFprintf( fp, "\tY: %f\n", position[1]);
        ProTKFprintf( fp, "\tZ: %f\n", position[2]);
        ProTKFprintf( fp, "\n" );
        ProTKFprintf( fp, "Transformed to drawing coordinates:\n");
        ProTKFprintf( fp, "\tX: %f\n", drw_point[0]);
        ProTKFprintf( fp, "\tY: %f\n", drw_point[1]);
        ProTKFprintf( fp, "\tZ: %f\n", drw_point[2]);
        fclose (fp);

        ProStringToWstring( path, "coord_trans.dat" );
        err = ProInfoWindowDisplay( path, NULL, NULL );
        ERROR_CHECK( "UserScreen2DrwExample", "ProInfoWindowDisplay", err );
    }

    return ( PRO_TK_NO_ERROR );
}

```

Transforming Coordinates of an Assembly Member

Function Introduced:

- **ProAsmcomppathTrfGet()**

The function **ProAsmcomppathTrfGet()** provides the matrix for transforming from the solid coordinate system of the assembly member to the solid coordinates of the parent assembly, or the reverse.

Transforming to Coordinate System Datum Coordinates

Functions Introduced:

- **ProCsysDataGet()**
- **ProMatrixInit()**

The function **ProCsysDataGet()** provides the location and orientation of the coordinate system datum in the solid coordinate system of the solid that contains it. The location is in terms of the directions of the three axes, and the position of the origin. When these four vectors are made into a transformation matrix using the function **ProMatrixInit()**, that matrix defines the transformation of a point described relative to the coordinate system datum back to solid coordinates. To transform the other way, which is the more usual requirement, you need to invert the matrix. The example function **ProUtilMatrixInvert()**, inverts the specified matrix.

Transforming Coordinates of Sketched Entities

Function Introduced:

- **ProSectionLocationGet()**

The function **ProSectionLocationGet()** provides the matrix for transforming from the solid coordinate system to the sketch coordinate system, or the reverse.

Example 5: Using Several Coordinate Transforms

The example code below demonstrates how to use several coordinate transformations. The function will zoom in on a solid model, with the results centered at the selected location.

```
#include <ProToolkit.h>
#include <ProAsmcomppath.h>
#include <ProSelection.h>
#include <ProView.h>
```

```
#include <ProMessage.h>

/*=====*\
FUNCTION: UserZoomAtSelPoint
PURPOSE: Zooms window in by factor of 2 at selected point on a solid
         model.
\*=====*/
ProError UserZoomAtSelPoint()
{
    ProError status;
    ProSelection *p_sel;
    int n_sel;
    ProAsmcomppathcomp_path;
    ProPoint3d p3d;
    ProMatrixmatrix, zoom_matrix;
    ProPoint3d t_point;
    ProFileName msgfile;
    ProModelitem item;
    ProMdl top_model;

    ProPoint3d scrpnt;
    int i, j, k;
    int window;
    double scale;

    /*-----*\
    The Pro/ENGINEER "virtual window". Used to calculate the amount
    of pan needed to center the zoomed window on the chosen point.
    \*-----*/
    double window_outline [2][3] = {{0.0, 0.0, 0.0}, {1000.0, 843.75, 0.0}};

    ProStringToWstring (msgfile, "msg_uggraph.txt");

    /*-----*\
    Prompt the user to select the center of the zoom
    \*-----*/
    ProMessageDisplay (msgfile,
        "USER Select a location on a solid model for zoom ");

    status = ProSelect ("prt_or_asm", 1, NULL, NULL, NULL, NULL,
        &p_sel, &n_sel);
    if (status != PRO_TK_NO_ERROR)
        return (status);

    /*-----*\
    Get the selected point
    \*-----*/
    ProSelectionPoint3dGet (p_sel[0], p3d);

    /*-----*\
    Get the assembly component path for the selected component
```

```

\*-----*/
status = ProSelectionAsmcomppathGet (p_sel[0], &comp_path);
if (status == PRO_TK_NO_ERROR && comp_path.owner != NULL)
{

top_model = comp_path.owner;
/*-----*\
Get the coordinate transformation matrix for the component to
the assembly
\*-----*/
ProAsmcomppathTrfGet (&comp_path, PRO_B_TRUE, matrix);

/*-----*\
Transform the selected point into the coordinate system of the
top level assembly
\*-----*/
status = ProPntTrfEval(p3d, matrix, t_point);

}
else
{

/*-----*\
If the selection is the top level component prepare the needed
parameters for zooming
\*-----*/
ProSelectionModelitemGet (p_sel[0], &item);
top_model = item.owner;
memcpy (t_point, p3d, sizeof (ProPoint3d));
}

/*-----*\
Get the view transformation matrix (from the current window)
\*-----*/
ProViewMatrixGet (top_model, NULL, matrix);

/*-----*\
Transform the solid model point to screen coordinates
\*-----*/
ProPntTrfEval (t_point, matrix, scrpnt);

/*-----*\
Get the window pan-zoom transformation matrix
\*-----*/
ProWindowCurrentMatrixGet (matrix);

/*-----*\
Zoom in on the created point
\*-----*/
for (j=0; j<4; j++)
{

```

```

for (k=0; k<4; k++)
zoom_matrix [j][k] = 0.0;
}

/*-----*\
Double the existing window scale
\*-----*/
scale = 2.0 * matrix [0][0];

for (j=0; j<3; j++)
{
zoom_matrix [j][j] = scale;
zoom_matrix [3][j] =
(window_outline [1][j] - window_outline [0][j])/2.0 -
scrpnt[j]*scale;
}

zoom_matrix [3][3] = 1.0;

ProMdlWindowGet (top_model, &window);

status = ProWindowPanZoomMatrixSet (window, zoom_matrix);

/*-----*\
Repaint the window
\*-----*/
ProWindowRepaint (window);

return (status);
}

```


9

Core: Family Tables

This chapter describes how to use Pro/TOOLKIT functions to manipulate the family table for an object.

Topic	Page
Family Table Objects	9 - 2
Family Table Utilities	9 - 2
Visiting Family Tables	9 - 3
Operations on Family Table Instances	9 - 4
Operations on Family Table Items	9 - 7

Family Table Objects

To enable access to family tables, Pro/TOOLKIT implements the following objects (all DHandles):

- *ProFamtable*—A structure that contains the owner, type, and integer identifier of a family table.
- *ProFaminstance*—A structure that contains the name of a family table instance and the handle of the family table to which it belongs.
- *ProFamtableItem*—A structure that contains the type, name, and owner of a family table item (or column).

Family Table Utilities

Functions Introduced:

- **ProFamtableInit()**
- **ProFamtableCheck()**
- **ProFamtableEdit()**
- **ProFamtableShow()**
- **ProFamtableErase()**
- **ProFamtableIsModifiable()**

Before you can manipulate the family table information stored in an object, you must get the handle to its family table using the function **ProFamtableInit()**. Then use **ProFamtableCheck()** to determine whether the family table is empty (for a ProSolid, ProPart, or ProAssembly object, use **ProSolidFamtableCheck()**).

The function **ProFamtableEdit()** opens a Pro/TABLE window for editing the specified family table, producing the same effect as the Pro/ENGINEER command **Family Tab, Edit**.

Similarly, a call to **ProFamtableShow()** presents the family table in the same manner as the Pro/ENGINEER command **Family Tab, Show**.

The function **ProFamtableErase()** clears the family from the current session, similar to the Pro/ENGINEER command **Family Tab, Erase Table**.

The function **ProFamtableIsModifiable()** checks whether the specified family table can be modified.

Visiting Family Tables

Functions Introduced:

- **ProFamtableInstanceVisit()**
- **ProFamtableItemVisit()**

As with the other Pro/TOOLKIT traversal functions, the traversal functions for family tables visit family table objects and pass each object to action and filter functions that you supply.

For example, the function **ProFamtableInstanceVisit()** visits all the family's instances and calls the user-supplied functions of type **ProFamtableInstanceAction()** and **ProFamtableInstanceFilter()**.

The function **ProFamtableItemVisit()** visits each family table item (or column) and calls the user-supplied functions of type **ProFamtableItemAction()** and **ProFamtableItemFilter()**.

Example 1: Writing a Family Table to a File on page 9 - 7 shows how to use **ProFamtableInstanceVisit()** and **ProFamtableItemVisit()** to completely traverse a family table and write its contents to a file.

Operations on Family Table Instances

Functions Introduced:

- **ProFamInstanceValueGet()**
- **ProFamInstanceValueSet()**
- **ProFamInstanceFamtableItemsDefault()**
- **ProFamInstanceAdd()**
- **ProFamInstanceCheck()**
- **ProFamInstanceIsModifiable()**
- **ProFamInstanceInit()**
- **ProFamInstanceRemove()**
- **ProFamInstanceCreate()**
- **ProFamInstanceSelect()**
- **ProFamInstanceMdlGet()**
- **ProFamInstanceErase()**
- **ProFamInstanceLock()**

- **ProFamInstanceRetrieve()**
- **ProFamInstanceGenericGet()**
- **ProFamInstanceImmediateGenericInfoGet()**
- **ProFamInstanceIsVerified()**
- **ProFamInstanceIsExtLocked()**
- **ProFamInstanceIsFlatState()**

The functions in this section enable you to programmatically manipulate instances that appear in a family table.

The function **ProFamInstanceValueGet()** retrieves the value of the family table item for the specified family table instance. Use the function **ProFamInstanceValueSet()** to change the value of the specified family table item.

For the specified instance, use the function **ProFamInstanceFamtableItemIsDefault()** to determine if the specified item has the default value, which is the value of the specified item in the generic model.

The functions **ProFamInstanceValueGet()**, **ProFamInstanceValueSet()**, and **ProFamInstanceFamtableItemIsDefault()** require the instance and the item handles as input values, both of which are available via the visit functions. Example 1: Writing a Family Table to a File uses the function **ProFamInstanceValueGet()** to write the contents of the visited family table items to a file.

The function **ProFamInstanceAdd()** adds an instance to a family table. Note that you must initialize the handle to the new instance (using **ProFamInstanceInit()**) before adding the instance to the table.

Use the function **ProFamInstanceRemove()** to remove the specified instance from the family table.

The function **ProFamInstanceCreate()** creates an instance model (a *ProMdl* object) of the specified instance handle. This function is particularly useful when used in conjunction with **ProFamInstanceSelect()**, which enables users to make one or more selections from a menu of family table instances. Note that you must call **ProSolidDisplay()** to display the instance model that results from the call to **ProFamInstanceCreate()**.

The function **ProFamInstanceMdlGet()** retrieves the handle to the instance model for the given instance that is in session.

To erase an instance model from memory, call the function **ProFaminstanceErase()**.

The function **ProFaminstanceCheck()** checks the existence and lock status of the specified instance. Use the function **ProFaminstanceLock()** to make changes to the lock status of an instance.

The function **ProFaminstanceIsModifiable()** checks whether the given instance of a family table can be modified.

Given the instance handle, the function **ProFaminstanceRetrieve()** retrieves an instance of a model from disk. Note that you must allocate space for the resulting *ProMdl* object. In addition, you must call **ProSolidDisplay()** to display the instance model.

The function **ProFaminstanceGenericGet()** retrieves the generic model handle for a given instance model. This function includes the ability to choose between the immediate and the top-level generic models.

From Pro/ENGINEER Wildfire 4.0 onwards, the behavior of the function **ProFaminstanceGenericGet()** has changed as a result of performance improvement in family table retrieval. When you now call the function **ProFaminstanceGenericGet()** with the flag "immediate" set to TRUE, the function returns a new error code PRO_TK_CANT_OPEN if the immediate generic is currently not in session. Use the function

ProFaminstanceImmediategenericinfoGet() to get the name and model type of the immediate generic model. This information can be used to retrieve the immediate generic model.

If you wish to turn-off this behavior and continue to run legacy applications in the pre-Wildfire 4.0 mode, set the configuration option "retrieve_instance_dependencies" to instance_and_generic_deps.

The function **ProFaminstanceIsVerified()** identifies whether the instance has been verified, and whether the verification succeeded or failed.

The function **ProFaminstanceIsExtLocked()** identifies whether the instance has been locked by an external application.

The function **ProFaminstanceIsFlatState()** identifies whether an instance is a sheetmetal flat state instance.

Operations on Family Table Items

Functions Introduced:

- **ProFamtableItemAdd()**
- **ProFamtableItemRemove()**
- **ProFamtableItemToModelitem()**
- **ProModelitemToFamtableItem()**
- **ProFamtableItemToParameter()**
- **ProParameterToFamtableItem()**

These functions enable you to programmatically manipulate family table items (column values).

The function **ProFamtableItemAdd()** adds the specified item to a family table. Similarly, **ProFamtableItemRemove()** removes the specified item from the family table.

The functions **ProFamtableItemToModelitem()** and **ProModelitemToFamtableItem()** convert between *ProFamtableItem* and *ProModelitem* objects. Note that user selections (*ProSelection* objects) can be converted to *ProFamtableItem* objects by calling the functions **ProSelectionModelitemGet()** and **ProModelitemToFamtableItem()**.

The functions **ProFamtableItemToParameter()** and **ProParameterToFamtableItem()** convert between *ProFamtableItem* and *ProParameter* objects. Note that you might need to call **ProParameterToFamtableItem()** after calling **ProParameterSelect()** (which enables users to select parameters from a menu).

Example 1: Writing a Family Table to a File

The following example shows how to use the Pro/TOOLKIT functions to write the contents of a family table to a file.

```
/*-----*\
Application data
\*-----*/
typedef struct userdata {
    FILE          *fp;          /* file pointer */
    ProFaminstance *fam_inst;    /* family table instance */
    ProFamtable    fam_table;    /* family table */
} UserData;
```

```

/*=====*\
FUNCTION: UserFamtableInstAct()
PURPOSE: Action function for ProFamtableInstanceVisit()
\*=====*/
ProError UserFamtableInstAct (
    ProFamInstance *instance,      /* In */
    ProError      filt_status,     /* In */
    ProAppData    appdata)        /* In */
{
    UserData      *data = (UserData *) appdata;
    int           status;
    ProCharName    inst_name;
    ProError       UserFamtableItemAct();
    ProFamtable    *p_table = &data->fam_table;

/*-----*\
    Add the instance handle to the data.
\*-----*/
    data->fam_inst = instance;

/*-----*\
    Print the instance name in the family table file.
\*-----*/
    ProWstringToString (inst_name, instance->name);
    fprintf (data->fp, "\nInstance name: %s\n", inst_name);

/*-----*\
    Visit each family table item.
\*-----*/
    status = ProFamtableCheck (p_table);
    status = ProFamtableItemVisit (p_table,
        (ProFamtableItemAction)UserFamtableItemAct, NULL, data);
    return (0);
}

/*=====*\
FUNCTION: UserFamtableItemAct()
PURPOSE: Action function for ProFamtableItemVisit()
\*=====*/
ProError UserFamtableItemAct (
    ProFamtableItem *fam_item,      /* In */
    ProError      filt_status,     /* In */
    ProAppData    appdata)        /* In */
{
    UserData      *data = (UserData *) appdata;
    ProCharName    item_name, string_val;
    char           str1[PRO_NAME_SIZE];
    int           status;
    ProParamvalue  item_val;
    int           lock_status;

/*-----*\
    Get the name of the current table item (column).
\*-----*/
    ProWstringToString (item_name, fam_item->string);

```

```

/*-----*\
    For the current instance, get the value of the current
    table item.
\*-----*/
    status = ProFamInstanceValueGet (data->fam_inst, fam_item,
                                     &item_val);

    switch (item_val.type)
    {
        case PRO_PARAM_DOUBLE:
            sprintf (str1, "%lf", item_val.value.d_val);
            break;
        case PRO_PARAM_STRING:
            ProWstringToString (string_val, item_val.value.s_val);
            sprintf (str1, "%s", string_val);
            break;
        case PRO_PARAM_INTEGER:
            sprintf (str1, "%d", item_val.value.i_val);
            break;
        case PRO_PARAM_BOOLEAN:
            sprintf (str1, "%d", item_val.value.l_val);
            break;
        case PRO_PARAM_NOTE_ID:
            sprintf (str1, "%s", "PRO_PARAM_NOTE_ID");
            break;
        case PRO_PARAM_VOID:
            sprintf (str1, "%s", "PRO_PARAM_VOID");
            break;
        default:
            sprintf (str1, "%s", "No value.");
            break;
    }

    fprintf (data->fp, "\tItem : %s \tValue: %s\n", item_name, str1);
    return (0);
}

/*=====*\
FUNCTION: UserFamtable()
PURPOSE:  Example function using the family table functions
\*=====*/
ProError UserFamtable (ProMdl model)
{
    ProError      UserFamtableInstAct();
    ProFileName   msgfile, wfname;
    ProFamtable   *p_famtable;
    ProCharName   fname;
    int           status;
    UserData      UsrData;

    ProStringToWstring (msgfile, "testmsg.txt");

```

```

/*-----*\
    Retrieve the model's family table and add it to the user data.
/*-----*/
    status = ProFamtableInit (model, p_famtable);
    UsrData.fam_table = *p_famtable;
/*-----*\
    Check the family table.
/*-----*/
    status = ProFamtableCheck (p_famtable);
    if (status == PRO_TK_E_NOT_FOUND)
    {
        ProMessageDisplay (msgfile, "TEST %0s", "The family table
            does not exist.");
        return (PRO_TK_E_NOT_FOUND);
    }
/*-----*\
    Open a file for the family table data.
/*-----*/
    UsrData.fp = (FILE *) ProUtilGenFilePtr (model, ".ftb", fname,
        "w+");
    fprintf (UsrData.fp, "\nFamily table filename: %s\n", fname);
/*-----*\
    Visit each instance in the family table.
/*-----*/
    status = ProFamtableInstanceVisit (p_famtable,
        (ProFamtableInstanceAction)UserFamtableInstAct, NULL,
        (ProAppData)&UsrData);
/*-----*\
    Close the family table file.
/*-----*/
    fclose (UsrData.fp);
/*-----*\
    Display the family table.
/*-----*/
    ProStringToWstring (wfname, fname);
    status = ProInfoWindowDisplay (wfname, NULL, NULL);
    return (0);
}

```


10

Core: External Data

This chapter describes how to store and retrieve external data. External data is a Pro/TOOLKIT application to be stored in the Pro/ENGINEER database in a way that is invisible to the Pro/ENGINEER user.

Topic	Page
Introduction to External Data	10 - 2
Storing External Data	10 - 4
Retrieving External Data	10 - 5

Introduction to External Data

External data provides a way for the Pro/TOOLKIT application to store its own private information about a Pro/ENGINEER model within the model file. The data is built and interrogated by the Pro/TOOLKIT application as a workspace data structure. It is saved to the model file when the model is saved, and retrieved when the model is retrieved. The external data is otherwise ignored by Pro/ENGINEER, so the Pro/TOOLKIT application has complete control over the form and content.

The external data for a particular Pro/ENGINEER model is broken down into classes and slots. A class is a named “bin” for your data, and simply identifies it as yours so no other Pro/TOOLKIT application (or other classes in your own application) will use it by mistake. A Pro/TOOLKIT application usually needs only one class. The class name should be unique for each model, and describe the role of the data in your application.

Each class contains a list of data slots. Each slot is identified by either a name or an identifier, and contains a single data item of one of the following types:

- Integer
- Double
- Wide string (maximum length = 512 characters)
- Stream (maximum size = 512 kilobytes). A slot of type stream contains a completely unformatted sequence of bytes with unrestricted values. The slot also records the number of bytes in the stream, so no termination rules are assumed. The stream type should be used only when the format is completely controlled by your application in a platform-independent way. For example, if the volume of external data is very large, the stream format might be used to store the data in a more compressed form for greater efficiency.

Stream slots could also be used as a shortcut way to store, for instance, an entire C structure, or an array of C structures, without any formatting. However, if you are supporting more than one platform with your Pro/TOOLKIT application, remember that the mapping of a C structure may differ between platforms.

If external data is stored during a Pro/ENGINEER session on one platform and retrieved on another, the values of integer, double, and wide string slots will be preserved correctly, regardless of any differences in the coding of those data types by the two C compilers. Stream slots will be preserved with exactly the same byte values and sequence that was saved, regardless of byte-swap conventions on the two platforms.

External data is stored in the workspace and is accessible only through the functions provided for that purpose. Two objects are used to reference the data contents: *ProExtdataClass* and *ProExtdataSlot*. These are both declared as DHandles—visible data structures. The declarations are as follows:

```
typedef struct pro_extdata_class
{
    ProMdl    p_model;
    ProName   class_name;
} ProExtdataClass;

typedef struct pro_extdata_slot
{
    ProExtdataClass *p_class;
    ProName          slot_name;
    int              slot_id;
} ProExtdataSlot;
```

Each slot has two ways to be identified: a name, which is defined by the application when the slot is created, or an identifier, which is allocated automatically. You can choose which kind of identifier to use for each slot.

The Pro/TOOLKIT functions for external data do not use the usual return type *ProError*. Instead, they use an enumerated type called *ProExtdataErr* that contains error statuses that are more specific to the needs of those functions.

All the declarations relevant to external data are in the header file *ProExtdata.h*.

Storing External Data

Functions Introduced:

- **ProExtdataInit()**
- **ProExtdataClassRegister()**
- **ProExtdataClassUnregister()**
- **ProExtdataSlotCreate()**
- **ProExtdataSlotWrite()**
- **ProExtdataSlotDelete()**
- **ProExtdataTerm()**

Note: For functions **ProExtdataClassRegister()** and **ProExtdataSlotCreate()**, the combined length of the class and slot names must not exceed PRO_NAME_SIZE.

The first step in manipulating external data for a model in a Pro/ENGINEER session is to call the initialize function **ProExtdataInit()** for that model.

Next, set up a class using the function **ProExtdataClassRegister()**. The inputs are the *ProMdl* object and the class name, in the form of a wide string. The function outputs a *ProExtdataClass* used thereafter by the application to reference the class.

You can delete a class that is no longer needed using the function **ProExtdataClassUnregister()**.

The function **ProExtdataSlotCreate()** creates an empty data slot. The inputs are the *ProExtdataClass* object and the slot name, in the form of a wide string. The function outputs a *ProExtdataSlot* object to identify the new slot. You can use NULL as the value of the slot name argument, in which case the function allocates a unique integer identifier for the slot (which becomes the value of the field *slot_id* in the *ProExtdataSlot* structure).

Note: Slot names cannot begin with a number.

The function **ProExtdataSlotWrite()** specifies the slot data type and writes an item of that type to the slot. The inputs are:

- The slot object *ProExtdataSlot*
- A flag showing whether the slot is identified by name or integer
- The data type of the slot

- The number of bytes in the data (needed only if the type is stream)
- A pointer to the data (cast to *void**)

A slot of type stream has a maximum size of 512 kilobytes. If this size is exceeded, **ProExtdataSlotWrite()** returns the status PROEXTDATA_TK_STREAM_TOO_LARGE.

You can delete an unused slot using the function **ProExtdataSlotDelete()**.

If the user and application no longer need external data in session, call **ProExtdataTerm()** to clean the external data from memory.

Note: **ProExtdataTerm()** does not affect the contents of any file on the disk. It only removes all external data from the memory. Changes made to external data during the current session are not stored in the file until you save the model. If you call **ProExtdataTerm()** after making changes to the model, all external data changes (such as newly created slots, changed slot value, and deleted slot) made since the last **ProMdlSave()** are lost.

Retrieving External Data

Functions Introduced:

- **ProExtdataLoadAll()**
- **ProExtdataClassNamesList()**
- **ProExtdataSlotIdsList()**
- **ProExtdataSlotNamesList()**
- **ProExtdataSlotRead()**
- **ProExtdataFree()**

For improved performance, external data is not loaded automatically into memory with the model. When the model is in session, call the function **ProExtdataLoadAll()** to retrieve all the external data for the specified model from the Pro/ENGINEER model file and put it in the workspace. The function needs to be called only once to retrieve all the data.

Note that the function **ProExtdataLoadAll()** provides better performance than **ProExtdataClassNamesList()**, **ProExtdataSlotNamesList()**, and **ProExtdataSlotRead()** because these functions load only specific information (class names, slot names, and slot files, respectively), which can be slow.

The **ProExtdataClassNamesList()** function provides an array of the names of all the external data classes registered in a specified model.

The function **ProExtdataSlotIdsList()** provides an array of the integer identifiers of all the slots in a specified class. The input is a *ProExtdataClass* structure that must be set up manually or programmatically. The function **ProExtdataSlotNamesList()** provides an array of the names of the slots in the specified class. The function allocates a term in the array for each slot, even if you did not assigned a name to the slot.

The function **ProExtdataSlotRead()** reads the data type and data from a specified slot. Its input is a *ProExtdataSlot* structure that must be set up manually. There is also an input argument to show whether the slot is identified by name or by integer. The function outputs the data type, the number of bytes (if the type was stream), and a pointer to the data itself.

The **ProExtdataSlotRead()** function allocates memory for the data it outputs. To free this memory, call **ProExtdataFree()**.

Note: If you call **ProExtdataSlotRead()** multiple times and do not free the memory, the function uses the same memory for each call.

11

Core: Cross Sections

The functions in this chapter enable you to access, modify, and delete cross sections, and create planar cross sections.

Topic	Page
Listing Cross Sections	11 - 2
Extracting Cross-Sectional Geometry	11 - 3
Visiting Cross Sections	11 - 6
Creating and Modifying Cross Sections	11 - 7
Mass Properties of Cross Sections	11 - 7

Listing Cross Sections

Functions Introduced:

- **prodb_first_xsec()**
- **prodb_next_xsec()**
- **prodb_xsec_name()**
- **ProXsecRename()**
- **ProXsecTypeGet()**

The functions **prodb_first_xsec()** and **prodb_next_xsec()** return handles to the cross sections in the specified model, and the number of components in each cross section. For an explanation of components, see the following section.

The function **prodb_xsec_name()** returns the name (as a wide string) of a cross section, specified by its handle. Function **ProXsecRename()** renames a specified cross section.

Use function **ProXsecTypeGet()** to retrieve the type of cross section. The input argument for this function is a handle to the specified cross section. The output argument is a *p_type* structure, which contains the following fields:

- *cutter*—Specifies the cutting object type. The valid cutting object types are contained in the enumerated type *ProXsecCut* and are as follows:
 - PRO_XSEC_PLANAR
 - PRO_XSEC_OFFSET
 - PRO_XSEC_PATTERN
- *cut_object*—Specifies the object that you want to cut. The valid object types are contained in the enumerated type *ProXsecCutobj* and are as follows:
 - PRO_XSECTYPE_MODEL
 - PRO_XSECTYPE_QUILTS
 - PRO_XSECTYPE_MODELQUILTS

Extracting Cross-Sectional Geometry

Functions Introduced:

- **ProXsecGeometryCollect()**
- **ProXsecGeometryArrayFree()**
- **ProXsecGeometryFree()**
- **ProXsecRegenerate()**
- **ProXsecDisplay()**
- **ProXsecPlaneGet()**

Superseded Functions:

- **prodb_xsec_component()**
- **prodb_regen_xsec()**
- **prodb_display_xsec()**
- **ProXsecGeometryRetrieve()**

The geometry of a cross section in an assembly is divided into components. Each component corresponds to one of the parts in the assembly that is intersected by the cross section, and describes the geometry of that intersection. A component can have disjoint geometry if the cross section intersects a given part instance in more than one place.

A cross section in a part has a single component.

The components of a cross section are identified by consecutive integer identifiers that always start at 0.

The function **ProXsecGeometryCollect()** returns an array containing the geometry of all components in the specified cross section. The function **ProXsecGeometryCollect()** supersedes the function **ProXsecGeometryRetrieve()**.

The function **ProXsecGeometryRetrieve()** returns an array containing the geometry of all components in the specified cross section and retrieves the following information about a specified cross-sectional component:

- The *memb_num* and *memb_id_tab* for the intersected part, with respect to the assembly that contains the cross section.
- A handle to the geometry of the intersection.

The geometry handle can be treated as an ordinary face pointer. Extract its contours with function **ProSurfaceContourVisit()**.

Use the function **ProXsecGeometryArrayFree()** to free the memory allocated to the array of cross section data and use the function **ProXsecGeometryFree()** to free memory allocated to the cross section data.

In an offset cross section, each component can contain several geometrical faces corresponding to the intersections of the cross-section planes with that part. In such a case, the face handle retrieved by **prodb_xsec_component()** is the first in a list of handles. Obtain the other members using successive calls to **prodb_first_face_contour()** and **prodb_next_face_contour()**.

The geometry of a cross section is not maintained constantly by Pro/ENGINEER—it is regenerated only when the user requests to see the cross section. Use function **ProXsecRegenerate()** to regenerate a specified planar or offset cross section. Use function **ProXsecDisplay()** to display a cross section. **ProXsecDisplay()** does not add the cross section to the associated objects list, and the displayed cross section disappears on the first screen redraw.

The function **ProXsecPlaneGet()** returns the plane geometry for a specified cross section.

Example 1: Listing Cross Sections

The following example code shows a function that lists the cross sections in the current model and summarizes their geometry.

```
/*----- Pro/Toolkit Includes -----*/
#include <ProToolkit.h>
#include <PTApplsUnicodeUtils.h>
/*----- Pro/Develop Includes -----*/
#include "prodevelop.h"

/*----- Application Includes -----*/
#include <TestError.h>

/*----- Function Prototypes -----*/
void user_List_Xsecs();
void user_member_path ();

void user_List_Xsecs()
{
    Prohandle    h_obj, h_xsec, xsec_geom, h_face, h_cont;
    ProName      w_name;
    ProCharName  name;
    char         path[200];
```

```

int          n_components, c, n_faces, n_conts;
int          memb_num, *memb_id_tab;
ProCharName filename = {'x','s','e','c','s','.','l','s','t','\0'};
ProFileName wfilename;

FILE  fp;

fp = PTApplsUnicodeFopen(filename, "w");

h_obj = pro_get_current_object();
/*-----*\
For each cross section...
\*-----*/
for (h_xsec = prodb_first_xsec (h_obj, &n_components);
    h_xsec != NULL;
    h_xsec = prodb_next_xsec (h_obj, h_xsec,
    &n_components))
{
/*-----*\
Get the name.
\*-----*/
    if (prodb_xsec_name (h_obj, h_xsec, w_name) == 0)
        continue;
    ProTKFprintf(fp, "Cross section %s\n", pro_wstr_to_str (name,
        w_name));
/*-----*\
Regenerate, so you can extract the geometry.
\*-----*/
    prodb_regen_parallel_xsec (h_obj, h_xsec);
/*-----*\
For each component of the cross section
\*-----*/
    for (c = 0; c < n_components; c++)
    {
/*-----*\
Get the assembly member and geometry.
\*-----*/
        prodb_xsec_component (h_obj, h_xsec, c, &memb_num,
            &memb_id_tab, &xsec_geom);
/*-----*\
Print the assembly path of the component.
\*-----*/
        user_member_path (h_obj, memb_num, memb_id_tab, path);
        ProTKFprintf(fp, "    Component %2d : %s\n", c, path);
/*-----*\
List the component faces and contours.
\*-----*/
        h_face = xsec_geom;
        n_faces = 1;
        do
        {

```

```

        ProTKFprintf(fp, "          Face %d, area %f\n", n_faces++,
pro_face_area (h_face));
        n_conts = 1;
        for (h_cont = prodb_first_face_contour (h_face);
h_cont != NULL;
h_cont = prodb_next_face_contour (h_cont))
            ProTKFprintf(fp, "          Contour %d\n", n_conts++);
    }
    while ((h_face = prodb_next_part_face (h_face)) != NULL);
}
}
ProInfoWindowDisplay(wfilename, NULL, NULL);
}

void user_member_path (h_obj, memb_num, memb_id_tab, path)
Prohandle      h_obj;
int memb_num, *memb_id_tab;
char          path[];
{
    Pro_object_info  info;
    char            name[80], type[10], buff[100];
    int             m;

    prodb_get_object_info (h_obj, &info);
    strcpy (path, pro_wstr_to_str (name, info.name));
    strcat (path, ".");
    strcat (path, pro_wstr_to_str (type, info.type));
    for (m = 0; m < memb_num; m++)
    {
        h_obj = prodb_member_to_object (h_obj, memb_id_tab[m]);
        prodb_get_object_info (h_obj, &info);
        ProTKSprintf(buff, "%s.%s[%d]", pro_wstr_to_str (name,
            info.name),
pro_wstr_to_str (type, info.type), memb_id_tab[m]);
        strcat (path, buff);
    }
}

```

Visiting Cross Sections

Function Introduced:

- **ProSolidXsecVisit()**
- **ProSolidXsecVisitAction()**

The function **ProSolidXsecVisit()** enables you to visit all named cross sections in the specified solid. Use **ProSolidXsecVisitAction()** to supply the function to be performed when visiting part or assembly cross sections.

Creating and Modifying Cross Sections

Functions Introduced:

- **proddb_create_parallel_xsec()**
- **ProXsecPlanarCreate()**
- **ProXsecDelete()**

Functions Superseded:

- **proddb_delete_xsec()**

The function **proddb_create_parallel_xsec()** creates a planar cross section using an offset from an existing plane (either a plane geometry surface or a datum plane). The specified offset results in a new model dimension, whose identifier is returned by this function.

Once such a cross section has been created, it can be moved by modifying the offset dimension using **prodim_set_dim()** in the usual way, followed by a call to **ProXsecRegenerate()** to recalculate the geometry of the cross section.

Use function **ProXsecPlanarCreate()** to make a cross section through a datum plane. Function **ProXsecDelete()** deletes a given cross section from a part or assembly.

Note: The function **proddb_create_parallel_xsec()** creates not only a parallel cross section, but also a new feature that contains a datum plane used to define the location of the cross section. Applications that use superseded function **proddb_delete_xsec()** to delete cross sections must first call **proddb_delete_xsec()** (to delete the cross section) and then call **ProFeatureDelete()** on the owning feature (to delete the datum plane).

Refer to Example 2: Calculating the Mass Properties of a Cross Section for data on using these functions.

Mass Properties of Cross Sections

Function Introduced:

- **ProXsecMassPropertyCompute()**

Function Superseded:

- **proddb_mass_prop_xsec()**

The function **ProXsecMassPropertyCompute()** calculates the center of gravity of a planar cross section. The function needs the name of a coordinate system datum whose X- and Y-axes are parallel to the cross section. The output from this function also refers to the coordinate system datum. Call **ProXsecRegenerate()** before **ProXsecMassPropertyCompute()**.

Example 2: Calculating the Mass Properties of a Cross Section

The following example code shows a command that calculates and displays the three-dimensional trajectory of the center of gravity (COG) of a cross section as the cross section is swept across the model. The user selects a starting plane, a point to define the finishing offset, and a number of steps. The cross section is then created, moved, analyzed, and finally deleted automatically.

```
/*=====*\
FUNCTION: user_Neutral_Axis()
PURPOSE: Calculate the mass properties of a cross section.
\*=====*/
int user_Neutral_Axis()
{
    ProError          status;
    ProPart           part;
    int               n_sel;
    ProSelection      *sel;
    ProModelitem      surface_modelitem, csys_modelitem;
    ProSurface        surface;
    ProSrftype        stype;
    ProPoint3d        finish, normal, pos, cross, xpos, cog;
    ProUvParam        uv;
    double            dist, absdist, offset;
    ProCsys           csys;
    ProGeomitemdata   *csys_geomdata;
    ProCsysdata       *csys_data;
    ProMatrix         transf, nulltransf;
    ProName           w_csys_name, w_name;
    int               n_steps, xsec_status, dim_id, id;
    Prohandle         xsec;
    PRODIMENSION      dimension;
    Pro_mass_property *mprop;
    ProPoint3d        points[100];
    int               n_points=0;
    ProFeature        feature;
    int               feat_ids[1];
    ProFeatureDeleteOptions opts[1];

    ProStringToWstring (msgfil, "msg_ugxsec.txt");
    ProMdlCurrentGet ((ProMdl*)&part);
```

```

/*-----*\
  Select a starting plane for the cross section.
/*-----*\
ProMessageDisplay (msgfil, "USER Pick the start plane");
status = ProSelect ("face", 1, NULL, NULL, NULL, NULL, &sel,
    &n_sel);
if (status != PRO_TK_NO_ERROR || n_sel < 1)
    return (0);

ProSelectionModelitemGet (sel[0], &surface_modelitem);
ProSurfaceInit (part, surface_modelitem.id, &surface);
ProSurfaceTypeGet (surface, &stype);
if (stype != PRO_SRF_PLANE)
    return (0);
/*-----*\
  Select a finish point.
/*-----*\
ProMessageDisplay (msgfil, "USER Pick the end position");
status = ProSelect ("face,edge", 1, NULL, NULL, NULL, NULL,
    &sel, &n_sel);
if (status != PRO_TK_NO_ERROR || n_sel < 1)
    return (0);
ProSelectionPoint3dGet (sel[0], finish);
/*-----*\
  Get the geometry of the start plane.
/*-----*\
uv[0] = uv[1] = 0.0;
ProSurfaceXyzdataEval (surface, uv, pos, NULL, NULL, normal);
dist = ProUtilPointPlaneDist (finish, pos, normal);
/*-----*\
  Select a Csys Datum whose X and Y are parallel to the
  start plane.
/*-----*\
ProMessageDisplay (msgfil, "USER Pick the csys");
while (1)
{
    status = ProSelect ("csys", 1, NULL, NULL, NULL, NULL, &sel,
        &n_sel);
    if (status != PRO_TK_NO_ERROR || n_sel < 1)
        return (0);
    ProSelectionModelitemGet (sel[0], &csys_modelitem);
    ProCsysInit (part, csys_modelitem.id, &csys);
    ProCsysDataGet (csys, &csys_geomdata);
    csys_data = csys_geomdata->data.p_csys_data;
    ProUtilVectorCross (normal, csys_data->z_vector, cross);
    if (fabs (ProUtilVectorLength (cross)) > EPSM6)
        ProMessageDisplay (msgfil, "USER Csys has wrong orientation");
    else
        break;
}
/*-----*\

```

```

    Get the name and location of the coordinate system.
/*-----*/
    ProMatrixInit (csys_data->x_vector, csys_data->y_vector,
                  csys_data->z_vector, csys_data->origin, transf);
    ProModelitemNameGet (&csys_modelitem, w_csys_name);
/*-----*/
    Get the number of steps in the analysis.
/*-----*/
    ProMessageDisplay (msgfil, "USER Number of steps [QUIT] : ");
    if (ProMessageIntegerRead (NULL, &n_steps) != PRO_TK_NO_ERROR)
        return (0);
/*-----*/
    Create an initial cross section halfway down.
/*-----*/
    ProStringToWstring (w_name, "ZZZ");
    xsec = prodb_create_parallel_xsec ((Prohandle)part, w_name,
                                       (Prohandle)surface, dist / 2.0, NULL, NULL, &dim_id,
                                       &xsec_status);
/*-----*/
    Step from the start plane to the finish.
/*-----*/
    absdist = fabs(dist);
    for (offset = 0.0; offset <= absdist; offset +=
        absdist / n_steps)
    {
/*-----*/
        Modify the cross-sectional offset dimension to move
        it to the right position.
/*-----*/
        prodim_get_dim ((Prohandle)part, dim_id, &dimension);
        dimension.value = offset;
        prodim_set_dim ((Prohandle)part, dim_id, &dimension);
/*-----*/
        Regenerate the cross section, and calculate the
        mass properties.
/*-----*/
        prodb_regen_xsec ((Prohandle)part, xsec);
        if (prodb_mass_prop_xsec ((Prohandle)part, xsec,
                                w_csys_name, &mprop) == 0)
            continue;
/*-----*/
        Transform the COG to model coordinates.
/*-----*/
        ProPntTrfEval(mprop->center_of_gravity, transf, cog);
/*-----*/
        Shift the cross COG from the XY-plane of the
        coordinate system to the plane of the cross section.
/*-----*/
        xpos[0] = pos[0] - normal[0] * offset;
        xpos[1] = pos[1] - normal[1] * offset;
        xpos[2] = pos[2] - normal[2] * offset;

```



```

        ProUtilPlaneLineX(xpos, normal, cog, normal, cog);
        ProUtilVectorCopy (cog, points[n_points++]);
    }
/*-----*/
    Delete the cross section and the owning feature (a datum
    plane).
/*-----*/
    UsrDimFeatureGet (part, dim_id, &feature);
    prodb_delete_xsec ((Prohandle)part, xsec);
    feat_ids[0] = feature.id;
    opts[0] = PRO_FEAT_DELETE_CLIP;
    ProFeatureDelete (part, feat_ids, 1, opts, 1);
/*-----*/
    Display the line of the COG as a 3D list of vectors.
/*-----*/
    ProDisplist3dCreate(id, UsrPolylineDraw, points,
        &n_points, NULL, NULL, NULL, NULL);
    ProUtilMatrixCopy (NULL, nulltransf);
    ProDisplist3dDisplay (id++, nulltransf);
    ProWindowRepaint (-1);
}
/*=====*/
    Function to draw a polyline, with arguments suitable
    for acting as a callback to ProDisplist3dCreate()
/*=====*/
void UsrPolylineDraw(
    ProPoint3d *points,
    int *n_points)
{
    ProColortype old_color, c;
    ProGraphicsColorSet (PRO_COLOR_WARNING, &old_color);
    ProGraphicsPolylineDraw (points, *n_points);
    ProGraphicsColorSet (old_color, &c);
}
/*=====*/
    Function to get the identifier of the feature to which
    a specified dimension belongs
/*=====*/
int UsrDimFeatureGet(
    ProSolid part,
    int dim_id,
    ProFeature *feature)
{
    Object_element input, *output;

    input.type = SEL_3D_DIMENSION;
    input.id = dim_id;
    input.ptr = NULL;

    if (pro_element_info ((Prohandle)part, &input, PRO_BELONG_TO,

```

```
        -1, &output) <= 0)
        return (0);

    ProModelitemInit (part, output[0].id, PRO_FEATURE, feature);
    return (1);
}
```

12

Core: Utilities

This chapter describes the Pro/TOOLKIT utility functions.

Topic	Page
Configuration Options	12 - 2
Registry File Data	12 - 2
Trail Files	12 - 3
Pro/ENGINEER License Data	12 - 3
Current Directory	12 - 3
File Handling	12 - 4
Wide Strings	12 - 6
Running ModelCHECK	12 - 7

Configuration Options

Functions Introduced:

- **ProConfigoptSet()**
- **ProConfigoptArrayGet()**
- **ProDisplistInvalidate()**

The functions **ProConfigoptArrayGet()** and **ProConfigoptSet()** enable you to retrieve and set the current option for the specified configuration file option, respectively.

To use **ProConfigoptSet()** on a configuration option that affects the display of Pro/ENGINEER, you must call the function **ProDisplistInvalidate()** before you repaint the screen. This function makes sure Pro/ENGINEER invalidates the two- or three-dimensional display list. The calling sequence of functions is as follows:

```
ProConfigoptSet (woption, value);
ProMdlCurrentGet (&model);
ProDisplistInvalidate (model)
ProWindowRepaint (-1);
```

The function **ProConfigoptSet()**, when applied to a multi string configuration option like "search_path", adds a new path entry into the session. It does not affect existing values. When applied to a single-valued configuration option, **ProConfigoptSet()** modifies the value of the configuration option.

The function **ProConfigoptArrayGet()** retrieves the value of a specified configuration file option. The function returns an array of values assigned to the configuration file. It returns a single value if the configuration file option is not a multi-valued option. This function supersedes the function **ProConfigoptGet()**.

Registry File Data

Functions Introduced:

- **ProToolkitAppExecPathGet()**
- **ProToolkitAppITextPathGet()**

The function **ProToolkitAppExecPathGet()** returns the path to the Pro/TOOLKIT executable file (`exec_file`) from the registry file.

The function **ProToolkitApplTextPathGet()** returns the path to the directory containing the "text" directory for the application.

Trail Files

Function Introduced:

- **ProTrailfileCommentWrite()**

To append a comment to the end of the current trail file, call the function **ProTrailfileCommentWrite()**. The comment should not be longer than (PRO_COMMENT_SIZE - 2) characters, and should not contain any nonprintable characters, such as "\n."

Pro/ENGINEER License Data

Function Introduced:

- **ProOptionOrderedVerify()**

The function **ProOptionOrderedVerify()** reports whether a specified Pro/ENGINEER license option such as Pro/MESH) is currently available in the Pro/ENGINEER session.

Current Directory

Functions Introduced;

- **ProDirectoryCurrentGet()**
- **ProDirectoryChange()**

These two functions are concerned with the current default directory in Pro/ENGINEER—the one in which it searches when you retrieve an object, for example. The Pro/ENGINEER user changes this directory using the command **File>Working Directory**.

The function **ProDirectoryChange()** enables you to do the exact equivalent of **File, Working Directory** in Pro/ENGINEER. Use this function if you need to save and retrieve objects in a directory other than the one the user chose.

The function **ProDirectoryCurrentGet()** returns the whole path to the directory, as a wide string.

File Handling

Functions Introduced:

- **ProFilesList()**
- **ProFileOpen()**
- **ProFileOpenRegister()**
- **ProFileSave()**
- **ProFileSaveRegister()**
- **ProDirectoryChoose()**
- **ProFilenameParse()**
- **ProPathCreate()**
- **ProInfoWindowDisplay()**
- **ProFileEdit()**
- **ProTexturePathGet()**

The function **ProFilesList()** provides a list of the contents of a directory, given the directory path. You can filter the list to include only files of a particular type, as specified by the file extension. Use the `PRO_FILE_LIST_ALL` option to include all versions of a file in the list; use `PRO_FILE_LIST_LATEST` to include only the latest version. In addition to an array of file names, the function returns an array of subdirectory names, regardless of the filter used.

Starting with Pro/ENGINEER Wildfire 5.0, the function **ProFilesList()** can also list instance objects when accessing Windchill workspaces or folders. A PDM location (for workspace or commonspace) must be passed as the directory path. The following options have been added in the *ProFileListOpt* enumerated type that can be passed as the *listing_option* argument to **ProFilesList()**:

- `PRO_FILE_LIST_ALL_INST`—Same as the `LIST_ALL` option. It returns instances only for PDM locations.
- `PRO_FILE_LIST_LATEST_INST`—Same as the `LIST_LATEST` option. It returns instances only for PDM locations.
- `PRO_FILE_LIST_ALL_SORTED_INST`—Same as the `LIST_ALL_SORTED` option. It returns instances only for PDM locations.

- **PRO_FILE_LIST_LATEST_SORTED_INST**—Same as the **LIST_LATEST_SORTED** option. It returns instances only for PDM locations.

The function **ProFileOpen()** opens the relevant Pro/ENGINEER dialog box for opening files and browsing directories. Note that this function uses the same filtering method as **ProFilesList()**. The **ProFileOpen()** function lets you specify the title of the dialog box, a set of shortcuts to other directories, and the name of a file to be preselected.

Note: Despite its name, this function does not actually open the file, but returns the file path. For example, to open a text file, use the function **ProFileEdit()** or **ProInfoWindowDisplay()**.

The function **ProFileOpenRegister()** registers a new file type in the **File>Open** dialog box in Pro/ENGINEER. This function takes the access function **ProFileOpenAccessFunction()** and the action function **ProFileOpenOperationAction()** as its input arguments.

The function **ProFileOpenAccessFunction()** is called to determine whether the new file type can be opened using the **File>Open** dialog box. The function **ProFileOpenOperationAction()** is called on clicking **Open** for the newly registered file type.

The function **ProFileSave()** opens the relevant Pro/ENGINEER dialog box for saving files. This function has input arguments similar to **ProFileOpen()**. The function returns the path to the saved file as output.

The function **ProFileSaveRegister()** registers a new file type in the **File>Save a Copy** dialog box in Pro/ENGINEER. This function takes the access function **ProFileSaveAccessFunction()** and the action function **ProFileSaveOperationAction()** as its input arguments.

The function **ProFileSaveAccessFunction()** is called to determine whether the new file type can be saved using the **File>Save a Copy** dialog box. The function **ProFileSaveOperationAction()** is called on clicking **OK** for the newly registered file type.

The function **ProDirectoryChoose()** prompts the user to select a directory using the Pro/ENGINEER dialog box for browsing directories. Specify the title of the dialog box, a set of shortcuts to other directories, and the default directory path to start browsing. If the default path is specified as null, the current directory is used. The function returns the selected directory path as output.

In general, the file utility functions refer to files using a single wide character string, which contains four, distinct pieces of information that uniquely identify the file: the directory path, file name, extension, and version. The function **ProFilenameParse()** takes such a string as input, and returns the four segments as separate arguments. The function **ProPathCreate()** performs the opposite operation—it builds the single wide string that identifies the file, given the path, file name, extension, and version.

The function **ProInfoWindowDisplay()** creates a text information window. It reads the contents from a text file in the current directory whose name is an input to the function. The function can also override the default size, shape, and location of the window. (These do not affect the properties of the Pro/ENGINEER Information Window.)

The function **ProFileEdit()** opens an edit window on a specified text file. The editor used is the current default editor for Pro/ENGINEER.

The function **ProTexturePathGet()** looks for the full path to the specified texture, decal, or bump map files and loads them from the texture path.

Note: For textures embedded inside a Pro/ENGINEER model, if the `create_temp_file` is set to `true` the **ProTexturePathGet()** function writes a temporary copy of the specified files.

Wide Strings

Functions Introduced:

- **ProStringToWstring()**
- **ProWstringToString()**
- **ProWcharSizeVerify()**

These three utilities are described in the section Wide Strings in the 'Fundamentals' chapter.

Freeing String Outputs

Many Pro/TOOLKIT functions provide outputs of non-fixed length strings or wide strings. These outputs must be freed using a special set of functions, because they have been allocated by a special function internally.

Note: These functions must be only used for strings and string arrays output from Pro/TOOLKIT functions. Check the function description to determine the function to use when freeing the output.

Functions Introduced:

- **ProStringFree()**
- **ProWstringFree()**
- **ProStringarrayFree()**
- **ProWstringarrayFree()**
- **ProStringproarrayFree()**
- **ProWstringproarrayFree()**

Use the functions **ProStringFree()** and **ProWstringFree()** to free a single *char** or *wchar_t** output from a Pro/TOOLKIT function.

Use the functions **ProStringarrayFree()** and **ProWstringarrayFree()** to free a standard array of *char** or *wchar_t** output from a Pro/TOOLKIT function.

Use the functions **ProStringproarrayFree()** and **ProWstringproarrayFree()** to free a *ProArray* of *char** or *wchar_t** output from a Pro/TOOLKIT function.

Running ModelCHECK

ModelCHECK is an integrated application that runs transparently within Pro/ENGINEER. ModelCHECK uses a configurable list of company design standards and best modeling practices. You can configure ModelCHECK to run interactively or automatically when you regenerate or save a model.

Functions Introduced:

- **ProModelcheckExecute()**

You can execute ModelCHECK from an external application using the function **ProModelcheckExecute()**. The input parameters of this function are:

- *mdl*—Specifies the model on which you want to execute ModelCHECK.
- *show_ui*—Specifies True to display the ModelCHECK report in the Web browser.
- *mcMode*—Specifies the mode in which you want to run ModelCHECK. The modes are:
 - PRO_MODELCHECK_GRAPHICS—Interactive mode
 - PRO_MODELCHECK_NO_GRAPHICS—Batch mode
- *config_dir*—Specifies the location of the configuration files. If this parameter is set to NULL, the default ModelCHECK configuration files are used.
- *output_dir*—Specifies a location for the reports. If this parameter is set to NULL, the default ModelCHECK output directory is used.

The output parameters of this function are:

- *errors*—Specifies the number of errors found.
- *warnings*—Specifies the number of warnings found.
- *model_saved*—True if the model is saved with updates, else false.

Creating Custom Checks

This section describes how to define custom checks in ModelCHECK that users can run using the standard ModelCHECK interface in Pro/ENGINEER.

To define and register a custom check:

1. Set the CUSTMTK_CHECKS_FILE configuration option in the start configuration file to a text file that stores the check definition. For example:

```
CUSTMTK_CHECKS_FILE text/custmtk_checks.txt
```
2. Set the contents of the CUSTMTK_CHECKS_FILE file to define the checks. Each check should list the following items:
 - DEF_<checkname>—Specifies the name of the check. The format must be CHKTK_<checkname>_<mode>, where mode is PRT, ASM, or DRW.

- **TAB_<checkname>**—Specifies the tab (category) in the ModelCHECK report under which the check is classified. Valid tab values are:
 - INFO
 - PARAMETER
 - LAYER
 - FEATURE
 - RELATION
 - DATUM
 - MISC
 - VDA
 - VIEWS
- **MSG_<checkname>**—Specifies the description of the check that appears in the lower part of the ModelCHECK report when you select the name.
- **DSC_<checkname>**—Specifies the name of the check as it appears in the ModelCHECK report table.
- **ERM_<checkname>**—If set to INFO, the check is considered an INFO check and the report table will display the text from the first item returned by the check, instead of a count of the items. Otherwise, this value must be included, but is ignored by Pro/ENGINEER.

See Example 1: Text File for Custom Checks for a sample custom checks text file.

3. Add the check and its values to the ModelCHECK configuration file. For an example of how this is done, see the sample code at the end of this section.
4. Register the ModelCHECK check from the Pro/TOOLKIT application.

Note: Other than the requirements listed above, Pro/TOOLKIT custom checks do not have access to the rest of the values in the ModelCHECK configuration files. All the custom settings specific to the check such as start parameters, constants, and so on, must be supported by the user application and not ModelCHECK.

Functions Introduced

- **ProModelcheckCheckRegister()**
- **ProModelcheckCheckFunction()**
- **ProModelcheckUpdateFunction()**
- **ProModelcheckCleanupFunction()**

The function **ProModelcheckCheckRegister()** registers the callback functions for each custom check. The following arguments are available:

- The custom check name. This name must match the name given to the check in the ModelCheck configuration file, and must begin with "CHKTK_"
- The check label. Currently unused; the label is taken from the check configuration file.
- The check options. Currently unused, specify the value as NULL.
- The check callback functions, described in details below.
- Labels for the action and update buttons, or specify NULL if these buttons should not be shown.
- Application data to pass to the callback functions.

The check callback functions are as follows:

- The check function (required)—This function, whose signature matches **ProModelcheckCheckFunction()**, should calculate the results of the check and provide them to ModelCHECK through the function output arguments.
- The action and update functions (optional)—These functions are called when users choose the action or update button when viewing the ModelCHECK report. These have the signature of the **ProModelcheckUpdateFunction()** function.
- The cleanup function (optional)—Gives your application a chance to free memory allocated by the check function. This has the signature of the **ProModelcheckCleanupFunction()** function.

Each callback function receives the following inputs:

- The name of the custom check, as defined in the original call to **ProModelcheckCheckRegister()**.
- The model being checked.

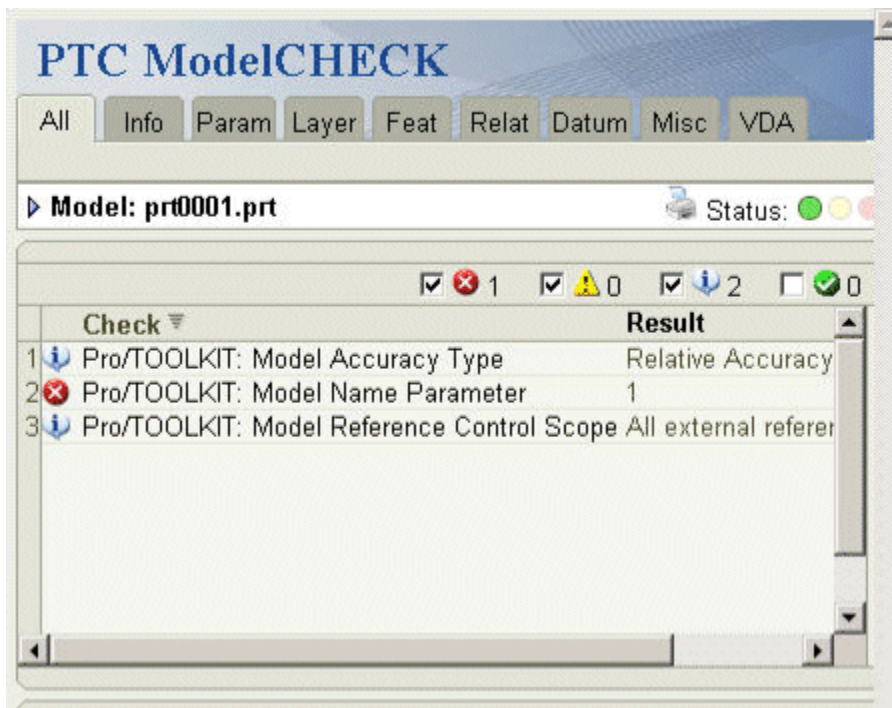
- A pointer to the application data, set during the call to **ProModelcheckCheckRegister()**.

The function whose prototype matches

ProModelcheckCheckFunction() is used to evaluate the custom defined checks. The user application runs the check on the specified model and populates the following output arguments:

- *results_count*—Specifies an integer indicating the number of errors found by the check.
- *results_url*—Specifies the link to an application-owned page that contains information specific to the check.
- *results_table*—Specifies an array of data for each error encountered that will be shown along with the results.

The following figure illustrates how the results of some custom checks might be displayed in the ModelCHECK report.



The function whose prototype matches

ProModelcheckCleanupFunction() is used to release memory allocated for the check callback function.

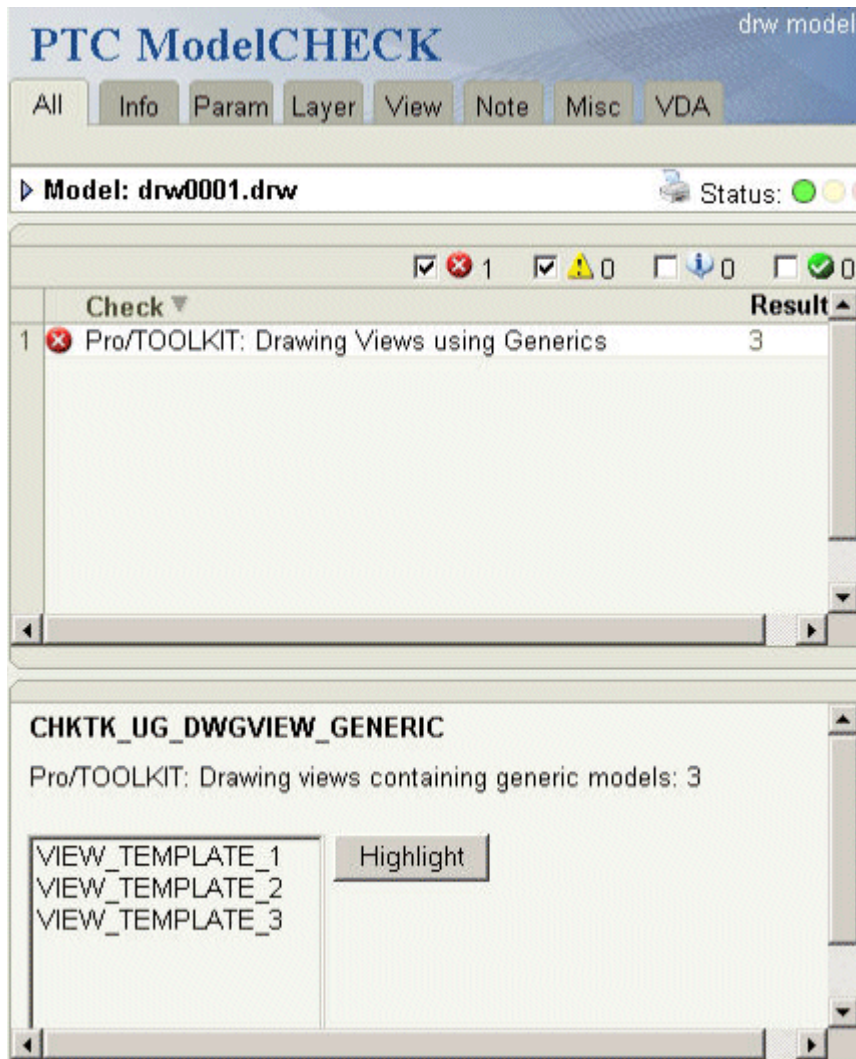
The functions whose prototypes matches

ProModelcheckUpdateFunction() are used for the following:

- To execute a check-defined action on a given item.
- To automatically update a given item to fix errors, if any.

The selected item's description string is passed as an argument to the update function.

ModelCHECK checks may have one "action" function to highlight the problem, and possibly an update function, to fix it automatically. The following figure displays the ModelCHECK report with an action button that invokes the action callback function.



Core: Utilities

Example 1: Text File for Custom Checks

The following is the text file for the custom check examples.

```

UG CustomCheck: MDL PARAM NOT FOUND
Parameter %0w is not found in the model %1w
#
#
UG CustomCheck: MDL PARAM OK
Parameter %0w is set correctly in the model %1w
#
#

```

```

UG CustomCheck: MDL PARAM INV TYPE
Parameter %0w in %1w is not a String parameter
#
#
UG CustomCheck: MDL PARAM INCORRECT
Parameter %0w value does not match model name %1w
#
#
UG CustomCheck: MDL PARAM NAME
Parameter %0w value matches model name
#
#
UG CustomCheckUpdate: MDL PARAM NAME
Fix Parameter
#
#
%CIUG CustomCheck: MDL PARAM UPDATED
Mdl name parameter %0w has been updated.
#
#
%CEUG CustomCheck: MDL PARAM UPDATE TYPE
Cannot modify the type of parameter %0w; this error should be fixed
manually.
#
#
UG CustomCheck: MDL ACC TYPE
Model accuracy
#
#
UG CustomCheck: MDL ACC ABS
Absolute Accuracy
#
#
UG CustomCheck: MDL ACC REL
Relative Accuracy
#
#
UG CustomCheck: DWGVIEW GENERIC
Drawing Views using Generics
#
#
UG CustomCheckAction: DWGVIEW GENERIC
Highlight
#
#

```

Example 2: Registering Custom ModelCheck Checks

The following example demonstrates how to register custom ModelCheck checks using Pro/TOOLKIT. The following custom checks are registered:

- CHKTK_MDL_NAME_PARAM—Determines if the model has a parameter whose value is equal to the model name
- CHKTK_MDL_REFC_SCOPE—Info check that prints the model reference control settings
- CHKTK_DWVIEW_GENERIC—Drawing mode check that looks for views that use generic models

```
#include <ProToolkit.h>
#include <ProMdlChk.h>
#include <ProParameter.h>
#include <ProSolid.h>
#include <ProDrawing.h>
#include <ProFamtable.h>
#include <ProSelection.h>

/*=====*\
  FUNCTION: UserCustomModelChecksDefine
  PURPOSE:  Registers all of the custom Pro/TOOLKIT ModelCheck checks.
\*=====*/
int UserCustomModelChecksDefine()
{
    ProFileName message_file;
    ProCharName model_check_name;
    ProLine check_label, action_label, update_label;
    static ProName param_name;
    ProError status;

/*-----*\
  Prepare the button labels
\*-----*/

/* Parameter name that should contain the model owner name */
ProStringToWstring (param_name, "MDL_NAME_PARAM");
ProStringToWstring (message_file, "pt_ug_modelcheck.txt");

/* The name of the check. */
strcpy(model_check_name, "CHKTK_UG_MDLPARAM_NAME");

/* The label for the check */
status = ProMessageToBuffer (check_label, message_file,
    "UG CustomCheck: MDL PARAM NAME", param_name);

/* Label for the button used to update the model for an
   item found by the check */
status = ProMessageToBuffer (update_label, message_file,
    "UG CustomCheckUpdate: MDL PARAM NAME");

/*-----*\
  Register the custom ModelCheck check.  This check allows update
  of the incorrect situation.
\*-----*/
    status = ProModelcheckCheckRegister (model_check_name, check_label,
```

```

        NULL,
        UserCustCheckMdlParamName,
        UserCustCheckCleanUtility,
        NULL,
        NULL,
        update_label,
        UserCustUpdateMdlParamName,
        (ProAppData) &param_name);
/*-----*\
Prepare the button labels
\*-----*/
    /* The name of the check. */
    strcpy(model_check_name, "CHKTK_UG_MDL_REFC_SCOPE");

    /* The label for the check */
    status = ProMessageToBuffer (check_label, message_file,
                                "UG CustomCheck: MDL REFC SCOPE");

/*-----*\
Register the custom ModelCheck check (an info check, so no
action or update is possible).
\*-----*/
    status = ProModelcheckCheckRegister (model_check_name, check_label,
                                         NULL, UserCustCheckRefScope,
                                         UserCustCheckCleanUtility,
                                         NULL, NULL, NULL, NULL, NULL);
/*-----*\
Prepare the button labels
\*-----*/
    /* The name of the check. */
    strcpy(model_check_name, "CHKTK_UG_DWGVIEW_GENERIC");
    /* The label for the check */
    status = ProMessageToBuffer (check_label, message_file,
                                "UG CustomCheck: DWGVIEW GENERIC",
                                param_name);

    /* Label for the button used to update the model for an
       item found by the check */
    status = ProMessageToBuffer (action_label, message_file,
                                "UG CustomCheckAction: DWGVIEW GENERIC");
/*-----*\
Register the custom ModelCheck check; this check has a Highlight
action that users can use.
\*-----*/
    status = ProModelcheckCheckRegister (model_check_name, check_label,
                                         NULL,
                                         UserCustCheckDwgviewGeneric,
                                         UserCustCheckCleanUtility,
                                         action_label,
                                         UserCustActionDwgviewGeneric,
                                         NULL,

```

```

        NULL,
        NULL);
    return (PRO_TK_NO_ERROR);
}

```

Example 3: Implementing a Model Name Parameter Check

The following functions are the implementation for the model name parameter check. This check has a check function callback and an update function callback. Also included is the cleanup callback used for all of the custom checks.

```

static wchar_t** s_results_table = NULL;
static wchar_t* s_results_url = NULL;

#define MISSING_PARAM (int)999
#define INVALID_PARAM_TYPE (int)9999

/*=====*\
FUNCTION: UserCheckMdlParamNameCompare
PURPOSE: Compare the model name to the value of the given param.
RETURNS: MISSING_PARAM, INVALID_PARAM_TYPE, or the integer results of
the string comparison.
\*=====*/
static int UserCheckMdlParamNameCompare (ProMdl mdl, ProName param_name)
{
    ProError status = PRO_TK_NO_ERROR;
    ProName mdl_name;
    ProModelitem modelitem;
    ProParamvalue param_value;
    ProParameter param;
    int compare_result;

    status = ProMdlNameGet (mdl, mdl_name);

    status = ProMdlToModelitem(mdl, &modelitem);

    status = ProParameterInit(&modelitem, param_name, &param);

/*-----*\
    Parameter is missing
\*-----*/
    if (status != PRO_TK_NO_ERROR)
    {
        compare_result = MISSING_PARAM;
    }
    else
    {
        status = ProParameterValueGet (&param, &param_value);
    }
}

```

```

/*-----*\
    Parameter is not a string parameter
\*-----*/
    if (status != PRO_TK_NO_ERROR || param_value.type != PRO_PARAM_STRING)
    {
        compare_result = INVALID_PARAM_TYPE;
    }
    else
    {
/*-----*\
        Compare the model name to the parameter value.
\*-----*/
        ProWstringCompare (param_value.value.s_val, mdl_name,
                           PRO_VALUE_UNUSED, &compare_result);
    }
}

return (compare_result);
}
/*=====*\
    FUNCTION: UserCustCheckMdlParamName
    PURPOSE:  Check function for the ModelCheck check. Outputs details of
               the comparison as a ModelCheck error.
\*=====*/
static ProError UserCustCheckMdlParamName (ProCharName name, ProMdl mdl,
                                           ProAppData appdata,
                                           int* results_count,
                                           wchar_t** results_url,
                                           wchar_t*** results_table)

{
    ProFileName message_file;
    ProError status = PRO_TK_NO_ERROR;
    ProName* param_name_ptr = (ProName*)appdata;
    ProName mdl_name;
    int compare_result = UserCheckMdlParamNameCompare (mdl,
                                                         *param_name_ptr);

    ProLine error_msg;

/*-----*\
    Parameter is set correctly.
\*-----*/
    if (compare_result == 0)
    {
        *results_count = 0;
        *results_url = NULL;
        *results_table = NULL;
        return (PRO_TK_NO_ERROR);
    }
}

```

```

/*-----*\
    Parameter is not set correctly. Show an appropriate error message.
\*-----*/
else
{
    ProStringToWstring (message_file, "pt_ug_modelcheck.txt");
    status = ProMdlNameGet (mdl, mdl_name);

    if (compare_result == MISSING_PARAM)
    {
        status = ProMessageToBuffer (error_msg, message_file,
                                     "UG CustomCheck: MDL PARAM NOT FOUND",
                                     *param_name_ptr, mdl_name);
    }
    else if (compare_result == INVALID_PARAM_TYPE)
    {
        status = ProMessageToBuffer (error_msg, message_file,
                                     "UG CustomCheck: MDL PARAM INV TYPE",
                                     *param_name_ptr, mdl_name);
    }
    else
    {
        status = ProMessageToBuffer (error_msg, message_file,
                                     "UG CustomCheck: MDL PARAM INCORRECT",
                                     *param_name_ptr, mdl_name);
    }

    status = ProArrayAlloc(1, sizeof(ProWstring), 1,
                           (ProArray*)&s_results_table);
    s_results_table [0] = (wchar_t*) calloc (PRO_LINE_SIZE,
                                              sizeof (wchar_t));
    ProWstringCopy (error_msg, s_results_table[0], PRO_VALUE_UNUSED);

/*-----*\
    This URL will be used for the "Check Details" link in the ModelCheck
    report.
\*-----*/
    s_results_url = (wchar_t*) calloc (PRO_PATH_SIZE, sizeof (wchar_t));
    ProStringToWstring (s_results_url, "http://www.ptc.com/");
    *results_table = s_results_table;
    *results_count = 1;
    *results_url = s_results_url;
    return (PRO_TK_NO_ERROR);
}

/*=====*\
    FUNCTION: UserCustCheckCleanUtility
    PURPOSE:  Cleanup function for all of the ModelCheck checks.
\*=====*/
    static ProError UserCustCheckCleanUtility (ProCharName name,
                                              ProMdl mdl,

```

```

ProAppData appdata)
{
    int size, i;
    ProError status = PRO_TK_NO_ERROR;
    if (s_results_table != NULL)
    {
        status = ProArraySizeGet (s_results_table, &size);

        if (status == PRO_TK_NO_ERROR)
        {
            for (i = 0; i < size; i++)
                free (s_results_table [i]);

            ProArrayFree ((ProArray*)&s_results_table);
        }
        s_results_table = NULL;
    }

    if (s_results_url != NULL)
    {
        free (s_results_url);
        s_results_url = NULL;
    }

    return (PRO_TK_NO_ERROR);
}

/*=====*\
FUNCTION: UserCustUpdateMdlParamName
PURPOSE: Update function for a ModelCheck error due to an invalid or
missing model name parameter.
\*=====*/
static ProError UserCustUpdateMdlParamName (ProCharName name,
                                           ProMdl mdl,
                                           wchar_t* selected_item,
                                           ProAppData appdata)
{
    ProFileName message_file;
    ProError status = PRO_TK_NO_ERROR;
    ProName* param_name_ptr = (ProName*)appdata;
    ProName mdl_name;
    int compare_result = UserCheckMdlParamNameCompare (mdl,
                                                         *param_name_ptr);

    ProModelitem modelitem;
    ProParamvalue param_value;
    ProParameter param;

    if (compare_result == 0)
    {
        /* Nothing to do */
        return (PRO_TK_NO_ERROR);
    }
    else

```

```

    {
        ProStringToWstring (message_file, "pt_ug_modelcheck.txt");
        status = ProMdlNameGet (mdl, mdl_name);
        param_value.type = PRO_PARAM_STRING;
        ProWstringCopy (mdl_name, param_value.value.s_val,
                        PRO_VALUE_UNUSED);

        status = ProMdlToModelitem (mdl, &modelitem);

/*-----*\
    Create the missing parameter with the correct value.
/*-----*\
        if (compare_result == MISSING_PARAM)
        {
            ProParameterCreate (&modelitem, *param_name_ptr, &param_value,
                                &param);
            status = ProMessageDisplay (message_file,
                                        "UG CustomCheck: MDL PARAM UPDATED",
                                        *param_name_ptr);
        }

/*-----*\
    Since there is no way to change a parameter's type except by deleting and
    recreating it, the check will not attempt to repair this situation.
    Instead, show a message to the user.
/*-----*\
        else if (compare_result == INVALID_PARAM_TYPE)
        {
            status = ProMessageDisplay (message_file,
                                        "UG CustomCheck: MDL PARAM UPDATE TYPE",
                                        *param_name_ptr);
        }

/*-----*\
    Change the value of the parameter appropriately.
/*-----*\
        else
        {
            ProParameterInit (&modelitem, *param_name_ptr, &param);
            ProParameterValueSet (&param, &param_value);
            status = ProMessageDisplay (message_file,
                                        "UG CustomCheck: MDL PARAM UPDATED",
                                        *param_name_ptr);
        }

    return (PRO_TK_NO_ERROR);
}
}

```

Example 4: Implementing a Reference Control Info Check

The following function is the implementation for an info check that will report on the reference control setting applied to the model. This check has a check function callback but no action or update function (because it is an info-only check).

```
/*=====*\
FUNCTION: UserCustCheckRefScope
PURPOSE: Check function for the ModelCheck check for reference scope.
Outputs the scope permitted for external refs.
\*=====*/
static ProError UserCustCheckRefScope (ProCharName name, ProMdl mdl,
                                       ProAppData appdata,
                                       int* results_count,
                                       wchar_t** results_url,
                                       wchar_t*** results_table)
{
    ProFileName message_file;
    ProError status = PRO_TK_NO_ERROR;
    ProExtRefScope scope;
    ProInvalidRefBehavior behavior;
    ProLine info_msg;
    ProCharLine message_key;

    ProStringToWstring (message_file, "pt_ug_modelcheck.txt");

/*-----*\
    Output the model ref control level as an info check.
\*-----*/
    status = ProRefCtrlSolidGet (mdl, &scope, &behavior);

    switch (scope)
    {
    case PRO_REFCTRL_ALLOW_ALL:
        strcpy (message_key, "UG CustomCheck: MDL REFC ALL");
        break;
    case PRO_REFCTRL_ALLOW_SUBASSEMBLY:
        strcpy (message_key, "UG CustomCheck: MDL REFC SUB");
        break;
    case PRO_REFCTRL_ALLOW_SKELETON:
        strcpy (message_key, "UG CustomCheck: MDL REFC SKEL");
        break;
    case PRO_REFCTRL_ALLOW_NONE:
        strcpy (message_key, "UG CustomCheck: MDL REFC NONE");
        break;

    default:
        return (PRO_TK_UNSUPPORTED);
    }
}
```



```

        status = ProMessageToBuffer (info_msg, message_file,
                                     message_key);

        status = ProArrayAlloc(1, sizeof(ProWstring), 1,
                               (ProArray*)&s_results_table);

        s_results_table[0] = (wchar_t*) calloc (PRO_LINE_SIZE,
                                                sizeof (wchar_t));
        ProWstringCopy (info_msg, s_results_table[0], PRO_VALUE_UNUSED);

        *results_table = s_results_table;
        *results_count = 1;
        *results_url = NULL;

        return (PRO_TK_NO_ERROR);
}

```

Example 5: Implementing a Check Looking for Drawing Views Using Generics

The following functions are the implementation for the drawing view using generics check. This check has a check function callback and an action function used to highlight the views that have this error.

```

/*=====*\
FUNCTION: UserCustCheckDwgviewGeneric
PURPOSE: Check function for the ModelCheck check for generics in
drawing views. Outputs a list of the view names.
\*=====*/
static ProError UserCustCheckDwgviewGeneric (ProCharName name, ProMdl
                                             mdl,
                                             ProAppData appdata,
                                             int* results_count,
                                             wchar_t** results_url,
                                             wchar_t*** results_table)
{
    ProFileName message_file;
    ProError status = PRO_TK_NO_ERROR;
    ProView* views;
    int i, size;
    ProSolid solid;
    ProFamtable fam_table;
    wchar_t* table_entry;
    ProStringToWstring (message_file, "pt_ug_modelcheck.txt");

    /*-----*\
        Check the model shown by each view
    \*-----*/
    status = ProDrawingViewsCollect ((ProDrawing)mdl, &views);

```

```

    if (status == PRO_TK_NO_ERROR)
    {
        status = ProArraySizeGet (views, &size);
        if (status == PRO_TK_NO_ERROR && size > 0)
        {
            status = ProArrayAlloc(0, sizeof(ProWstring), 1,
                                   (ProArray*)&s_results_table);
            for (i = 0; i < size; i++)
            {
                status = ProDrawingViewSolidGet ((ProDrawing)mdl,
                                                  views[i], &solid);
            }
        }
    }

/*-----*\
    If ProFamtableCheck() succeeds, this means that the model has a family
    table with at least one instance or item in it.
/*-----*/

    status = ProFamtableInit (solid, &fam_table);
    status = ProFamtableCheck (&fam_table);

    if (status == PRO_TK_NO_ERROR)
    {
        table_entry = (wchar_t*) calloc (PRO_NAME_SIZE,
                                          sizeof (wchar_t));
        ProDrawingViewNameGet ((ProDrawing)mdl, views
                               [i], table_entry);

        ProArrayObjectAdd ((ProArray*)&s_results_table, -1, 1,
                           &table_entry);
    }
}

/*-----*\
    If no items were found...
/*-----*/

    status = ProArraySizeGet (s_results_table, &size);
    if (size == 0)
    {
        ProArrayFree ((ProArray*)&s_results_table);
        s_results_table = NULL;
    }
}

*results_table = s_results_table;
*results_count = size;
*results_url = NULL;

return (PRO_TK_NO_ERROR);
}

/*=====*\
    FUNCTION: UserDwgviewFilterByName
    PURPOSE:  Filter function to find drawing views by matching the input
    name.
/*=====*/

```

```

static ProError UserDwgviewFilterByName (ProDrawing drawing, ProView view,
                                         ProAppData app_data)
{
    ProError status = PRO_TK_NO_ERROR;
    ProName name;
    wchar_t* target_name = (wchar_t*)app_data;
    int compare_result;

    status = ProDrawingViewNameGet (drawing, view, name);

    ProWstringCompare (name, target_name, PRO_VALUE_UNUSED,
&compare_result);
    if (compare_result == 0)
        return (PRO_TK_NO_ERROR);
    else
        return (PRO_TK_CONTINUE);
}
#define DELTA_X 10.0
#define DELTA_Y 10.0
/*=====*\
    FUNCTION: UserDwgviewHighlight
    PURPOSE:  Action function to highlight the drawing view.
\*=====*/
static ProError UserDwgviewHighlight (ProDrawing drawing, ProView view,
                                      ProError filter_status,
                                      ProAppData app_data)
{
    ProError status = PRO_TK_NO_ERROR;
    int sheet;
    int win_id;
    ProPoint3d outline [2];
    ProPoint3d points [5];
    ProColorType old_color;
    ProLineStyle old_ls;

    status = ProDrawingViewSheetGet (drawing, view, &sheet);
    status = ProDrawingCurrentSheetSet (drawing, sheet);
    status = ProWindowCurrentGet (&win_id);
    ProWindowRefresh (win_id);

    /*-----*\
        Draw the outline of the view to highlight it.
    \*-----*/
    status = ProDrawingViewOutlineGet (drawing, view, outline);

    points[0][0] = outline[0][0] - DELTA_X;
    points[0][1] = outline[0][1] - DELTA_Y;
    points[0][2] = 0.0;
    points[1][0] = outline[0][0] - DELTA_X;
    points[1][1] = outline[1][1] + DELTA_Y;
    points[1][2] = 0.0;

```

```

points[2][0] = outline[1][0] + DELTA_X;
points[2][1] = outline[1][1] + DELTA_Y;
points[2][2] = 0.0;
points[3][0] = outline[1][0] + DELTA_X;
points[3][1] = outline[0][1] - DELTA_Y;
points[3][2] = 0.0;
points[4][0] = outline[0][0] - DELTA_X;
points[4][1] = outline[0][1] - DELTA_Y;
points[4][2] = 0.0;

ProLinestyleSet (PRO_LINESTYLE_PHANTOM, &old_ls);
ProGraphicsColorSet (PRO_COLOR_HIGHLIGHT, &old_color);

ProGraphicsPolylineDraw(points, 5);

ProGraphicsColorSet (old_color, NULL);
ProLinestyleSet (old_ls, NULL);

return (PRO_TK_E_FOUND);
}

/*=====*\
FUNCTION: UserCustActionDwgviewGeneric
PURPOSE: Action function for the ModelCheck check that locates drawing
views using generics.
\*=====*/
static ProError UserCustActionDwgviewGeneric (ProCharName name,
                                              ProMdl mdl,
                                              wchar_t* selected_item,
                                              ProAppData appdata)
{
    ProFileName message_file;
    ProError status = PRO_TK_NO_ERROR;

    ProStringToWstring (message_file, "pt_ug_modelcheck.txt");

    ProDrawingViewVisit ((ProDrawing)mdl, UserDwgviewHighlight,
                        UserDwgviewFilterByName,
                        (ProAppData)selected_item);

    return (PRO_TK_NO_ERROR);
}

```

Example 6: Changes to the ModelCheck Configuration Files to enable Custom Checks

The following show the changes made to the ModelCheck configuration files to enable the custom checks.

```

Lines from the custom TK checks file (custmtk_checks.txt)
# Custom TK Check File

```

```

#
# UG_MDLPARAM_NAME
DEF_UG_MDLPARAM_NAME CHKTK_UG_MDLPARAM_NAME_PRT
TAB_UG_MDLPARAM_NAME PARAMETER
MSG_UG_MDLPARAM_NAME Pro/TOOLKIT: Models with missing or invalid model
name parameters:
ERM_UG_MDLPARAM_NAME N/A
DSC_UG_MDLPARAM_NAME Pro/TOOLKIT: Model Name Parameter
# UG_MDL_REFC_SCOPE
DEF_UG_MDL_REFC_SCOPE
CHKTK_UG_MDL_REFC_SCOPE_PRT
TAB_UG_MDL_REFC_SCOPE INFO
MSG_UG_MDL_REFC_SCOPE Pro/TOOLKIT: Model reference control scope:
ERM_UG_MDL_REFC_SCOPE INFO
DSC_UG_MDL_REFC_SCOPE Pro/TOOLKIT: Model reference control scope:

# UG_DWGVIEW_GENERIC
DEF_UG_DWGVIEW_GENERIC CHKTK_UG_DWGVIEW_GENERIC_DRW
TAB_UG_DWGVIEW_GENERIC VIEWS
MSG_UG_DWGVIEW_GENERIC Pro/TOOLKIT: Drawing views containing generic
models:
ERM_UG_DWGVIEW_GENERIC N/A
DSC_UG_DWGVIEW_GENERIC Pro/TOOLKIT: Drawing Views using Generics

Lines added to the ModelCheck configuration file (default_checks.mch)
CHKTK_UG_MDLPARAM_NAME_PRT      YNEW      E E E E Y
CHKTK_UG_MDL_REFC_SCOPE_PRT     YNEW      Y Y Y Y Y
CHKTK_UG_DWGVIEW_GENERIC_DRW    YNEW      E E E E Y

Lines added to the ModelCheck start file (nostart.mcs)
CUSTMTK_CHECKS_FILE      text/custmtk_checks.txt

```


13

Core: Asynchronous Mode

This chapter explains using Pro/TOOLKIT in Asynchronous Mode.

Topic	Page
Overview	13 - 2
Simple Asynchronous Mode	13 - 3
Full Asynchronous Mode	13 - 9

Overview

Asynchronous mode is a multiprocess mode in which the Pro/TOOLKIT application and Pro/ENGINEER can perform concurrent operations. Unlike synchronous mode, the asynchronous mode uses remote procedure calls (rpc) as the means of communication between the application and Pro/ENGINEER.

Another important difference between synchronous and asynchronous modes is in the startup of the Pro/TOOLKIT application. In synchronous mode, the application is started by Pro/ENGINEER, based on information contained in the registry file. In asynchronous mode, the application (containing its own *main()* function) is started independently of Pro/ENGINEER and subsequently either starts or connects to a Pro/ENGINEER process. Note that an asynchronous application will not appear in the Auxiliary Applications dialog box.

The section How Pro/TOOLKIT Works in 'Fundamentals' chapter describes two modes—DLL and multiprocess (or “spawned”). These modes are synchronous modes in the sense that the Pro/TOOLKIT application and Pro/ENGINEER do not perform operations concurrently. In spawn mode, each process can send a message to the other to ask for some operation, but each waits for a returning message that reports that the operation is complete. Control alternates between the two processes, one of which is always in a wait state.

Asynchronous mode applications operate with the same method of communication as spawn mode (multiprocess). The use of rpc in spawn mode causes this mode to perform significantly slower than DLL communications. For this reason, you should be careful not to apply asynchronous mode when it is not needed. Note that asynchronous mode is not the only mode in which your application can have explicit control over Pro/ENGINEER. Pro/ENGINEER calls the **user_initialize()** function when applications start, and a synchronous application can take control by initiating all the operations in **user_initialize()** (thus forestalling any user interaction), and then ending the Pro/ENGINEER session after the application is complete.

This synchronous mode technique is important when you want to run Pro/ENGINEER as a batch job. (See the section Using Pro/TOOLKIT to Make a Batch Pro/ENGINEER Session in the 'Fundamentals' chapter.)

Setting up an Asynchronous Pro/TOOLKIT Application

For your asynchronous application to communicate with Pro/ENGINEER, you must set the environment variable PRO_COMM_MSG_EXE to the full path of the executable pro_comm_msg. On UNIX systems, set PRO_COMM_MSG_EXE using the following command:

```
% setenv PRO_COMM_MSG_EXE  
    <Pro/ENGINEER>/<MACHINE>/obj/pro_comm_msg
```

In this command, % signifies the shell prompt and <Pro/ENGINEER> and <MACHINE> have the definitions given in the section Registering a Pro/TOOLKIT Application in the 'Fundamentals' chapter.

On Windows systems, set PRO_COMM_MSG_EXE in the **Environment** section of the **System** window (which can be accessed from the Control Panel).

Depending on how your asynchronous application handles messages from Pro/ENGINEER, your application can be classified as either "simple" or "full." The following sections describe simple and full asynchronous mode.

Note: For asynchronous mode applications, you are not required to supply **user_initialize()** and **user_terminate()**. If you do not, Pro/TOOLKIT will supply a default implementation for these functions. If you do wish to include a custom version of either function, both the functions **user_initialize()** and **user_terminate()** must be supplied in the application. A custom **user_initialize()** will make it possible to access the **user_initialize()** input arguments and to reuse code from a synchronous application without making modifications.

Simple Asynchronous Mode

A simple asynchronous application does not implement a way to handle requests from Pro/ENGINEER. Therefore, Pro/ENGINEER cannot call functions in the Pro/TOOLKIT application. Consequently, Pro/ENGINEER cannot invoke the callback functions that must be supplied when you add, for example, menu buttons or notifications to Pro/ENGINEER.

Despite this limitation, Pro/ENGINEER running with graphics is still an interactive process available to the user.

When you design a Pro/TOOLKIT application to run in simple asynchronous mode, keep the following in mind:

- The Pro/ENGINEER process and the application perform operations concurrently.
- None of the application's functions are invoked by Pro/ENGINEER.
- Simple asynchronous mode supports Pro/TOOLKIT visit functions (ProSolidFeatVisit(), for example), but does not support notification callbacks.

These considerations imply that the Pro/TOOLKIT application does not know the state (the current mode, for example) of the Pro/ENGINEER process at any moment.

Starting and Stopping Pro/ENGINEER

Functions introduced:

- **ProEngineerStart()**
- **ProEngineerConnectionStart()**
- **ProEngineerEnd()**

A simple asynchronous application can spawn and connect to a Pro/ENGINEER process via the function **ProEngineerStart()**. During this startup, Pro/ENGINEER calls **user_initialize()** if it is present in the application. The Pro/ENGINEER process “listens” for requests from the application and acts on the requests at suitable breakpoints—normally between commands.

The function **ProEngineerConnectionStart()** performs the same task as **ProEngineerStart()**, except that **ProEngineerConnectionStart()** outputs a ProProcessHandle which can be used for later connect and disconnect operations. Using this function requires building with a C++ compiler — see the description of **ProEngineerConnect()** for more information.

To connect to an existing Pro/ENGINEER process from an asynchronous application, see the section Connecting to a Pro/ENGINEER Process.

Unlike applications running in synchronous mode, asynchronous applications are not terminated when Pro/ENGINEER terminates. This functionality is useful when the application needs to perform Pro/ENGINEER operations only intermittently, and therefore start and stop Pro/ENGINEER more than once during a session. To end a Pro/ENGINEER process, call the function **ProEngineerEnd()**.

Connecting to a Pro/ENGINEER Process

Functions introduced:

- **ProEngineerConnectIdGet()**
- **ProEngineerConnect()**
- **ProEngineerConnectWS()**
- **ProEngineerDisconnect()**
- **ProEngineerConnectIdExtract()**

The function **ProEngineerConnectIdGet()** returns the asynchronous connection id of the Pro/ENGINEER session that the application is running with. This function can be called from any synchronous (DLL or spawn mode) or asynchronous application. It allows the application to send the connection id to any other asynchronous application that needs to connect to this specific Pro/ENGINEER session.

A simple asynchronous application can also connect to a Pro/ENGINEER process that is already running on that machine. The function **ProEngineerConnect()** performs this function. It allows you to specify the name of the user who owns the Pro/ENGINEER process to which you want to connect, and the name of the machine used for the display. If multiple Pro/ENGINEER sessions meet this specification, **ProEngineerConnect()** can optionally choose one process at random or return an error status.

The function **ProEngineerConnectWS()** is identical to **ProEngineerConnect()**, but has an additional argument to cause Pro/ENGINEER to simultaneously register a given Pro/INTRALINK workspace.

To disconnect from a Pro/ENGINEER process, call **ProEngineerDisconnect()**.

The connection to a Pro/ENGINEER process uses information that is provided by the name service daemon. The name service daemon accepts and supplies information about the processes running on the specified hosts. The application manager, for example, uses name service when it starts up Pro/ENGINEER and other processes. The name service daemon is set up as part of the Pro/ENGINEER installation.

The function **ProEngineerConnectIdExtract()** returns a string connection identifier for the Pro/ENGINEER process. This identifier can be used later to connect to the same process using a call to **ProEngineerConnect()** or **ProEngineerConnectWS()**. Pass the connection id as the first argument to the connection function.

To use the functions in this section, and also the function **ProEngineerConnectionStart()**, you must link your application with the library `pt_asynchronous.a`, which is in the following location:

```
<TK_LOADPOINT>/protoolkit/<MACHINE>/obj/pt_asynchronous.a
```

Because this is a C++ library, you must use a C++ compiler to build your application. For sample makefiles containing C++ settings, see the makefiles under the following directory:

```
<TK_LOADPOINT>/protoolkit/<MACHINE>/obj/
```

Note: You do not have to link with `pt_asynchronous.a` (or use a C++ compiler) if you do not use the functions just described or **ProEngineerConnectionStart()**.

Example 1: Connecting to Pro/ENGINEER

This example and Example 2: Retrieve, Calculate, Delete show the use of a simple asynchronous mode, including connecting to an existing or starting Pro/ENGINEER session, and then using the Pro/ENGINEER session to retrieve a part, calculate its mass, and erase it again.

```
/*=====*\
FUNCTION : main()
PURPOSE  : Connect to, or start, a Pro/ENGINEER session, and use it to
           find the mass of a Pro/E part called prt0001.prt
\*=====*/
main()
{
    ProBoolean connected=PRO_B_FALSE;
    ProBoolean random;
    ProProcessHandle process;
```

```

    double mass;

/*-----*\
    Try to connect to a Pro/ENGINEER session
/*-----*\
    if(ProEngineerConnect(NULL, "bab5","alistair",".",
                          PRO_B_TRUE, 5, &random, &process) !=
PRO_TK_NO_ERROR)
    {
/*-----*\
        That failed, so start a Pro/ENGINEER
/*-----*\
        if(ProEngineerStart("pro2001",".") !=
          PRO_TK_NO_ERROR)
        {
            printf("Could not connect to, or start, Pro/ENGINEER\n");
            return(0);
        }
    }
    else
        connected = PRO_B_TRUE;

    if(connected)
        printf("Connected to an existing Pro/ENGINEER session\n");
    else
        printf("Started a new Pro/ENGINEER session\n");

/*-----*\
    Get the mass properties of a part called prt0001.prt
/*-----*\
    if(UsrMassGet("prt0001", PRO_MDL_PART, &mass))
        printf("Mass of prt0001.prt = %f\n", mass);

/*-----*\
    Disconnect, or end the Pro/E session we started.
/*-----*\
    if(connected)
        ProEngineerDisconnect(&process, 5);
    else
        ProEngineerEnd();

    return(1);
}

```

Example 2: Retrieve, Calculate, Delete

```

#include "ProToolkit.h"
#include "ProCore.h"
#include "ProMdl.h"
#include "ProSolid.h"
#include "ProUtil.h"

```

```

/*=====*\
FUNCTION : UsrMassGet()
PURPOSE  : Output the mass of a specified Pro/ENGINEER solid
\*=====*/
int UsrMassGet(
    ProCharName name,
    ProMdlType type,
    double *mass)
{
    ProName wname;
    ProError status;
    ProMdl mdl;
    ProMassProperty massprops;

/*-----*\
    Check that the model is a part or assembly
\*-----*/
    if(type != PRO_MDL_PART &&
        type != PRO_MDL_ASSEMBLY)
        return(0);

/*-----*\
    Retrieve it
\*-----*/
    ProStringToWstring(wname, name);
    if(ProMdlRetrieve(wname, type, &mdl) != PRO_TK_NO_ERROR)
        return(0);

/*-----*\
    Get the mass properties
\*-----*/
    if(ProSolidMassPropertyGet((ProSolid)mdl, NULL, &massprops) !=
PRO_TK_NO_ERROR)
        return(0);
    *mass = massprops.mass;

/*-----*\
    Erase it from memory
\*-----*/
    ProMdlErase(mdl);

    return(1);
}

```

Status of a Pro/ENGINEER Process

Function introduced:

- **ProEngineerStatusGet()**

It might be useful for your application to know whether a Pro/ENGINEER process is running. The function **ProEngineerStatusGet()** returns this information.

Full Asynchronous Mode

Functions introduced:

- **ProEventProcess()**
- **ProAsynchronousEventLoop()**
- **ProAsynchronousEventLoopInterrupt()**
- **ProTermFuncSet()**
- **ProTerminationAction()**

Full asynchronous mode is identical to simple asynchronous mode except in the way the Pro/TOOLKIT application handles requests from Pro/ENGINEER. In simple asynchronous mode, it is not possible to process such requests. In full asynchronous mode, the application must implement a control loop that “listens” for messages that arrive from Pro/ENGINEER. As a result, Pro/ENGINEER can call functions in the application, including callback functions for menu buttons and notifications.

The control loop of an application running in full asynchronous mode must contain a call to the function **ProEventProcess()**, which takes no arguments. This function responds to Pro/ENGINEER messages in a manner similar to synchronous mode. For example, if the user selects a menu button that is added by your application, **ProEventProcess()** processes the call to your callback function and returns when the call completes. (For more information on callback functions and adding menu buttons, see the ‘User Interface: Menus, Commands, and Popupmenus’ chapter.)

The function **ProAsynchronousEventLoop()** provides an alternative to the development of an event processing loop in a full asynchronous mode application. Call this function to have the application wait in a loop for events to be passed from Pro/ENGINEER. No other processing will take place while the application is waiting. The loop will continue until **ProAsynchronousEventLoopInterrupt()** is called from a Pro/TOOLKIT callback action, or until the application detects that Pro/ENGINEER has terminated.

It is often necessary for your full asynchronous application to be notified of the termination of the Pro/ENGINEER process. In particular, your control loop need not continue to listen for Pro/ENGINEER messages if Pro/ENGINEER is no longer running. The function **ProTermFuncSet()** binds a termination action to be executed when Pro/ENGINEER is terminated. The termination action is a function that you supply and identify in the input of **ProTermFuncSet()** by a function pointer of type *ProTerminationAction*. The input to the termination action is the termination type, which is one of the following:

- PROTERM_EXIT—Normal exit (the user picks Exit from the menu).
- PROTERM_ABNORMAL—Exit with error status.
- PROTERM_SIGNAL—Fatal signal raised.

Your application can interpret the termination type and take appropriate action.

Setting Up a Non-Interactive Session

You can spawn a Pro/ENGINEER session that is both noninteractive and nongraphical. In asynchronous mode, include the following arguments in the call to **ProEngineerStart()**:

- *-g:no_graphics*—Turn off the graphics display.
- *-i:rpc_input*—Cause Pro/ENGINEER to expect input from your asynchronous application only.

Both of these arguments are required, but the order is not important. The syntax of the call for a noninteractive, nongraphical session is as follows:

```
ProEngineerStart ("pro -g:no_graphics -i:rpc_input",
                 <text_dir>);
```

In the syntax, “pro” is the command to start Pro/ENGINEER.

Example 3: Asynchronous Mode Application shows an application that uses Full Asynchronous Mode. It starts a Pro/ENGINEER session and defines a command “Mass” on the part menu which writes a message reporting the mass of the current part. Then it looks for a file called “partlist.txt”. Whenever it finds a file with that name, it reports the mass of all Pro/ENGINEER parts listed in the file, and deletes the file. It also makes regular calls to **ProEventProcess()** so that the Pro/ENGINEER user can select the Mass command at any time and get a prompt response, even while a “partlist.txt” file is being processed.

Note that in this example the function **UsrMassGet()** looks for the required part in memory, and does not erase it if it was found. Doing this minimizes interference with the actions that the Pro/ENGINEER user may be performing in the same session.

Example 3: Asynchronous Mode Application

```
#include "ProToolkit.h"
#include "ProCore.h"
#include "ProMdl.h"
#include "ProMenu.h"
#include "ProSolid.h"
#include "ProUtil.h"

static ProName msgfil;

/*=====*\
FUNCTION : UsrMassGet()
PURPOSE  : Output the mass of a Pro/ENGINEER solid
\*=====*/
int UsrMassGet(
    ProCharName name,
    ProMdlType type,
    double *mass)
{
    ProBoolean already_in_memory = PRO_B_FALSE;
    ProName wname;
    ProError status;
    ProMdl mdl;
    ProMassProperty massprops;

/*-----*\
    Check that the model is a part or assembly
\*-----*/
    if(type != PRO_MDL_PART &&
        type != PRO_MDL_ASSEMBLY)
        return(0);

/*-----*\
    If it is not in memory, retrieve it
```

```

\*-----*/
    ProStringToWstring(wname, name);
    if(ProMdlInit(wname, PRO_MDL_PART, &mdl) == PRO_TK_NO_ERROR)
        already_in_memory = PRO_B_TRUE;
    else
        if(ProMdlRetrieve(wname, type, &mdl) != PRO_TK_NO_ERROR)
            return(0);

/*-----*\
    Get the mass properties
\*-----*/
    if(ProSolidMassPropertyGet((ProSolid)mdl, NULL, &massprops) !=
        PRO_TK_NO_ERROR)
        return(0);

    *mass = massprops.mass;

/*-----*\
    Erase it from memory, if we retrieved it ourselves.
\*-----*/
    if(!already_in_memory)
        ProMdlErase(mdl);

    return(1);
}

static int terminated=0;

/*=====*\
FUNCTION : UsrTermAction()
PURPOSE  : Terminate callback to be called when Pro/ENGINEER exits
\*=====*/
ProError UsrTermAction(
    ProTerminationStatus status)
{
    terminated=1;

    return(PRO_TK_NO_ERROR);
}

/*=====*\
FUNCTION : UsrMass()
PURPOSE  : To implement the Mass command on the part menu - writes a
           message reporting the mass of the current part
\*=====*/
int UsrMass()
{
    ProMdl mdl;
    ProMassProperty massprops;

    ProMdlCurrentGet(&mdl);

```

```

        if(ProSolidMassPropertyGet((ProSolid)mdl, NULL, &massprops) ==
            PRO_TK_NO_ERROR)
            ProMessageDisplay(msgfil,"USER Mass of current solid is %0f",
                &massprops.mass);

        return(1);
    }

/*=====*\
FUNCTION : main()
PURPOSE  : Start a Pro/E session, set up a Mass command on the part
           menu, and also report the masses of part listed
           in a file called "partlist.txt" whenever it is found
\*=====*/
main()
{
    double mass;
    int menu_id;
    FILE *fp;
    char *partname;
    ProCharLine line;

/*-----*\
    Start a Pro/ENGINEER session
\*-----*/
    if(ProEngineerStart("/tools/bin/proshipi02",".") != PRO_TK_NO_ERROR)
    {
        printf("Could not connect to, or start, Pro/ENGINEER\n");
        return(0);
    }

/*-----*\
    Set up the name of the message file
\*-----*/
    ProStringToWstring(msgfil,"umsg.txt");

/*-----*\
    Add the command "Mass" to the part menu
\*-----*/
    ProMenuFileRegister("part","part.mnu",&menu_id);
    ProMenuAuxfileRegister("part","part.aux",&menu_id);

ProMenubuttonActionSet("part","Mass",(ProMenubuttonAction)UsrMass,NULL,0)
;

/*-----*\
    Define the terminate action
\*-----*/
    ProTermFuncSet(UsrTermAction);

```

```

/*-----*\
    While Pro/ENGINEER is still running
/*-----*/
    while(!terminated)
    {
/*-----*\
        Process any events from Pro/ENGINEER (e.g. the user selected
        the Mass command.
/*-----*/
        ProEventProcess();
/*-----*\
        Open the "partlist.txt" file if it exists
/*-----*/
        fp = fopen("partlist.txt","r");
        if(!fp)
            continue;
/*-----*\
        For each part in the file
/*-----*/
        while(fgets(line, PRO_LINE_SIZE, fp))
        {
            partname = strtok(line, " \t\n");
/*-----*\
            Report the mass
/*-----*/
            if(UsrMassGet(partname, PRO_MDL_PART, &mass))
                printf("Mass of part %s = %f\n", partname, mass);
/*-----*\
            Process any waiting events from Pro/ENGINEER
/*-----*/
            ProEventProcess();
        }
/*-----*\
        Close and delete the file
/*-----*/
        fclose(fp);
        remove("partlist.txt");
    }

    return(1);
}

```

14

User Interface: Messages

This chapter describes the functions used to communicate with the user through the text message area, including keyboard entry.

Topic	Page
Writing a Message Using a Popup Dialog	14 - 2
Writing a Message to the Message Window	14 - 4
Message Classification	14 - 7
Writing a Message to an Internal Buffer	14 - 8
Getting Keyboard Input	14 - 9
Using Default Values	14 - 9

Writing a Message Using a Popup Dialog

The function **ProUIMessageDialogDisplay()** displays the UI message dialog. The input arguments to the function are:

- The type of message to be displayed.
- The text to display as the title of the dialog. If you want to support displaying localized text, use the message files and the function **ProMessageToBuffer()** to generate this string. Message files are described later in this chapter.
- The message text to be displayed in the dialog. If you want to support displaying localized text, use the message files and function **ProMessageToBuffer()** to generate this string. Message files are described later in this chapter.
- ProArray of possible button identifiers for the dialog.
- The identifier of the default button for the dialog.

The function outputs the button that the user selected.

Example 1: Displaying the UI Message Dialog

```
#include <ProToolkit.h>
#include <ProMessage.h>
#include <ProUIMessage.h>
#include <ProArray.h>

/*=====*\
FUNCTION: UserDisplayPopupConfirmation
PURPOSE: Display a hardcoded confirmation message and handle user's
choice.
\*=====*/
ProError UserDisplayPopupConfirmation ()
{
    ProUIMessageButton* buttons;
    ProUIMessageButton user_choice;

/*-----*\
    Setup array of choices to display in the popup dialog.
\*-----*/
    ProArrayAlloc (2, sizeof (ProUIMessageButton),
                  1, (ProArray*)&buttons);

    buttons [0] = PRO_UI_MESSAGE_YES;
    buttons [1] = PRO_UI_MESSAGE_NO;

    ProUIMessageDialogDisplay (PROUIMESSAGE_QUESTION,
                              L"Confirmation",
                              L"Do you really want to delete the feature?",
```

```

        buttons,
        PRO_UI_MESSAGE_YES,
        &user_choice);

ProArrayFree ((ProArray*)&buttons);

if (user_choice == PRO_UI_MESSAGE_YES)
{
/*-----*\
Confirmed. Continue with action.
/*-----*/
;
}
else if (user_choice == PRO_UI_MESSAGE_NO)
{
/*-----*\
Denied. Cancel action;
/*-----*/
;
}

return PRO_TK_NO_ERROR;
}

/*=====*\
FUNCTION: UserDisplayPopupTranslatedWarning
PURPOSE: Display a translated warning message in a popup dialog.
/*=====*/
ProError UserDisplayPopupTranslatedWarning ()
{
ProLine message;
ProUIMessageButton* buttons;
ProUIMessageButton user_choice;
/*-----*\
Obtain the message text from the message file into the message variable.
/*-----*/
ProMessageToBuffer (message,
                    L"msg_ugui.txt",
                    "USER Warning: value exceeded specified
                    range of 0 - 100");

ProArrayAlloc (1, sizeof (ProUIMessageButton),
               1, (ProArray*)&buttons);

buttons [0] = PRO_UI_MESSAGE_OK;

/*-----*\
Display the popup dialog.
/*-----*/

```

```

ProUIMessageDialogDisplay (PROUIMESSAGE_WARNING,
                           L"Warning",
                           message,
                           buttons,
                           PRO_UI_MESSAGE_OK,
                           &user_choice);

ProArrayFree ((ProArray*)&buttons);

return PRO_TK_NO_ERROR;
}

/*=====*\
FUNCTION: UserDisplayMessageDialogs
PURPOSE: Display message dialogs
\*=====*/
ProError UserDisplayMessageDialogs()
{
    UserDisplayPopupConfirmation();
    UserDisplayPopupTranslatedWarning();

    return PRO_TK_NO_ERROR;
}

```

Writing a Message to the Message Window

This section describes the following topics:

- Displaying and clearing messages
- The text message file

Functions Introduced:

- **ProMessageDisplay()**
- **ProMessageClear()**
- **ProUIMessageDialogDisplay()**

The function **ProMessageDisplay()** is similar to the C function *printf()*, but with some important differences:

- The first argument is the name (as a wide string) of the message file. The name must include the file extension, but not the path. See the section Text Message File Format and Restrictions.
- The second argument, instead of being a format string, is a keyword used to look up the format string in the message file.

- The subsequent arguments for the values inserted into the format string are pointers, not values. These values can be data inserted into the message or default values for the data to be read from user input. See the section Getting Keyboard Input for more information.
- Although the list of arguments for the values is variable in number, there is a maximum of 9. See Contents of the Message File for more information on using these arguments with a message format.

The function **ProMessageClear()** scrolls the text in the message area up one line. This could be used to indicate that Pro/ENGINEER has received the user's response to a message.

Text Message File Format and Restrictions

The text message file enables you to provide your own translation of the text message, just as the menu files enable you to provide your own translation of the button name and the one-line command help.

Restrictions on the Text Message File

You must observe the following restrictions when you name your message file:

- The name of the file must be 30 characters or less, including the extension.
- The name of the file must contain lowercase characters only.
- The file extension must be three characters.
- The version number must be in the range 1 to 9999.
- All message file names must be unique, and all message key strings must be unique across all applications that run with Pro/ENGINEER. Duplicate message file names or message key strings can cause Pro/ENGINEER to exhibit unexpected behavior. To avoid conflicts with the names of Pro/ENGINEER or Pro/TOOLKIT application message files or message key strings, PTC recommends that you choose a prefix unique to your application, and prepend that prefix to each message file name and each message key string corresponding to that application.

Pro/ENGINEER looks for the message file using the following search path:

- The current Pro/ENGINEER directory

- The directory text under the directory named in the *text_dir* statement in the registry file (*protk.dat*).

Note that message files are loaded into Pro/ENGINEER only once during a session, during the first call to **ProMessageDisplay()**. Consequently, if you make a change to the message file while Pro/ENGINEER is running, you must exit and restart Pro/ENGINEER to have the changes take effect.

Contents of the Message File

The message file consists of groups of four lines—one group for each message you want to write. The four lines are as follows:

1. A string that acts as the keyword to identify the message when you call **ProMessageDisplay()**. This keyword must be unique for all Pro/ENGINEER messages.
2. A string that will be substituted for the first string when you call **ProMessageDisplay()**. This string acts like the format string in a *printf()* statement. By modifying this line in the message file, you can modify the text of the message without modifying your C code.
3. The translation of the message into another language (can be blank).
4. An intentionally blank line reserved for future extensions.

The format string (line 2 in the message file) differs from the format string of a *printf()* in the following respects:

- The conversion specifications (*%d*, *%s*, and so on) must include an argument number corresponding to the position of that argument in the subsequent list (starting at 0). For example, instead of *%d*, *%s*, you must have *%0d*, *%1s*, and so on. If you want to specify a field width, put it in parentheses between the position number and the type specifier; for example, *%0(5.3)f*.
- The separator *|||* between message text and a conversion specification signifies that the conversion specification is for a default value for user input. This default value will appear in the text box created using the keyboard input functions, such as **ProMessageIntegerRead()**. Refer to Using Default Values for more on default values.
- The conversion character *w* is available for wide strings.
- You do not need the character constant (*\n*) at the end of the format. Pro/ENGINEER automatically inserts a new line when necessary.

The following table lists the conversion characters and their corresponding data types.

Conversion Character	Data Type
<i>f</i>	Float (or double)
<i>d</i>	Decimal integer
<i>s</i>	Ordinary string (or type <i>char[]</i>)
<i>w</i>	Wide character strings
<i>e</i>	Exponential
<i>g</i>	Either float or exponential, as appropriate

Ensure that the keyword string is similar to the format string to make your C code easy to interpret. Add a prefix that is unique to your application to the keyword string. The examples in this manual use the unique prefix “USER.”

Message Classification

Messages displayed in Pro/ENGINEER include a symbol which identifies the message type. Each message type is identified by a classification which begins with the characters %C. A message classification requires that the message key (line 1 in the message file) be preceded by the classification code. Note that the message key string used in the code should NOT contain the classification.

Pro/TOOLKIT applications can now display any or all of these message symbols:

- Prompt—the Pro/ENGINEER message displayed is preceded by a green arrow. The user must respond to this message type (to either input information, accept the default value offered, or cancel the application). Without such action, no progress can be made. The response may be either textual or in the form of a selection. The classification code for prompt messages is %CP.
- Info—the Pro/ENGINEER message displayed is preceded by a blue dot. This message type contains information such as user requests or feedback from either Pro/ENGINEER or the Pro/TOOLKIT application. The classification code for prompt messages is %CI.

Note: Do not classify as Info any message which informs users of a problem with an operation or process. These messages should be classified as Warnings.

- **Warning**—the Pro/ENGINEER message displayed is preceded by a triangle containing an exclamation point. Warnings alert the user to situations which may lead to potentially erroneous situations at a later stage, for example, possible process restrictions imposed or a suspected data problem. However, warnings do not prevent or interrupt task completion, nor should they be used to indicate a failed operation. Warnings only caution the user that the operation has been completed, but may not have been performed in a completely desirable way. The classification code for prompt messages is %CW.
- **Error**—the Pro/ENGINEER message is preceded by a broken square. This message type informs the user when a required task was not successfully completed. It may or may not require intervention or correction before work can continue, depending on the application. Whenever possible, provide a path to redress this situation. The classification code for prompt messages is %CE.
- **Critical**—the Pro/ENGINEER message displayed is preceded by a red X. This message type informs the user of extremely serious situations, especially those which could cause the loss of user data. Options for redressing the situation (if available) should be provided with the message. The classification code for prompt messages is %CC.

Writing a Message to an Internal Buffer

Function Introduced:

- **ProMessageToBuffer()**

This function has the same relationship to **ProMessageDisplay()** that the C library function *sprintf()* has to *printf()*: it enables you to write a message to an internal, wide-string buffer instead of to the Pro/ENGINEER message area. The **ProMessageToBuffer()** function performs exactly the same argument substitution and translation as **ProMessageDisplay()**. You provide a wide-string buffer as the first argument, and the subsequent arguments are the same as those for **ProMessageDisplay()**.

Getting Keyboard Input

Functions Introduced:

- **ProMessageIntegerRead()**
- **ProMessageDoubleRead()**
- **ProMessageStringRead()**
- **ProMessagePasswordRead()**

These four functions obtain keyboard input from a text box at the bottom of the Pro/ENGINEER window. The functions check the syntax of the user's entry and indicate when the entry is simply a carriage return. Each of the functions enable you to restrict numeric input to a specified range, or string input to a specified string length. The functions continue to prompt the user until a valid response is entered.

The function **ProMessageStringRead()** supports string lengths up to 259 characters.

User Interface:
Messages

Using Default Values

Prior to Release 20, Pro/TOOLKIT applications implemented default values by checking for a user-entered carriage return. Beginning with Release 20, you can specify default values within the call to **ProMessageDisplay()**, provided that the separator ||| appears in the format string in the message file. (See the section Contents of the Message File for the specific placement of the ||| separator.)

Default values are displayed in the text box as input. Note that this value will not be passed to the Pro/TOOLKIT function if the user hits a carriage return; instead, the function will return PRO_TK_GENERAL_ERROR and the application must interpret that the user intends to use the default.

To specify a constant default value, the format string would appear as follows:

Enter a double: |||3.0

Specifying constant defaults is not recommended as changing the default requires revising the Pro/TOOLKIT application. Specifying defaults that are variables is more flexible.

To specify a default integer that is a variable, for example, the format string in the message file would appear as follows:

```
Enter any integer: |||%0d
```

Example 2: Displaying Messages and Retrieving Keyboard Input

This example shows how to print messages and accept keyboard input with default values. The example also shows how to write a message to an internal, wide-string buffer. The name of the message file is *msg_ugmessage.txt*.

```
#include <ProToolkit.h>
#include <ProMessage.h>
#include <PTApplsUnicodeUtils.h>
#include <TestError.h>
#include <ProUtil.h>

#define MAX_IN_LEN PRO_NAME_SIZE

/*=====*\
FUNCTION: UserMessageDemo()
PURPOSE:  Display messages and read user input.
\*=====*/
ProError UserMessageDemo()
{
    int          i1, i2;
    const int     default_int = 0;
    double        d;
    int           irange[2] = {0, 10};
    ProName       wstr, default_wstr;
    ProCharName   str;
    ProLine       msg_line;
    ProCharLine   line;
    ProError       err;
    ProFileName   msgfil;
    FILE          *msg_test;

/*-----*\
    Set up the name of the message file.
\*-----*/
    ProStringToWstring (msgfil, "msg_ugmessage.txt");
/*-----*\
    Read an integer without a default value.  Message will display as a
    prompt.
\*-----*/
    err = ProMessageDisplay (msgfil, "USER Enter any integer: ");
    err = ProMessageIntegerRead (NULL, &i1);
    if (err != PRO_TK_NO_ERROR)
        return (err);
/*-----*\
    Read an integer with a restricted range and a default value.  Message
```

```

will display as a prompt.
/*-----*/
err = ProMessageDisplay (msgfil,
    "USER Enter any integer between %0d and %1d: |||%2d", &irange[0],
    &irange[1], &default_int);
err = ProMessageIntegerRead (irange, &i2);
if (err != PRO_TK_NO_ERROR && err != PRO_TK_GENERAL_ERROR)
    return (err);
/*-----*/
If user entered the default value - warn the user.  Message will
display as a warning.
/*-----*/
if (err == PRO_TK_GENERAL_ERROR)
{
    i2 = default_int; /* Using the default */
    err = ProMessageDisplay (msgfil, "USER Warning: using default value",
        &default_int);
}
/*-----*/
Read a double without a default value.  Message will display as a
prompt.
/*-----*/
err = ProMessageDisplay (msgfil, "USER Enter any double: ");
err = ProMessageDoubleRead (NULL, &d);
if (err != PRO_TK_NO_ERROR)
    return (err);
/*-----*/
Read a string with a default value.  Message will display as a
prompt.
/*-----*/
ProStringToWstring (default_wstr, "default string");
err = ProMessageDisplay (msgfil, "USER Enter any string: |||%0w",
    default_wstr);
err = ProMessageStringRead (MAX_IN_LEN, wstr);
if (err != PRO_TK_NO_ERROR && err != PRO_TK_GENERAL_ERROR)
    return (err);
if (err == PRO_TK_GENERAL_ERROR)
    ProUtilWstrcpy (wstr, default_wstr); /* Using the default */
/*-----*/
Write a message that states the values entered.  Message will display
as info.
/*-----*/
err = ProMessageDisplay (msgfil, "USER Values entered were %0d, %1d,
    %2(5.3)f, %3w", &i1, &i2, &d, wstr);
/*-----*/
Write the values to a file.
/*-----*/
msg_test = PTApplsUnicodeFopen ("msg_test.dat", "w");
if (msg_test != NULL)
{
    err = ProMessageToBuffer (msg_line, msgfil, "USER Values entered",

```

```

        wstr, &d, &i2, &i1);
    ProWstringToString (line, msg_line);
    ProTKFprintf (msg_test, "ProMessageToBuffer output |%s|\n", line);
    fclose (msg_test);
}
return (PRO_TK_NO_ERROR);
}

#undef MAX_IN_LEN

Message file msg_ugmessage.txt
    %CPUSER Enter any integer:
    Enter any integer:
    #
    #
    %CPUSER Enter any integer between %0d and %1d: |||%2d
    Enter any integer between %0d and %1d: |||%2d
    #
    #
    %CPUSER Enter any double:
    Enter any double:
    #
    #
    %CPUSER Enter any string: |||%0w
    Enter any string: |||%0w
    #
    #
    %CIUSER Values entered were %0d, %1d, %2(5.3)f, %3w
    Values entered were %0d, %1d, %2(5.3)f, %3w
    #
    #
    %CIUSER Values entered
    Values entered were %3d, %2d, %1(5.3)f, "%0w"
    #
    #
    %CWUSER Warning: using default value
    Warning: using default value %0d
    #
    #

```


15

User Interface: Menus, Commands, and Popupmenus

This chapter describes all the functions provided by Pro/TOOLKIT to create and manipulate menus and menu buttons.

Topic	Page
Introduction	15 - 2
Menu Bar Buttons and Menus	15 - 2
Designating Commands	15 - 20
Popup Menus	15 - 31
Mode-Specific Buttons and Menus	15 - 41
Customizing the Pro/ENGINEER Navigation Area	15 - 60
Entering Pro/ENGINEER Commands	15 - 69

Introduction

Using Pro/TOOLKIT, you can modify and supplement the Pro/ENGINEER menu structure. This functionality incorporates the Pro/ENGINEER ability to provide language translation for menu buttons and help strings.

When you are designing your Pro/TOOLKIT application, you should carefully consider the context of the buttons and menus that you add to the Pro/ENGINEER User Interface (UI). Buttons specific to a particular mode (such as PART) should be located on the menu related to that mode. Buttons that initiate some action on a part, for example, should be located on the PART menu. For another example, user programs that use the 3-D model and have their own interface are best located on the APPLICATIONS menu

There are fundamental differences in the files and functions used to manipulate menu bar and mode-specific menus. For this reason, this manual describes these subjects in separate sections.

Menu Bar Buttons and Menus

The menu bar of the Pro/ENGINEER interface contains menus composed of both buttons and submenus. Using Pro/TOOLKIT, you can create similar menu structures in the Pro/ENGINEER menu bar. These are the menu bar object definitions

- Menu bar—The top level horizontal bar in the Pro/ENGINEER UI, containing the main menus like **File**, **Edit**, and **Applications**.
- Menu bar menu—A menu, such as the **File** menu, or a sub-menu, such as the **Export** menu under the **File** menu.
- Push button—A named item in a menu bar menu that is used to launch a set of instructions. An example is the **Exit** button in the **File** menu.
- Check button—An item in a menu bar menu that may be toggled on and off. An example is the **Model Tree** toggle in the **View** menu.
- Radio group—An item in a menu bar menu that may be set to one and only one of any number of options. An example is the group of windows listed in the bottom of the **Window** menu which allow you to switch between different windows.

- **Command**—A procedure in Pro/ENGINEER that may be activated from a menu bar object or a toolbar icon.

Note: Pro/TOOLKIT does not allow adding items to the toolbar in Release 2000i².

- **Command ID**—An opaque pointer to a command, used as an input to other Pro/TOOLKIT functions.
- **Action command**—A command which executes a set of instructions. Launched by push buttons.
- **Option command**—A command which executes a set of instructions based on the state of a UI component. Launched by check buttons and radio groups.

The Pro/TOOLKIT menu bar functions enable you to modify existing Pro/ENGINEER menu bar menus and to create new menu bar menus. These operations are described in the following topics:

- Adding a PushButton to a Menu Bar Menu
- Adding a Check Button to a Menu Bar Menu
- Adding a RadioButton Group to a Menu Bar Menu
- Adding a Menu to a Menu Bar Menu
- Adding a Menu to the Menu Bar
- Manipulating Existing Commands

Note:

- PTC cannot guarantee that the organization of the Pro/ENGINEER interface will not change in future releases. Therefore, you should avoid any dependence on the presence of certain Pro/ENGINEER menus. For example, if you add a button to the menu bar menu **Tools** and PTC later removes **Tools**, you will need to rewrite at least part of your Pro/TOOLKIT application code. To avoid such dependence, PTC recommends that you supplement the Pro/ENGINEER menu bar beginning at the highest level. That is, add a menu to the menu bar and add new buttons and menus to this new menu bar menu.
- From Pro/ENGINEER Wildfire 5.0 onward, the user interface in the Detailed Drawing module has been restructured. Therefore, if existing Pro/TOOLKIT applications add buttons to menus that no longer exist, a new menu Pro/TOOLKIT will be added to the Pro/ENGINEER menu bar as the right most menu by default. This new menu will contain the menu buttons added by the Pro/TOOLKIT application.

Using the Trail File to Determine UI Names

Several functions dealing with menu bar UI components require the input of strings that Pro/ENGINEER uses to identify commands and menu buttons.

Note: From Pro/ENGINEER Wildfire 4.0 onward, you must set the configuration option `cmdmgr_trail_output` to `no` in order to display component names in the trail file.

To find the name of an action command (not a menu button), click the corresponding icon on the toolbar (not the button in the menu bar) and then check the last entry in the trail file. For example, for the save icon, the trail file will have the corresponding entry:

```
~ Activate 'main_dlg_cur' 'ProCmdModelSave.file'
```

The action name for the save button is `ProCmdModelSave`. This string can be used as input to **`ProCmdCmdIdFind()`** to get the command ID.

A way to determine a command ID string for an option without an icon would be to search through the resource files located under the `<Pro/ENGINEER Loadpoint>/text/resources`. If you search for the menu button name, the line will contain the corresponding action command for the button.

In order to find the name of a menu button (used to determine the placement of new menu buttons), click on the menu bar menu button (not the icon) and then check the trail file. For example, for the Auxiliary Applications button, the trail file will have the following entry:

```
~ Select 'main_dlg_cur' 'MenuBar1' \  
1 'Utilities'  
~ Close 'main_dlg_cur' 'MenuBar1'  
~ Activate 'main_dlg_cur' 'Utilities.psh_util_aux'
```

The menu name selected is `Utilities`, and the menu button name is `Utilities.psh_util_aux`.

Adding a PushButton to a Menu Bar Menu

To add a button to the menu bar, your Pro/TOOLKIT application must do the following:

1. Define the action command to be initiated by the button. The action is defined in a function known as the “callback function.”
2. Add the button to the menu bar. This operation binds the added action to the button.

These procedures are described in the sections that follow.

Adding an Action to Pro/ENGINEER

Function Introduced:

- **ProCmdActionAdd()**

The function **ProCmdActionAdd()** adds an action to Pro/ENGINEER. The syntax of this function is as follows:

```
ProError ProCmdActionAdd (
    char                *action_name,
    uiCmdCmdActFn       action_cb,
    uiCmdPriority        priority,
    uiCmdAccessFn       access_func,
    ProBoolean          allow_in_non_active_window,
    ProBoolean          allow_in_accessory_window,
    uiCmdCmdId          *action_id );
```

This function takes the following arguments:

- *action_name*—The name of the command as it will be used in Pro/ENGINEER. This name must be unique, and it must occur only once in your applications or in Pro/ENGINEER. To prevent conflicts, PTC recommends prepending or appending a unique identifier to your command names, similar to `ptc_openfile` or `openfile_ptc`.
- *action_cb*—The action function (callback function) that will be called when the command is activated by pressing the button, cast to a *uiCmdCmdActFn*:

```
typedef int (*uiCmdCmdActFn) (
    uiCmdCmdId  command,
    uiCmdValue  *p_value,
    void        *p_push_command_data
);
```

- *command*—Identifier of the action or option.

- *p_value*—For options passed to ValueGet functions. Ignored for actions.
- *p_push_command_data*—Not implemented in this release.
- *priority*—The command priority. The priority of the action refers to the level of precedence the added action takes over other Pro/ENGINEER actions.

The available action priorities are defined in the enumerated type *uiCmdPriority*. The possible values are as follows:

```
typedef int uiCmdPriority;
#define uiCmdPrioDefault      ((uiCmdPriority) 0)
#define uiProeImmediate      ((uiCmdPriority) 2)
#define uiProeAsynch         ((uiCmdPriority) 3)
#define uiProe2ndImmediate   ((uiCmdPriority) 5)
#define uiProe3rdImmediate   ((uiCmdPriority) 6)
#define uiCmdNoPriority       ((uiCmdPriority) 999)
```

The following table describes the enumerated values in detail.

Value	Description
uiCmdPrioDefault	Normal priority actions Normal priority actions dismiss all other actions except asynchronous actions. Note that buttons of this priority can lead to the dismissal of mode-specific menus such as Part or Assembly. Dismissing these menus can result in unexpected behavior from functions that depend on the mode and the context of the Pro/ENGINEER session. One example of a function which can exhibit unintended behavior is ProSelect() when selecting objects from an active simplified representation. Menu buttons should have lesser priority if they depend on the context of the Pro/ENGINEER session.
uiProeImmediate, uiProe2ndImmediate, and uiProe3rdImmediate	Levels of immediate priority. Actions of each level of priority dismiss actions with lower level priorities.
uiProeAsynch	Asynchronous priority. Actions with asynchronous priority are independent of all other actions.

- *access_func*—The access function (callback function) that determines if the menu button should be available, unavailable, or hidden. Action accessibility refers to whether an added menu button or menu is available for user selection. This function is called each time the button parent menu is displayed. The accessibility is evaluated based on the conditions pertaining at the time the button is pressed. The access function must be cast to a *uiCmdAccessFn*:

```
typedef uiCmdAccessState (*uiCmdAccessFn)
(uiCmdAccessMode access_mode);
```

The potential return values are listed in the enumerated type *uiCmdAccessState*:

ACCESS_REMOVE—The button is not visible, and the containing menus might also be removed from the menu, if all of the menu buttons in the containing menu possess an access function returning **ACCESS_REMOVE**.

ACCESS_INVISIBLE—The button is not visible.

ACCESS_UNAVAILABLE—The button is visible, but gray and cannot be selected.

ACCESS_DISALLOW—The button shows as available, but the command will not be executed when it is chosen.

ACCESS_AVAILABLE—The button is not gray and can be selected by the user.

- *allow_in_non_active_window*—A ProBoolean determining whether or not to show this command in any non active window. A non-active window is a window that exists and contains a model, but that is not the active window in the Pro/ENGINEER session. A window becomes active when the user chooses **Window>Activate** or by opening a model in a new window.
- *allow_in_accessory_window*—A ProBoolean determining whether or not to show this command in an accessory window. An accessory window is smaller than a main Pro/ENGINEER window. Usually, this window has no toolbar and allows only the **File>Exit** command from the menu bar.

- *action_id*—The function will return a *uiCmdCmdId*, the command identifier. This identifier is required when associating the added action to a push button with function **ProMenubarmenuPushbuttonAdd()**.

Notes:

- The function **ProCmdActionAdd()** is executed only once per Pro/ENGINEER session for each action. Subsequent calls to this function for a previously loaded action are ignored (therefore you cannot redefine an action within a Pro/ENGINEER session).
- Menu bar buttons are not intended to be mode-specific. For this reason, it is not possible to pass data to the callback function.

Adding a Menu Button

Function Introduced:

- **ProMenubarmenuPushbuttonAdd()**

Adding a button to the Pro/ENGINEER menu bar requires that the necessary information is specified in a text message file and in the input arguments to **ProMenubarmenuPushbuttonAdd()**.

The syntax of the function is as follows:

```
ProError ProMenubarmenuPushbuttonAdd (
    ProMenuItemName    parent_menu,
    ProMenuItemName    push_button_name,
    ProMenuItemLabel    push_button_label,
    ProMenuLineHelp    one_line_help,
    ProMenuItemName    neighbor,
    ProBoolean          add_after_neighbor,
    uiCmdCmdId          action_id,
    ProFileName         filename
);
```

The arguments are as follows:

- *parent_menu*—The name of the parent menu under which the new push button is to appear. For information on finding the name of the menu, see Using the Trail File to Determine UI Names.

- *push_button_name*—The name (in character string format) of the added button. For more information on valid characters that you can use to specify the name, refer to the section Naming Convention for UI Components. Button names are important when specifying the neighbors of items added to a menu.

Note: The pushbutton name will be the name placed in the trail file when the user selects your menu button.

- *push_button_label*—The label for the new push button. The label is a keyword that is used to look up the text in the message file. It identifies the text seen when the button is displayed.
- *one_line_help*—The one-line Help for the button. The label is a keyword that is used to look up the text in the message file. It identifies the help line seen when the mouse moves over the button.

The appearance and the one-line Help of the added menu bar button are both specified in a text message file. This message file follows the format of other Pro/ENGINEER text message files in that it contains groups of four lines of information. (See the section Text Message File Format and Restrictions for more information.) The message file must contain one four-line group for the button itself (to set the text that appears in the menu) and one four-line group for the one-line Help text that appears to the user. (You specify the name of the text message file in the *filename* argument.)

- *neighbor*—The name of the neighboring menu button. For information on finding the name of the neighboring button, see Using the Trail File to Determine UI Names.

Note: Set this value to NULL to add the menu button at the beginning or end of a menu (based on the *add_after_neighbor* parameter).

- *add_after_neighbor*—Set this to PRO_B_TRUE if the button is to appear below the neighboring button. Otherwise, set this to PRO_B_FALSE. If the neighbor argument is NULL, PRO_B_TRUE means put the button at the end of the menu, and PRO_B_FALSE places the button at the beginning.
- *filename*—The name of the text message file.

Note: This file must be in the directory <text_path>/text or <text_path>/text/<language>. It *cannot* be in the directory <text_path>/text/menus or <text_path>/text/<language>/menus.

In the text message file, the first line of each four-line group is a keyword by which Pro/ENGINEER identifies the group. The second line is the English text that will appear on the menu button or one-line Help. The third line, which can be blank, is the translation of the text into another language. The fourth line should be left blank in the current release.

- *action_id*—To bind the added menu bar button to the added action, you must specify this identifier (which is output by a previous call to **ProCmdActionAdd()**).

Notes:

- If you are adding the first item to a new menu created using **ProMenubarMenuAdd()**, set the *neighbor* argument to NULL.

If there are already items in the menu:

- To add an item to the top of the menu, set the *neighbor* argument to NULL and set *add_after_neighbor* to PRO_B_FALSE.
- To add an item to the bottom of the menu, set the *neighbor* argument to NULL and set *add_after_neighbor* to PRO_B_TRUE.
- The function **ProMenubarmenuPushbuttonAdd()** is executed only once per Pro/ENGINEER session for each added push button. Subsequent calls to this function for a previously loaded button are ignored (therefore, it is not possible to redefine a push button within a Pro/ENGINEER session).
- The keywords and text for added menu bar buttons can reside in the same message files as the keywords and text used in the rest of your application. That is, the message file used by **ProMenubarmenuPushbuttonAdd()** does not need to be separate from the message file used by the function **ProMessageDisplay()**.

Adding a Check Button to a Menu Bar Menu

To add a check button to the menu bar, your Pro/TOOLKIT application must do the following:

1. Define the option command to be initiated by the button. The definition of this command includes the definition of three callback functions.
2. Add the check button to the menu bar. This operation binds the added action to the button.

These procedures are described in the sections that follow.

Adding an Option Command to Pro/ENGINEER—Check Button

Functions Introduced:

- **ProCmdOptionAdd()**
- **ProMenubarmenuChkbuttonValueGet()**
- **ProMenubarmenuChkbuttonValueSet()**

The function **ProCmdOptionAdd()** adds a command to Pro/ENGINEER.

The syntax of this function is as follows:

```
ProError ProCmdOptionAdd (
    char                *option_name,
    uiCmdCmdActFn       option_cb,
    ProBoolean          boolean_operation,
    uiCmdCmdValFn       set_value_cb,
    uiCmdAccessFn       access_func,
    ProBoolean          allow_in_non_active_window,
    ProBoolean          allow_in_accessory_window,
    uiCmdCmdId          *option_id );
```

This function requires the following arguments:

- *option_name*—The name of the option command. This must be unique, in the same way as action command.
- *option_cb*—The action command to be executed when the check button is toggled, cast to a *uiCmdCmdActFn*. This function should include a call to **ProMenubarmenuChkbuttonValueGet()**, to determine the value of the check button.
- *boolean_operation*—Specifies whether or not the option has two values. Set this to **PRO_B_TRUE** for a check button.
- *set_value_cb*—The callback function that sets the value of the check button, cast to a *uiCmdCmdValFn*:

```
typedef int  (*uiCmdCmdValFn) (
    uiCmdCmdId  command,
    uiCmdValue  *p_value
);
```

This function should include a call to **ProMenubarmenuChkbuttonValueSet()** to set the value of the check button when the UI is displayed or refreshed.

- *access_func*—The callback function that determines if the command is accessible.
- *allow_in_non_active_window*—A ProBoolean determining whether or not to show this command in any non-active window. A non-active window is a window that exists and contains a model, but that is not the active window in the Pro/ENGINEER session. A window becomes active when the user chooses **Window>Activate** or by opening a model in a new window.
- *allow_in_accessory_window*—A ProBoolean determining whether or not to show this command in an accessory window. An accessory window is smaller than a main Pro/ENGINEER window. Usually, this window has no toolbar and allows only the **File>Exit** command from the menu bar.

The functions **ProMenubarmenuChkbuttonValueGet()** and **ProMenubarmenuChkbuttonValueSet()** provide access to the value of the check button. These functions require the option command value (provided by the callback functions as input), and the value is expressed as a ProBoolean.

Adding a Check Button

Function Introduced:

- **ProMenubarmenuChkbuttonAdd()**

Adding a check button to the Pro/ENGINEER menu bar requires you to specify the necessary information in a text message file and in the input arguments to **ProMenubarmenuChkbuttonAdd()**.

The syntax of this function is as follows:

```
ProError ProMenubarmenuChkbuttonAdd (
    ProMenuItemName      parent_menu,
    ProMenuItemName      check_button_name,
    ProMenuItemLabel      check_button_label,
    ProMenuItemLineHelp   one_line_help,
    ProMenuItemName      neighbor,
    ProBoolean            add_after_neighbor,
    uiCmdCmdId            option_id,
    ProFileName           filename );
```

The arguments are identical to the function **ProMenubarmenuPushbuttonAdd()**.

Notes:

- If you are adding the first item to a new menu created using **ProMenubarMenuAdd()**, set the *neighbor* argument to NULL.

If there are already items in the menu:

- To add an item to the top of the menu, set the *neighbor* argument to NULL and set *add_after_neighbor* to PRO_B_FALSE.
- To add an item to the bottom of the menu, set the *neighbor* argument to NULL and set *add_after_neighbor* to PRO_B_TRUE.

Adding a RadioButton Group to a Menu Bar Menu

To add a radio button group to the menu bar, your Pro/TOOLKIT application must:

1. Define the option command to be initiated by the group of buttons. The definition of this command includes the definition of three callback functions.
2. Add the radio button group to the menu bar. This operation binds the added action to the button.

These procedures are described in the sections that follow.

Adding an Option Command to Pro/ENGINEER—Radio Group

Functions Introduced:

- **ProCmdOptionAdd()**
- **ProMenubarMenuRadiogrpValueGet()**
- **ProMenubarMenuRadiogrpValueSet()**

The function **ProCmdOptionAdd()** is used to create the option command corresponding to the button group.

The arguments should be similar to the usage for creating the option command for a check button, with the following exceptions:

- *output_callback_function*—Must include a call to **ProMenubarMenuRadiogrpValueGet()** to determine the selected value in the radio group.
- *boolean_operations* —Must be PRO_B_FALSE for radio groups.

- *set_value_cb*—Must include a call to **ProMenubarMenuRadiogrpValueSet()** to set the value of the group upon redisplay of the radio group UI.

The functions **ProMenubarMenuRadiogrpValueGet()** and **ProMenubarMenuRadiogrpValueSet()** provide access to getting or setting the selected item in the group. They require the option command value (provided by the callback functions) as an input. The selected value is returned as a ProMenuItemName string.

Adding a Radio Button Group

Function Introduced:

- **ProMenubarmenuRadiogrpAdd()**

The function **ProMenubarmenuRadiogrpAdd()** adds a radio button group to a menu. Its syntax is:

```
ProError ProMenubarmenuRadiogrpAdd
(ProMenuItemName    parent_menu,
 ProMenuItemName    radio_group_name,
 int                number_radio_group_items,
 ProMenuItemName    *radio_group_items,
 ProMenuItemLabel   *radio_group_labels,
 ProMenuItemLineHelp *one_line_helps,
 ProMenuItemName    neighbor,
 ProBoolean         add_after_neighbor,
 uiCmdCmdId         option_id,
 ProFileName        filename );
```

The arguments to this function are:

- *parent_menu*—The name of the menu on which to place the group.
- *radio_group_name*—A unique name for the radio button group. For more information on valid characters that you can use to specify the name, refer to the section Naming Convention for UI Components.
- *number_radio_group_items*—An integer number of selections for the group. The user will only be able to select one of these options.
- *radio_group_items*—An array of radio-group item names (will be returned when the item is selected).

- *radio_group_labels*—An array of labels for the radio buttons. These labels are keywords that are used to look up the text in the message file. The labels identify the text seen when the button is displayed.
- *one_line_helps*—An array of one-line Help labels. These labels are keywords that are used to look up the text in the message file. The labels identify the help line seen when the mouse moves over the button.
- *neighbor*—The neighboring item.
- *add_after_neighbor*—PRO_B_TRUE if add the group after the neighbor, PRO_B_FALSE if add before the neighbor.
- *option_id*—The command option ID.
- *filename*—The message file name. All of the labels and one-line Help labels must be present in the message file.

Notes:

- If you are adding the first item to a new menu created using **ProMenubarMenuAdd()**, set the *neighbor* argument to NULL.

If there are already items in the menu:

- To add an item to the top of the menu, set the *neighbor* argument to NULL and set *add_after_neighbor* to PRO_B_FALSE.
- To add an item to the bottom of the menu, set the *neighbor* argument to NULL and set *add_after_neighbor* to PRO_B_TRUE.

Adding a Menu to a Menu Bar Menu

Function Introduced:

- **ProMenubarmenuMenuAdd()**

To add a sub-menu to a Pro/ENGINEER menu bar menu, call the function **ProMenubarmenuMenuAdd()**. This function is similar to **ProMenubarmenuPushbuttonAdd()** as both functions require the following input arguments:

- Parent menu
- Placement of added menu item
- Message file that contains the text of the menu item
- Keyword used to find the text in the message file

The function **ProMenubarmenuMenuAdd()**, however, does not require arguments that specify access or priority. You build the buttons of your new menu in the same way as you add buttons to existing Pro/ENGINEER menu bar menus: add actions using the function **ProCmdActionAdd()** and add buttons using the function **ProMenubarmenuPushbuttonAdd()**.

Notes:

- If you are adding the first item to a new menu created using **ProMenubarMenuAdd()**, set the *neighbor* argument to NULL.

If there are already items in the menu:

- To add an item to the top of the menu, set the *neighbor* argument to NULL and set *add_after_neighbor* to PRO_B_FALSE.
- To add an item to the bottom of the menu, set the *neighbor* argument to NULL and set *add_after_neighbor* to PRO_B_TRUE.

Adding a Menu to the Menu Bar

Function Introduced:

- **ProMenubarMenuAdd()**

To add a menu to the menu bar, call the function **ProMenubarMenuAdd()**, which is similar to **ProMenubarmenuMenuAdd()**. Both functions require the following input arguments:

- The neighbor of the added menu
- A Boolean flag that specifies the placement of the new menu relative to the neighbor
- The message file
- Keywords for the text of the menu

Notes:

- If you are adding the first menu to a menu bar, set the *neighbor* argument to NULL.

If there are already menus in the menu bar:

- To add a menu to the farthest left of the menu bar, set the *neighbor* argument to NULL and set *add_after_neighbor* to PRO_B_FALSE.

- To add a menu to the farthest right of the menu bar, set the neighbor argument to NULL and set add_after_neighbor to PRO_B_TRUE.

Example 1: Adding to the Menu Bar

The following example code shows how to add menus and buttons to the Pro/ENGINEER menu bar. For simplicity, all the added buttons perform the same action—they display a message for the user. The text message file for the example contains the following lines:

```
[Start of file on next line]
USER %0s
%0s
#
#
USER -UserMenu
-UserMenu
#
#
USER -MainBtn1
-MainBtn1
#
#
USER New Button help.
New Button help.
#
#
USER -Sub1
-Sub1
#
#
USER -Sub1Btn1
-Sub1Btn1
#
#
USER -Sub1Btn2
-Sub1Btn2
#
#
[End of file on previous line]
```

```
/*=====*\
FUNCTION: TestAccessDefault()
PURPOSE: Define the accessibility of menu buttons.
\*=====*/
static uiCmdAccessState TestAccessDefault (uiCmdAccessMode access_mode)
{
    return (ACCESS_AVAILABLE);
}
```

```

/*=====*\
FUNCTION: MiscAction()
PURPOSE:  Generic action function
\*=====*/
int MiscAction()
{
    ProMessageDisplay (UserMsg, "USER %0s", "Action function called.");
    return (0);
}
/*=====*\
FUNCTION: user_initialize()
PURPOSE:  Pro/TOOLKIT standard initialize
\*=====*/
int user_initialize (...)
{
    /*-----*\
    Message file.
    \*-----*/
    ProStringToWstring (UserMsg, "msg_ugfund.txt");
    ProMessageDisplay (UserMsg, "USER %0s",
        "Demo of ProMenuBar.h functions.");
    /*-----*\
    Add a new menu to the menu bar (to the right of Utilities).
    \*-----*/
    status = ProMenubarMenuAdd ("UserMenu", "USER -UserMenu",
        "Utilities", PRO_B_TRUE, UserMsg);
    /*-----*\
    Add to the new menu.
    \*-----*/
    status = ProCmdActionAdd ("UserDispMsg", (uiCmdCmdActFn)MiscAction,
        uiCmdPrioDefault, TestAccessDefault, PRO_B_TRUE, PRO_B_TRUE,
        &cmd_id);

    status = ProMenubarmenuPushbuttonAdd ("UserMenu", "MainBtn1",
        "USER -MainBtn1", "USER New Button help.", NULL, PRO_B_TRUE,
        cmd_id, UserMsg);

    status = ProMenubarmenuMenuAdd ("UserMenu", "Sub1", "USER -Sub1",
        "MainBtn1", PRO_B_TRUE, UserMsg);
    /*-----*\
    Fill in the buttons of Sub1.
    \*-----*/
    status = ProMenubarmenuPushbuttonAdd ("Sub1", "Sub1Btn1",
        "USER -Sub1Btn1", "USER New Button help.", NULL, PRO_B_TRUE,
        cmd_id, UserMsg);
    status = ProMenubarmenuPushbuttonAdd ("Sub1", "Sub1Btn2",
        "USER -Sub1Btn2", "USER New Button help.", NULL, PRO_B_TRUE,
        cmd_id, UserMsg);

    return (0);
}

```

Manipulating Existing Commands

Functions Introduced:

- **ProCmdCmdIdFind()**
- **ProCmdAccessFuncAdd()**
- **ProCmdAccessFuncRemove()**
- **ProCmdBracketFuncAdd()**

The function **ProCmdCmdIdFind()** allows you to find the command ID for an existing command so you can add an access function or bracket function to the command. You must know the name of the command in order to find its ID. See section Using the Trail File to Determine UI Names to determine UI names in order to determine the name of the command.

The functions **ProCmdAccessFuncAdd()** and **ProCmdAccessFuncRemove()** allow you to impose an access function on a particular command. (See function ProCmdActionAdd() for a description of access functions.) The Add function provides an *access_id*. This ID must be saved for later use when you deactivate the access function.

The function **ProCmdBracketFuncAdd()** allows the creation of a function that will be called immediately before and after execution of a given command. this function would be used to add company logic to the start or end (or both) of an existing Pro/ENGINEER command. It could also be used to cancel an upcoming command. This function is declared as:

```
ProError ProCmdBracketFuncAdd (
    uiCmdCmdId      cmd_id,
    uiCmdCmdBktFn   bracket_func,
    char            *bracket_func_name,
    void            **pp_bracket_data );
```

The function takes the following arguments:

- *cmd_id*—The command identifier.
- *bracket_func*—The callback function to be called before and after the command, cast to a *uiCmdCmdBktFn*:

```
typedef int      (*uiCmdCmdBktFn)
( uiCmdCmdId    command,
  uiCmdValue    *p_new_value,
  int           entering_command,
  void          **pp_bracket_data );
```

The entering command argument will be 1 before execution and 0 after. If the operation is before the upcoming command execution, and you want to cancel the upcoming command execution, return 0. Otherwise, return non-zero.

- *bracket_func_name*—The name of the bracket function.
- *pp_bracket_data*—A void** containing data to be passed to the bracket function.

Designating Commands

Using Pro/TOOLKIT you can designate Pro/ENGINEER commands. These commands can later be called in a Pro/ENGINEER session.

In Pro/TOOLKIT you can set an button to refer to a command and subsequently drag this button on to the Pro/ENGINEER toolbar. When the button is clicked, the command is executed.

To add a command to the toolbar, your Pro/TOOLKIT application must do the following:

1. Define or add the command to be initiated on clicking the icon.
2. Optionally designate an icon button to be used with the command defined by you.
3. Designate the command (icon) to appear in the **Screen Customization** dialog of Pro/ENGINEER.
4. Save the configuration in Pro/ENGINEER so that changes to the toolbar appear when a new session of Pro/ENGINEER is started.

Adding the Command

Functions Introduced:

- **ProCmdActionAdd()**
- **ProCmdOptionAdd()**

The functions **ProCmdActionAdd()** and **ProCmdOptionAdd()** allow you to define or register a Pro/ENGINEER command. See the section Adding an Action to Pro/ENGINEER for more information on the function **ProCmdActionAdd()** and the section Adding an Option Command to Pro/ENGINEER—Check Button for more information on the function **ProCmdOptionAdd()**.

Designating the Icon

Function Introduced:

- **ProCmdIconSet()**

The function **ProCmdIconSet()** allows you to designate an icon to be used with the command you created. The function adds the icon to the Pro/ENGINEER command. The function takes the command identifier as one of the inputs and the name of the icon file, including the extension as the other input. A valid format for the icon file is a standard .GIF, .JPG, or .PNG. The Pro/ENGINEER toolbar button is replaced with the icon image.

Toolbar commands that do not have an icon assigned to them display the button label.

You may also use this function to assign a small icon to a menubar menu button. The icon appears to the left of the button label.

Before using the function **ProCmdIconSet()**, you must place the command in a menu using the function **ProMenubarmenuPushbuttonAdd()**.

Designating the Command

Function Introduced:

- **ProCmdDesignate()**

This function allows you designate the command to be available in the **Screen Customization** dialog of Pro/ENGINEER.

After a Pro/TOOLKIT application has used the function **ProCmdDesignate()** on a command, the user can drag the toolbar button associated with this command onto the Pro/ENGINEER toolbar in an interactive session of Pro/ENGINEER.

If this function is not called, the toolbar button will not be visible in the **Screen Customization** dialog of Pro/ENGINEER.

Note: Before using the function **ProCmdDesignate()**, you must place the command in a menu using the function **ProMenubarmenuPushbuttonAdd()**.

The syntax of the function **ProCmdDesignate()** is:

```
ProError ProCmdDesignate ( uiCmdCmdId cmd_id,
                           ProMenuItemLabel button_label,
                           ProMenuLineHelp one_line_help,
                           ProMenuDescription description,
                           ProFileName msg_file);
```

The arguments to this function are:

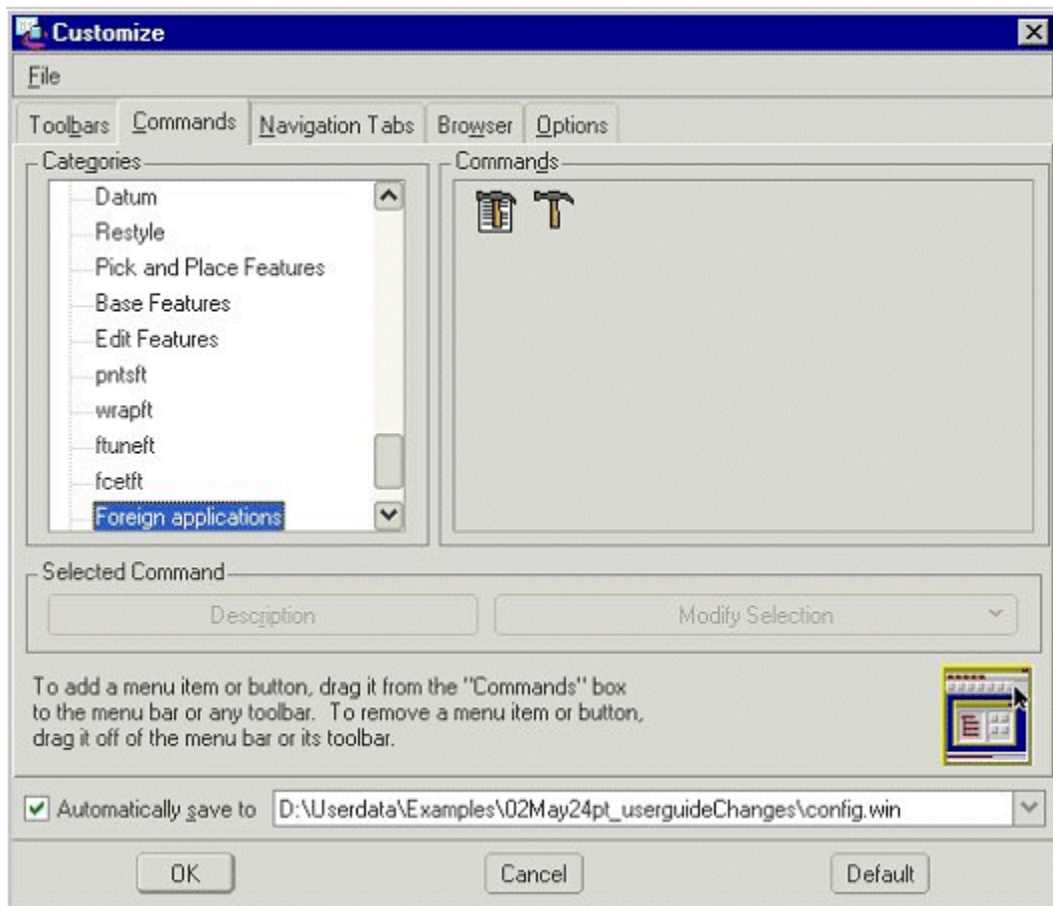
- *cmd_id*—The command identifier.
- *button_label*—The message string that refers to the icon label. This label (stored in the message file) identifies the text seen when the button is displayed. If the command is not assigned an icon, the *button_label* string appears on the toolbar button by default.
- *one_line_help*—The one-line Help for the icon. This label (stored in the message file) identifies the help line seen when the mouse moves over the icon.
- *description*—The message appears in the Screen Customization dialog and also when "Description" is clicked in Pro/ENGINEER.
- *msg_file*—The message file name. All the labels including the one-line Help labels must be present in the message file.

Note: This file must be in the directory *<text_path>/text* or *<text_path>/text/<language>*.

Placing the Toolbar Button

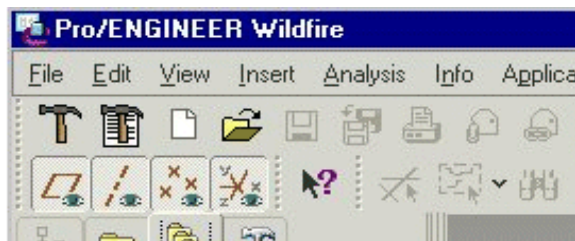
Once the toolbar button has been created using the functions discussed, place the toolbar button on the Pro/ENGINEER toolbar. Click **Tools>Customize Screen**. The designated buttons will be stored under the category “**Foreign Applications**” in the **Commands** tab. Drag the toolbar button on to the Pro/ENGINEER toolbar as shown. Save the window configuration settings in the *config.win* file so that the settings are loaded when a new session of Pro/ENGINEER is launched. For more information, see the Pro/ENGINEER menus portion of the Pro/ENGINEER Help.

Figure 15-1: The Customize Screen With The Icons To be Designated



User Interface: Menus,
Commands, and
Popupmenus

Figure 15-2: The Pro/ENGINEER Toolbar With The Designated Icons



From Pro/ENGINEER Wildfire 5.0 onward, the user interface in the Detailed Drawing module has been restructured. Therefore, existing Pro/TOOLKIT applications, in which commands are designated to be added to toolbars, may display the following changes in behavior:

- If the existing Pro/TOOLKIT application adds buttons to a toolbar that has been redesigned, click **Tools>Customize Screen**. The designated command will be available under the category “**Foreign Applications**” in the **Commands** tab. Drag the toolbar button on to the Pro/ENGINEER toolbar and save the *config.win* file.
- If the menu in which the command is placed before it is designated no longer exists, a new menu pane **Pro/TOOLKIT** will be added to the Pro/ENGINEER menu bar as the right most menu by default. This new menu will contain the command placed by the Pro/TOOLKIT application.

Example 2: Designating a Command

This example code illustrates how to designate a command to be available for placement as a toolbar button.

```
/*----- Pro/Toolkit Includes -----*/
#include <ProToolkit.h>
#include <ProMenu.h>
#include <ProMdl.h>
#include <ProMenuBar.h>
#include <ProUtil.h>
#include <PTAppUnicodeUtils.h>
/*----- Application Includes -----*/
#include <TestError.h>

/*----- Function Prototypes -----*/
int user_initialize(int argc, char *argv[], char *proe_vsn, char *build);
void user_terminate();

/*----- External Data -----*/
extern int UserPartSetup();
extern int UserAssemblySetup();
extern int UserManufactureSetup();
extern int UserDrawingSetup();
extern ProError UserREADMESetup();
extern int UserFEMSetup();
extern ProError UserGuideMain();
extern int UserNewDrawingMenuSetup();
extern int UserUIMenuSetup();

extern int UserPopupmenusSetup();
```



```

extern int UserCustomRelFunctionsDefine();
extern int UserCustomModelChecksDefine();
extern int UserSolidNoteCreate();

int user_initialize(
    int   argc,           /** Number of arguments          **/
    char *argv[],         /** Pro/E arguments          **/
    char *proe_vsn,       /** Pro/E version            **/
    char *build)          /** Pro/E build              **/
{
    int menu,i;
    uiCmdCmdId  cmd_id;
    ProFileName wmsgfil;
    ProError err;

/*-----*\
    Print out Pro/E version and build and set up logging if required
\*-----*/
    ProTKPrintf("\n");
    for(i=0;i<argc;i++)
        ProTKPrintf("\targv[%d]      = %s\n", i, argv[i]);
    ProTKPrintf("\tPro/E version = %s\n", proe_vsn);
    ProTKPrintf("\tPro/E build   = %s\n", build);

    if (argc > 4)
        ProTestErrlogOpen(argv[4], proe_vsn, build);
    else
        ProTestErrlogOpen("pt_userguide", proe_vsn, build);

/*-----*\
    Set up the entry point to the demos
\*-----*/
    ProStringToWstring (wmsgfil, "utilities.txt");

    err = ProMenubarMenuAdd ("UGMENU", "-UG MENU",
"Utilities", PRO_B_TRUE, wmsgfil);
    ERROR_CHECK("user_initialize","ProMenubarMenuAdd()",err);

    err = ProCmdActionAdd("-UG README!",
        (uiCmdCmdActFn)UserREADMESetup,
        uiProe2ndImmediate, NULL,
        PRO_B_TRUE, PRO_B_TRUE, &cmd_id);
    ERROR_CHECK("user_initialize","ProCmdActionAdd()",err);

    err = ProMenubarmenuPushbuttonAdd(
        "UGMENU", "-UG README!", "-UG README!",
        "Enter the README menu of the User Guide",
        NULL, PRO_B_FALSE, cmd_id, wmsgfil);
    ERROR_CHECK("user_initialize","ProMenubarmenuPushbuttonAdd()",err);

/*-----*\

```

```

Register a menu button icon for the command
\*-----*/
    err = ProCmdIconSet (cmd_id, "tkreadme.gif");
    ERROR_CHECK("user_initialize", "ProCmdIconSet()", err);

/*-----*\
Make the command available as a Toolbar button
\*-----*/
    err = ProCmdDesignate (cmd_id, "-UG README!", "Enter the README menu of
the User Guide", "UG README description", wmsgfil);
    ERROR_CHECK("user_initialize", "ProCmdDesignate()", err);

    err = ProCmdActionAdd("-UsrGuide Main",
        (uiCmdCmdActFn)UserGuideMain,
        uiProe2ndImmediate, NULL,
        PRO_B_TRUE, PRO_B_TRUE, &cmd_id);
    ERROR_CHECK("user_initialize", "ProCmdActionAdd()", err);

    err = ProMenubarmenuPushbuttonAdd(
        "UGMENU", "-UsrGuide Main", "-UsrGuide Main",
        "Enter the Main menu of the User Guide.",
        NULL, PRO_B_FALSE, cmd_id, wmsgfil);
    ERROR_CHECK("user_initialize", "ProMenubarmenuPushbuttonAdd()", err);

err = ProCmdIconSet (cmd_id, "tkmain.gif");
    ERROR_CHECK("user_initialize", "ProCmdIconSet()", err);

    err = ProCmdDesignate (cmd_id, "-UsrGuide Main", "Enter the Main menu of
the User Guide.", "UG Main description", wmsgfil);
    ERROR_CHECK("user_initialize", "ProCmdDesignate()", err);

    err = ProCmdActionAdd("-TK Round",
        (uiCmdCmdActFn)UserEdgeRoundCreate,
        uiProe2ndImmediate, NULL,
        PRO_B_TRUE, PRO_B_TRUE, &cmd_id);
    ERROR_CHECK("user_initialize", "ProCmdActionAdd()", err);

    err = ProCmdIconSet (cmd_id, "tkround");
    ERROR_CHECK("user_initialize", "ProCmdIconSet()", err);

    err = ProCmdDesignate (cmd_id, "-UsrGuide Round", "Userguide Round
Example", "UG Round description", wmsgfil);
    ERROR_CHECK("user_initialize", "ProCmdDesignate()", err);

    err = ProCmdActionAdd("-TK Note Create",
        (uiCmdCmdActFn)UserSolidNoteCreate,
        uiProe2ndImmediate, NULL,
        PRO_B_TRUE, PRO_B_TRUE, &cmd_id);
    ERROR_CHECK("user_initialize", "ProCmdActionAdd()", err);

    err = ProCmdIconSet (cmd_id, "tknote");

```

```

ERROR_CHECK("user_initialize", "ProCmdIconSet()", err);

err = ProCmdDesignate (cmd_id, "-UsrGuide Solid Note", "Userguide Solid
Note Example", "UG Solid Note description", wmsgfil);
ERROR_CHECK("user_initialize", "ProCmdDesignate()", err);

/*-----*\
Part menu

\*-----*/
err = ProCmdActionAdd("-UsrGuide Part",
    (uiCmdCmdActFn)UserPartSetup,
    uiProe2ndImmediate, NULL,
    PRO_B_TRUE, PRO_B_TRUE, &cmd_id);
ERROR_CHECK("user_initialize", "ProCmdActionAdd()", err);

err = ProMenubarmenuPushbuttonAdd(
    "UGMENU", "-UsrGuide Part", "-UsrGuide Part",
    "Enter the Part menu of the User Guide.",
    NULL, PRO_B_TRUE, cmd_id, wmsgfil);
ERROR_CHECK("user_initialize", "ProMenubarmenuPushbuttonAdd()", err);

err = ProMenuFileRegister("part", "part.mnu", &menu);
ERROR_CHECK("user_initialize", "ProMenuFileRegister()", err);

err = ProMenuAuxfileRegister("part", "part.aux", &menu);
ERROR_CHECK("user_initialize", "ProMenuAuxfileRegister()", err);

err = ProMenubuttonActionSet("part", "-UG Part",
    (ProMenubuttonAction)UserPartSetup, NULL, 0);
ERROR_CHECK("user_initialize", "ProMenubuttonActionSet()", err);

/*-----*\
Assembly menu

\*-----*/
err = ProCmdActionAdd("-UsrGuide Asm",
    (uiCmdCmdActFn)UserAssemblySetup,
    uiProe2ndImmediate, NULL,
    PRO_B_TRUE, PRO_B_TRUE, &cmd_id);
ERROR_CHECK("user_initialize", "ProCmdActionAdd()", err);

err = ProMenubarmenuPushbuttonAdd(
    "UGMENU", "-UsrGuide Asm", "-UsrGuide Asm",
    "Enter the Assembly menu of the User Guide.",
    NULL, PRO_B_TRUE, cmd_id, wmsgfil);
ERROR_CHECK("user_initialize", "ProMenubarmenuPushbuttonAdd()", err);

err = ProMenuFileRegister("assembly", "assembly.mnu", &menu);
ERROR_CHECK("user_initialize", "ProMenuFileRegister()", err);

```

```

err = ProMenuAuxfileRegister("assembly","assembly.aux",&menu);
ERROR_CHECK("user_initialize","ProMenuAuxfileRegister()",err);

err = ProMenubuttonActionSet("assembly","-UG Assembly",
    (ProMenubuttonAction)UserAssemblySetup,NULL,0);
ERROR_CHECK("user_initialize","ProMenubuttonActionSet()",err);

/*-----*\
FEM menu

\*-----*/
err = ProCmdActionAdd("-UsrGuide FEM",
    (uiCmdCmdActFn)UserFEMSetup,
    uiProe2ndImmediate, NULL,
    PRO_B_TRUE, PRO_B_TRUE, &cmd_id);
ERROR_CHECK("user_initialize","ProCmdActionAdd()",err);

err = ProMenubarmenuPushbuttonAdd(
    "UGMENU", "-UsrGuide FEM", "-UsrGuide FEM",
    "Enter the FEM menu of the User Guide.",
    NULL, PRO_B_TRUE, cmd_id, wmsgfil);
ERROR_CHECK("user_initialize","ProMenubarmenuPushbuttonAdd()",err);

err = ProMenuFileRegister("part","part.mnu",&menu);
ERROR_CHECK("user_initialize","ProMenuFileRegister()",err);

err = ProMenuAuxfileRegister("part","fem_part.aux",&menu);
ERROR_CHECK("user_initialize","ProMenuAuxfileRegister()",err);

err = ProMenubuttonActionSet("part","-UG FEM",
    (ProMenubuttonAction)UserFEMSetup,NULL,0);
ERROR_CHECK("user_initialize","ProMenubuttonActionSet()",err);

err = ProMenuFileRegister("assembly","assembly.mnu",&menu);
ERROR_CHECK("user_initialize","ProMenuFileRegister()",err);

err = ProMenuAuxfileRegister("assembly","fem_assembly.aux",&menu);
ERROR_CHECK("user_initialize","ProMenuAuxfileRegister()",err);

err = ProMenubuttonActionSet("assembly","-UG FEM",
    (ProMenubuttonAction)UserFEMSetup,NULL,0);
ERROR_CHECK("user_initialize","ProMenubuttonActionSet()",err);

/*-----*\
MFG menu

\*-----*/
err = ProCmdActionAdd("-UsrGuide MFG",
    (uiCmdCmdActFn)UserManufactureSetup,
    uiProe2ndImmediate, NULL,

```

```

        PRO_B_TRUE, PRO_B_TRUE, &cmd_id);
ERROR_CHECK("user_initialize", "ProCmdActionAdd()", err);

err = ProMenubarmenuPushbuttonAdd(
    "UGMENU", "-UsrGuide MFG", "-UsrGuide MFG",
    "Enter the MFG menu of the User Guide.",
    NULL, PRO_B_TRUE, cmd_id, wmsgfil);
ERROR_CHECK("user_initialize", "ProMenubarmenuPushbuttonAdd()", err);

err = ProMenuFileRegister("manufacture", "mfg.mnu", &menu);
ERROR_CHECK("user_initialize", "ProMenuFileRegister()", err);

err = ProMenuAuxfileRegister("manufacture", "mfg.aux", &menu);
ERROR_CHECK("user_initialize", "ProMenuAuxfileRegister()", err);

err = ProMenubuttonActionSet("manufacture", "-UG MFG",
    (ProMenubuttonAction)UserManufactureSetup, NULL, 0);
ERROR_CHECK("user_initialize", "ProMenubuttonActionSet()", err);

/*-----*\
Drawing menu

\*-----*/
err = ProCmdActionAdd("-UsrGuide Drw",
    (uiCmdCmdActFn)UserDrawingSetup,
    uiProe2ndImmediate, NULL,
    PRO_B_TRUE, PRO_B_TRUE, &cmd_id);
ERROR_CHECK("user_initialize", "ProCmdActionAdd()", err);

err = ProMenubarmenuPushbuttonAdd(
    "UGMENU", "-UsrGuide Drawing", "-UsrGuide Drawing",
    "Enter the Drawing menu of the User Guide.",
    NULL, PRO_B_TRUE, cmd_id, wmsgfil);
ERROR_CHECK("user_initialize", "ProMenubarmenuPushbuttonAdd()", err);

err = ProCmdActionAdd("-UsrGuide New Drw",
    (uiCmdCmdActFn)UserNewDrawingMenuSetup,
    uiProe2ndImmediate, NULL,
    PRO_B_TRUE, PRO_B_TRUE, &cmd_id);
ERROR_CHECK("user_initialize", "ProCmdActionAdd()", err);

err = ProMenubarmenuPushbuttonAdd(
    "UGMENU", "-UsrGuide New Drawing", "-UsrGuide New Drawing",
    "Enter the New Drawing menu of the User Guide.",
    NULL, PRO_B_TRUE, cmd_id, wmsgfil);
ERROR_CHECK("user_initialize", "ProMenubarmenuPushbuttonAdd()", err);

err = ProMenuFileRegister("drawing", "draw.mnu", &menu);
ERROR_CHECK("user_initialize", "ProMenuFileRegister()", err);

err = ProMenuAuxfileRegister("drawing", "draw.aux", &menu);

```

```

ERROR_CHECK("user_initialize", "ProMenuAuxfileRegister()", err);

err = ProMenubuttonActionSet("drawing", "-UG Drawing",
    (ProMenubuttonAction)UserDrawingSetup, NULL, 0);

err = ProMenubuttonActionSet("drawing", "-UG New Drawing",
    (ProMenubuttonAction)UserNewDrawingMenuSetup, NULL, 0);

ERROR_CHECK("user_initialize", "ProMenubuttonActionSet()", err);

err = ProCmdActionAdd("-UsrGuide UI",
    (uiCmdCmdActFn)UserUIMenuSetup,
    uiProe2ndImmediate, NULL,
    PRO_B_TRUE, PRO_B_TRUE, &cmd_id);
ERROR_CHECK("user_initialize", "ProCmdActionAdd()", err);

err = ProMenubarmenuPushbuttonAdd(
    "UGMENU", "-UsrGuide UI", "-UsrGuide UI",
    "Enter the UI menu of the User Guide.",
    NULL, PRO_B_TRUE, cmd_id, wmsgfil);
ERROR_CHECK("user_initialize", "ProMenubarmenuPushbuttonAdd()", err);

err = ProMenuFileRegister("drawing", "draw.mnu", &menu);
ERROR_CHECK("user_initialize", "ProMenuFileRegister()", err);

err = ProMenuAuxfileRegister("drawing", "ui.aux", &menu);
ERROR_CHECK("user_initialize", "ProMenuAuxfileRegister()", err);

err = ProMenubuttonActionSet("drawing", "-UG UI",
    (ProMenubuttonAction)UserUIMenuSetup, NULL, 0);
ERROR_CHECK("user_initialize", "ProMenubuttonActionSet()", err);

/*-----*\
    Initialize popup menus for selected examples
\*-----*/
UserPopupmenusSetup();

/*-----*\
    Initialize custom relation functions defined in this application
\*-----*/
UserCustomRelFunctionsDefine();

/*-----*\
    Initialize ModelCheck custom checks defined in this application
\*-----*/
UserCustomModelChecksDefine();

/*-----*\
Call init functions for menu examples
\*-----*/

```

```

err = ProUserMenubuttonMoveInit();
ERROR_CHECK("user_initialize", "ProUserMenubuttonMoveInit", err);
err = ProUserAddMenuInit();
ERROR_CHECK("user_initialize", "ProUserAddMenuInit", err);
err = ProUserConfirmationInit();
ERROR_CHECK("user_initialize", "ProUserConfirmationInit", err);

return(0);
}

void user_terminate()
{
}

```

Popup Menu

Pro/ENGINEER provides shortcut menus that contain frequently used commands appropriate to the currently selected items. You can access a shortcut menu by clicking the right mouse button (RMB) after selecting an item. Shortcut menus are accessible in:

- Graphics window
- Model tree
- Some dialog boxes
- Any area where you can perform an object-action operation by selecting an item and then choosing a command to perform on the selected item.

Pro/TOOLKIT provides different procedures to add custom buttons to popup menus, depending on the buttons' context. To add to the model tree popup menu, use the procedure described in Adding a Button to the Model Tree Popup Menu. To add a popup menu to a custom dialog box, see the User Interface: Dialogs chapter (you cannot modify the popup menus in an existing UI dialog box). To add to a graphics window popup menu, refer to Adding a Popup Menu to the Graphics Window.

Adding a Popup Menu to the Graphics Window

Different popup menus can be activated during a given session of Pro/ENGINEER. Every time the Pro/ENGINEER context changes (by opening a different model type, by entering different tools, or by entering special modes like "Edit") a different popup menu is created. When Pro/ENGINEER moves to the next context, the popup menu may be destroyed.

Because of this, Pro/TOOLKIT applications must attach a button to the popup menu during initialization of the popup menu. The Pro/TOOLKIT application is notified each time a particular popup menu is created, which then allows the user to add to the popup menu.

Use the following procedure to add items to graphics window popup menus:

1. Obtain the name of the existing popup menus to which you want to add a new menu using the trail file.
2. Register the Pro/TOOLKIT notifications for creation and destruction of popup menus.
3. Create commands for the new popup menu items.
4. Implement access functions to provide visibility information for the items.
5. When the notification is called for the desired popup menu, add the custom buttons to the menu.
6. When the destroy notification is called, free the associated memory for the custom buttons.

The following sections describe each of these steps in detail. You can add push buttons, check buttons, and cascade menus to the popup menu. You can add popup menu items only in the active window. You cannot use this procedure to remove items from existing menus. To remove items using an access function see the section on Manipulating Existing Commands.

Using the Trail File to Determine Existing Popup Menu Names

The trail file in Pro/ENGINEER contains a comment that identifies the name of the popup menu if the configuration option `auxapp_popup_menu_info` is set to yes.

For example, the popup menu, **Edit Properties**, has the following comment in the trail file:

```
~ Close `rmb_popup` `PopupMenu`  
~ Activate `rmb_popup` `EditProperties`  
!Command ProCmdEditPropertiesDtm was pushed from the  
software.  
!Item was selected from popup menu 'popup_mnu_edit'
```


Registering Notifications to Create and Destroy Popup Menus

Popup menus are created at runtime in Pro/ENGINEER and only when the required menus exist in the active window. Hence it is not possible to pre-register the custom buttons. Notification functions notify applications to add a new popup menu. For more information on using notifications see the chapter Event-driven Programming: Notifications.

Functions Introduced:

- **ProNotificationSet()**
- **ProPopupmenuCreatePostAction()**
- **ProPopupmenuDestroyPreAction()**

If the notification value argument type is set to **PRO_POPMENU_CREATE_POST**, a registered callback function whose signature matches **ProPopupmenuCreatePostAction()** is called. This function is called after the popup menu is created internally in Pro/ENGINEER and must be used to assign application-specific buttons to the popup menu.

If the notification value argument type is set as **PRO_POPUPMENU_DESTROY_PRE**, a registered callback notification function **ProPopupmenuDestroyPreAction()** is called before the popup menu is destroyed. Use this function to free memory allocated by the application for the custom buttons in the popup menu.

Accessing the Popup Menu

The functions described in this section provide the name and ID of the popup menus that are used to access these menus while using other functions.

Functions Introduced:

- **ProPopupmenuIdGet()**
- **ProPopupmenuNameGet()**

The function **ProPopupmenuIdGet()** returns the popup menu ID for a given popup menu name. The popup menu ID is required for the functions that add buttons to the popup menu. IDs are dependent on the context of Pro/ENGINEER and are not maintained between sessions.

The function **ProPopupMenuNameGet()** returns the name of the popup menu assigned to a given ID.

Creating Commands for the New Popup Menu Buttons

Functions Introduced:

- **ProCmdActionAdd()**
- **ProCmdOptionAdd()**
- **ProCmdCmdIdFind()**

The functions **ProCmdActionAdd()** or **ProCmdOptionAdd()** are used to create commands for the popup menus. Only push buttons (action commands) and check buttons (option commands) are supported in popup menus. Commands with a given name are created only once in a session of Pro/ENGINEER. Hence PTC recommends that you create the required commands in the **user_initialize()** function of the application.

Use **ProCmdCmdIdFind()** to obtain the command ID for the command (in the notification callback for PRO_POPUPMENU_CREATE_POST) to add the button to the popup menu.

Checking the Access State of a Popup Menu Item

Functions Introduced:

- **ProPopupMenuAccessFunction()**

A popup menu uses an additional access function to determine whether the popupmenu must be visible based on the currently selected items. Use the function whose signature matches **ProPopupMenuAccessFunction()** to set the access state of the button in the popup menu.

The syntax for this function is as follows:

```
typedef uiCmdAccessState (*ProPopupMenuAccessFunction)
(uiCmdCmdId command,
 ProAppData appdata,
 ProSelection* sel_buffer);
```

The last argument contains an array of selected items that are used to determine the visibility of the popup menu button. It is PTC standard practice to remove popup menu buttons using ACCESS_REMOVE instead of graying them out using ACCESS_UNAVAILABLE when unusable item types have been selected. This is to minimize the size of the popup menu.

Adding Pro/ENGINEER Popup Menus

Functions Introduced:

- **ProPopupmenuButtonAdd()**
- **ProPopupmenuCascadebuttonAdd()**

Use **ProPopupmenuButtonAdd()** to add a new item to a popup menu. The input arguments are:

- *menu_ID*—Specifies the ID of the popup menu obtained from **ProPopupmenuIdGet()**.
- *position*—Specifies the position in the popup menu at which to add the menu button. Pass PRO_VALUE_UNUSED to add to the bottom of the menu.
- *button_name*—Specifies the name of the added button. The button name is placed in the trail file when the user selects the menu button. For more information on valid characters that you can use to specify the name, refer to the section Naming Convention for UI Components.
- *button_label*—Specifies the message that refers to the button label. This label identifies the text seen when the button is displayed. To localize this text, obtain and pass a string from **ProMessageToBuffer()**.
- *button_helptext*—Specifies the help message associated with the button. This label acts as a keyword that identifies the help text in the message file. To localize this text, obtain and pass a string from **ProMessageToBuffer()**.
- *cmd_ID*—Specifies the command identifier for the action or option.
- *access_status*—Specifies the callback function used to determine the visibility status of the added button.
- *appdata*—Specifies the user application data.

Use the function **ProPopupmenuCascadebuttonAdd()** to add a new cascade menu to an existing popup menu.

The input arguments are:

- *menu_ID*—Specifies the ID of the popup menu obtained from **ProPopupmenuIdGet()**.
- *position*—Specifies the position in the menu at which to add the cascade button. Pass PRO_VALUE_UNUSED to add to the bottom of the menu.

- *cascade_menu_name*—Specifies the name of the cascade menu. The name is placed in the trail file when the user selects the menu button. For more information on valid characters that you can use to specify the name, refer to the section Naming Convention for UI Components.
- *cascade_menu_label*—Specifies the message that refers to the cascade menu label. This label identifies the text seen when the menu is displayed. To localize this text, obtain and pass a string from **ProMessageToBuffer()**.
- *cascade_menu_helptext*—Specifies the help message associated with the cascade menu. This label acts as a keyword that identifies the help text in the message file. To localize this text, obtain and pass a string from **ProMessageToBuffer()**.
- *access_status*—Specifies The callback function used to determine the visibility status of the added item.
- *appdata*—Specifies the user application data.

The output argument is *casc_menuId*, the menu ID of the cascade menu.

Adding a Button to the Model Tree Popup Menu

To add a button to an existing menu bar menu, refer to section Menu Bar Buttons and Menus. Ensure that the following conditions are met:

- The menu name used should be “ActionMenu.”
- The command access function should be configured to read the selection buffer using **ProSelbufferSelectionsGet()**. If the buffer is inactive or empty, the access function should make the menu item invisible.

Example 3: Assigning Pro/ENGINEER command to popup menus

The following example demonstrates how to assign a Pro/ENGINEER command to the popup menus in the graphics window and the model tree. It adds the example **UserAsmcompConstraintsHighlight()** to the popup menu (see the chapter on Assembly: Basic Assembly Access for details). The popup menus use an access function to check the currently selected item in the selection buffer that is an assembly component. If it is not an assembly component, the button will be removed from the menu. The example adds the popup menu button using **ProPopupMenuButtonAdd()** if the menu name is the name of the current main popup menu in assembly mode.

```
#include <ProToolkit.h>
#include <ProNotify.h>
#include <ProPopupmenu.h>
#include <ProSelbuffer.h>
#include <ProMenuBar.h>

#define OA_MSGFILE L"pt_ug_oa.txt"
#define MDLTREE_MENU "ActionMenu"
#define CMD_UGASMPCONSTRSHighlight "UGAsmcompConstrsHighlight"

/*=====*\
Function: UserPopupmenusSetup
Purpose : Setup popup menus for selected examples
\*=====*/
int UserPopupmenusSetup()
{
    ProError status;
    uiCmdCmdId cmd_id;

/*-----*\
    Create commands for Object/Action wrappers around old-style examples
    that aren't setup for Object/Action
\*-----*/
    status = ProCmdActionAdd ("UGAsmcompConstrsHighlight",
        (uiCmdCmdActFn)UserAsmcompConstraintsHighlight_OA,
        uiProe2ndImmediate,
        UserAsmcompConstraintsHighlight_TestMdlTree,
        PRO_B_FALSE, PRO_B_FALSE, &cmd_id);
    ERROR_CHECK ("UserPopupmenusSetup", "ProCmdActionAdd()", status);

/*-----*\
    Add Object/Action commands to the model tree popup menu
\*-----*/
    status = ProMenubarmenuPushbuttonAdd (MDLTREE_MENU,
"UGMdlTree.ConstrsHighlight",
"UGAsmcompConstrsHighlightLabel",
"UGAsmcompConstrsHighlightHelp",
NULL, PRO_B_TRUE,
cmd_id, OA_MSGFILE);
    ERROR_CHECK ("UserPopupmenusSetup", "ProMenubarmenuPushbuttonAdd()",
        status);

/*-----*\
    Add the popup menu notification to the session
\*-----*/
    status = ProNotificationSet (PRO_POPUPMENU_CREATE_POST,
        (ProFunction)UserPopupmenuNotification);
    ERROR_CHECK ("UserPopupmenusSetup", "ProNotificationSet()", status);

    return(0);
}
```

```

/*=====*\
Function: UserPopupMenuNotification
Purpose: Add buttons to popup menus when they are created.
\*=====*/
ProError UserPopupMenuNotification (ProMenuName name)
{
    ProPopupMenuId menu_id;
    uiCmdCmdId cmd_id;
    ProError status;
    ProLine label;
    ProLine help;

/*-----*\
    Check if the menu being created matches the menu name we want.
\*-----*/
    ProPopupMenuIdGet (name, &menu_id);

    if (strcmp (name, "Sel Obj Menu") == 0)
    {
        status = ProCmdCmdIdFind ("UGasmcompConstrsHighlight", &cmd_id);

/*-----*\
        Extract the button label and helptext from a message file.
\*-----*/
        status = ProMessageToBuffer (label, OA_MSGFILE,
            "UGasmcompConstrsHighlightLabel");
        status = ProMessageToBuffer (help, OA_MSGFILE,
            "UGasmcompConstrsHighlightHelp");

/*-----*\
        Add the button to the end of the popup menu.
\*-----*/
        status = ProPopupMenuButtonAdd (menu_id, PRO_VALUE_UNUSED,
            "UGPM.ConstrsHighlight",
            label, help,
            cmd_id,
            UserAsmcompConstraintsHighlight_TestPM,
            NULL);
        ERROR_CHECK ("UserPopupMenuNotification",
            "ProPopupMenuButtonAdd()", status);

    }

    return PRO_TK_NO_ERROR;
}

#define OA 1 /* Standard OA in the menu system, typically grays out
the button when not usable */

```

```

#define PM 2 /* OA in a popup menu, typically remove the button when
             not usable */

/*=====*\
FUNCTION: UserAsmcompConstraintsHighlight_TestMdlTree
PURPOSE: Test function for access for constraint highlighting from model
         tree RMB popup menu.
\*=====*/
uiCmdAccessState
UserAsmcompConstraintsHighlight_TestMdlTree(uiCmdAccessMode mode)
{
    return UserAsmcompConstraintsHighlight_TestLow (NULL, PM);
}

/*=====*\
FUNCTION: UserAsmcompConstraintsHighlight_TestPM
PURPOSE: Test function for access for constraint highlighting from
         graphics window RMB popup menu.
\*=====*/
uiCmdAccessState UserAsmcompConstraintsHighlight_TestPM(uiCmdCmdId id,
                                                         ProAppData data,
                                                         ProSelection* sels)
{
    return UserAsmcompConstraintsHighlight_TestLow (sels, PM);
}

/*=====*\
FUNCTION: UserAsmcompConstraintsHighlight_TestLow
PURPOSE: Test function for access for constraint highlighting
\*=====*/
uiCmdAccessState UserAsmcompConstraintsHighlight_TestLow (ProSelection*
sels,
int mode)
{
    uiCmdAccessState access_result;
    ProBoolean should_free = PRO_B_FALSE;
    ProError status;
    int size;

/*-----*\
    Set the default return if the button is unusable.
\*-----*/
    if (mode == OA)
        access_result = ACCESS_UNAVAILABLE;
    else
        access_result = ACCESS_REMOVE;

/*-----*\
    If called without selections, extract the current selections from
    the buffer.
\*-----*/

```

```

if (sels == NULL)
{
    status = ProSelbufferSelectionsGet (&sels);

    if (status != PRO_TK_NO_ERROR)
        return access_result;

    if (sels == NULL)
        return access_result;

    should_free = PRO_B_TRUE;
}

/*-----*\
This command allows only one selection.
\*-----*/
status = ProArraySizeGet (sels, &size);

if (status != PRO_TK_NO_ERROR)
    return access_result;

if (size == 1)
{
    ProAsmcomp asmcomp;

    status = ProSelectionModelitemGet (sels [0], &asmcomp);

/*-----*\
If the selected type is feature,its feature type must be component.
\*-----*/
    if (asmcomp.type == PRO_FEATURE)
    {
        ProFeattype ftype;

        status = ProFeatureTypeGet (&asmcomp, &ftype);

        if (ftype == PRO_FEAT_COMPONENT)
        {
            access_result = ACCESS_AVAILABLE;
        }
    }

/*-----*\
If the selected type is part or assembly,it must have a parent
assembly to use to construct the component feature handle.
\*-----*/
    if (asmcomp.type == PRO_PART || asmcomp.type == PRO_ASSEMBLY)
    {
        ProAsmcomppath path;

        status = ProSelectionAsmcomppathGet (sels [0], &path);
    }
}

```



```

        if (path.table_num > 0)
        {
            access_result = ACCESS_AVAILABLE;
        }
    }
}

if (should_free)
    ProSelectionarrayFree (sels);

return access_result;
}

#undef OA
#undef PM

```

Mode-Specific Buttons and Menus

The PART menu is displayed only when Pro/ENGINEER is in Part mode, which occurs when a part has been created or retrieved. Consequently, the PART menu and its buttons are called “mode-specific.” Modifying and supplementing the mode-specific Pro/ENGINEER interface is fundamentally different from similar operations on the menu bar menus.

This section describes the files and functions necessary to manipulate the mode-specific buttons and menus of Pro/ENGINEER. This section covers the following topics:

- Menu Files
- Adding a Menu Button
- New Menus
- Preempting Pro/ENGINEER Commands
- Submenus
- Manipulating Menus
- Data Menus
- Setting Menu Buttons
- Controlling Accessibility of Menu Buttons
- Pushing and Popping Menus
- Run-time Menus

Menu Files

Menu files enable you to specify your own text for the name of a menu button and the one-line help text that appears when you place the cursor over that button, along with translations for both of these.

Pro/ENGINEER looks for the Pro/TOOLKIT menu files in the following locations:

- The current Pro/ENGINEER startup directory
- The subdirectory `text/menus` under the directory named by the `text_dir` statement in the registry file

PTC recommends that during development you place your menu files in `text/menus` under your working directory and specify the following registry file entry:

```
text_dir .
```

Names and Contents of Menu Files

There are two conventional extensions used in naming menu files:

- `.mnu`—Files that describe complete menus
- `.aux`—Files that describe new buttons to be added to existing Pro/ENGINEER menus

The following restrictions apply to file names:

- The name must be unique through out Pro/ENGINEER.
- The name must have no more than 30 characters, including the extension.

To find out what menu file names are used by Pro/ENGINEER, look in the Pro/ENGINEER menu directory, `text/usascii/menus`, under the loadpoint.

When you create an `.aux` file to extend an existing Pro/ENGINEER menu, use the same file name root as Pro/ENGINEER used for that menu.

Syntax and Semantics of Menu Files

The two types of files—`.mnu` and `.aux`—have identical formats.

The format consists of groups of three lines (one group for each menu button) and a group at the top for the menu title. The title group contains the menu title on the first line, and then two blank lines.

The menu title is the name that appears at the top of the menu when you run Pro/ENGINEER in English. The menu title is also used to refer to the menu from your Pro/TOOLKIT code, so it is essential that this name is unique in all Pro/ENGINEER menus. For example, if you are writing an *.aux* file to add buttons to a Pro/ENGINEER menu, make sure you use the title that appears in the corresponding *.mnu* file in Pro/ENGINEER. If you are creating a new menu, make sure that the title you use has not already been used in Pro/ENGINEER.

If the menu title is followed by a second word, Pro/ENGINEER displays the second word instead of the first one. This is how a translation is provided. If there is no second word, Pro/ENGINEER displays the first word.

Each menu button group consists of the following three lines:

- *Button name*—If the button name as it appears on the Pro/ENGINEER screen contains spaces, each space must be replaced by the character # in the menu file. If the button name is followed by another name, separated by white space, the second name will be what is actually displayed.

The first name is still used to refer to the button from your Pro/TOOLKIT code. The second provides an optional translation of that button name.

- *One-line Help*—This is a single line of text that explains what the menu button does. When you place the mouse pointer on the menu button, Pro/ENGINEER displays the one-line Help text in the Message Window.
- *Alternate Help*—If this line is not blank (or does not start with the comment character “#”), it will be used in place of the one-line Help. This provides a translation of the Help message.

Example 4: Sample Menu File

The following example code shows the menu file you would create to add a new button, **Check Part**, to the Pro/ENGINEER PART menu.

```
Menu file "part.aux":
[Start of file on next line]
PART
<blank line>
<blank line>
```

```
Check#Part
Check the validity of the current part.
<blank line>
[End of file on previous line]
```

Example 5: Adding Alternate Names and Help Text to a Button

This example code creates an alternate button name and Help text for the previous example.

```
Menu file "part.aux":
[Start of file on next line]
PART
<blank line>
<blank line>
Check#Part DRC#Check
Check the validity of the current part.
Perform a DRC (Design Rule Check) on the part.
[End of file on previous line]
```

Adding a Menu Button

Functions Introduced:

- **ProMenuFileRegister()**
- **ProMenuAuxfileRegister()**
- **ProMenubuttonActionSet()**
- **ProMenubuttonGenactionSet()**

When you add a new button to an existing menu in **user_initialize()**, you are modifying the Pro/ENGINEER definition of the menu in its memory before that menu has been used by Pro/ENGINEER, and therefore before Pro/ENGINEER has loaded it from its menu file. You must call the function **ProMenuFileRegister()** to tell Pro/ENGINEER to load its own menu file before you can add your own buttons.

To add a button to a menu, first write a menu file, and then add the following calls to **user_initialize()**:

1. Load the Pro/ENGINEER menu into memory, using **ProMenuFileRegister()**.
2. Add the buttons in your menu file to the menu, using **ProMenuAuxfileRegister()**.
3. Define the actions of the new buttons, using **ProMenubuttonActionSet()**.

Calling ProMenuFileRegister()

The input arguments to **ProMenuFileRegister()** are as follows:

- *ProMenuName menuname*—The unique title of the menu that appears as the first word on the first line of the menu file and on the heading of the menu on the screen when you run Pro/ENGINEER in English. This argument is case-insensitive.
- *ProMenufileName filename*—The name of the menu file, including the extension but not the directory.

The function outputs the integer identifier of the menu, which you do not normally need. If the function fails for some reason (for example, the menu file did not exist), it returns PRO_TK_GENERAL_ERROR. If you call this function a second time on the same menu file, it has no effect.

Calling ProMenuAuxfileRegister()

This function has the same arguments and return value as **ProMenuFileRegister()**. Instead of loading a new menu into memory, the function adds the buttons in the file to a menu already in memory.

Calling ProMenubuttonActionSet()

The first three arguments to **ProMenubuttonActionSet()** are as follows:

- *ProMenuName menuname*—The title of the menu that contains the button.
- *ProMenubuttonName button*—The first name for the button in the menu file (not the second, which provides the translation), but with spaces instead of pound signs (#). This argument is case-insensitive.
- *ProMenubuttonAction action*—A pointer to the Pro/TOOLKIT callback function to be called when the user selects this menu button. To pass a pointer to a function, supply the name of the function without the following parentheses. If your function does not precede the call to **ProMenubuttonActionSet()** in the same file, you must add a declaration of it to show the compiler that this is a function.

The other two arguments, *app_data* and *app_int*, are optional arguments to your command function. These arguments enable your command function to be more flexible in what it does. If you do not want to use *app_data* and *app_int*, supply the values NULL and 0, respectively.

Sample declarations and the use of the optional arguments are shown in Example 6: Adding a Menu Button to a Pro/ENGINEER Menu; Example 7: Defining a New Menu that Closes Itself; and Example 8: Defining a New Menu the User Must Close.

Example 6: Adding a Menu Button to a Pro/ENGINEER Menu

This example code adds the button **Check Part** to the Pro/ENGINEER PART menu. The example uses the menu file from the previous examples.

```
ProUserAddMenuInit()
{
    ProError err;
    int      menuId;
/*-----*\
    Declare the command functions used here.
/*-----*/
    int ProCheckPart (void *a, int b);
/*-----*\
    Load the menu files for the Part menu.
/*-----*/
    ProMenuFileRegister ("part", "part.mnu", &menuId);
    ProMenuAuxfileRegister ("part", "part.aux", &menuId);
/*-----*\
    Define the new Part menu buttons
/*-----*/
    ProMenubuttonActionSet ("part", "Check Part",
        ProCheckPart, NULL, 0);
}
/*=====*\
FUNCTION: ProCheckPart
PURPOSE: Perform a check on a part.
/*=====*/
int ProCheckPart (void *a, int b)
{
    .
    .
    .
}
```

New Menu

Functions Introduced:

- **ProMenuProcess()**
- **ProMenuDelete()**
- **ProMenuCreate()**
- **ProMenuHold()**
- **ProMenuDeleteWithStatus()**

Pro/TOOLKIT enables you to create new menus. Defining a new menu differs from adding buttons to an existing menu in the following ways:

- The menu file you supply should end in *.mnu*, not *.aux*. (It has the same syntax, though.)
- You do not need to call **ProMenuAuxfileRegister()** because the whole menu is defined in a single menu file.
- You need to define an exit action for the menu, in addition to an action for each button on the menu.
- You can either specify the new menu in **user_initialize()** or you can set up the new menu locally before you use it.

Exit Actions

You must not only tell the menu manager inside Pro/ENGINEER which function to call for each button on your menu, but also which function to call if the user selects a button on another menu. This function is called an exit action because it is often used to close the menu.

Note: If you do not define an exit action, Pro/ENGINEER's behavior is undefined if the user selects from another menu.

There are two types of exit action:

- Nothing—The menu selection is ignored. This is useful if you want the user to take some definite action before leaving the current menu.
- Close the current menu—The menus unwind to the level of the menu selected and the selected command is entered. This is the usual way to leave a menu.

Defining a New Menu

To define a new menu, first write a menu file. Before you need to use the menu, add the following calls to your Pro/TOOLKIT program:

1. Load the Pro/ENGINEER menu into memory, using **ProMenuFileRegister()**.
2. Define the actions of the new buttons, using the function **ProMenubuttonActionSet()**.
3. Define the exit action of the new menu, using the functions **ProMenubuttonActionSet()** and one of the exit action functions described in the following section.

Defining an Exit Action

To define an exit action, make an extra call to **ProMenubuttonActionSet()**, but instead of the button name (the third argument), specify the menu name.

If you want the menus to unwind and the new command to be entered, use **ProMenuDelete()** as the action function.

If you want the selection to be ignored, use the function **ProMenuHold()** as the exit action. If you use this function, you must provide some other exit route for the menu. For example, you can specify an explicit menu button (such as **Done**) whose command function calls **ProMenuDelete()**.

If you want to perform some additional action in these cases (such as sending a warning to the user), you can provide your own exit function that performs the action and then calls **ProMenuHold()**.

Example 7: Defining a New Menu that Closes Itself

This example code defines a new menu, MYMENU, that closes itself using the function **ProMenuDelete()**.

```
[Start of file on next line]
MYMENU
<blank line>
<blank line>
Partial#Check
Perform a partial check on the part.
<blank line>
Full#Check
Perform a full check on the part.
<blank line>
```


[End of file on previous line]

The following code sets up the menu:

```
int menuId;

ProMenuFileRegister ("mymenu", "mymenu.mnu", &menuId);
ProMenubuttonActionSet ("mymenu", "Partial Check", ProCheckPart,
    NULL, 0);
ProMenubuttonActionSet ("mymenu", "Full Check", ProCheckPart, NULL,
    1);
ProMenubuttonActionSet ("mymenu", "Quit Checks",
    (ProMenubuttonAction)ProMenuDelete, NULL, 0);
ProMenubuttonActionSet ("mymenu", "mymenu",
    (ProMenubuttonAction)ProMenuDelete, NULL, 0);
```

Example 8: Defining a New Menu the User Must Close

In the following example code, the user has to close MYMENU.

```
int menuId;

ProMenuFileRegister ("mymenu", "mymenu.mnu", &menuId);
ProMenubuttonActionSet ("mymenu", "Partial Check", ProCheckPart,
    NULL, 0);
ProMenubuttonActionSet ("mymenu", "Full Check", ProCheckPart,
    NULL, 1);
ProMenubuttonActionSet ("mymenu", "Quit Checks",
    (ProMenubuttonAction)ProMenuDelete, NULL, 0);
ProMenubuttonActionSet ("mymenu", "mymenu",
    (ProMenubuttonAction)ProMenuHold, NULL, 0);
```

Using a New Menu

After you have defined your new menu, you need to know how to use it. This is normally done inside the command function of another menu button.

To Use a New Menu

1. Display the menu, using **ProMenuCreate()**.
2. Make the menu active so the user can select from it, using **ProMenuProcess()**.

Calling ProMenuCreate()

The first argument to **ProMenuCreate()** is either **PROMENUTYPE_MAIN** or **PROMENUTYPE_SUB**. The usual choice is **PROMENUTYPE_MAIN** (see the section Submenus for detailed information about submenus). The second argument is the title of the menu. The last argument is the identifier of the displayed menu.

Calling ProMenuProcess()

The function **ProMenuProcess()** takes a single input argument—the title of the menu. If the menu is the last one displayed, you can pass an empty string. The return value is meaningful only if you use the function **ProMenuDeleteWithStatus()** as the exit action for the menu.

The function **ProMenuProcess()** returns only when the menu is closed, as the result of a call to either **ProMenuDelete()** or **ProMenuDeleteWithStatus()**. The following is true for any code following the call to **ProMenuProcess()**:

1. The code does not get executed until the menu is closed.
2. The code gets executed before any command that causes an exit from the menu. When the user closes a menu by selecting another command, that command is put into the input buffer and is not executed until control passes from your application back to Pro/ENGINEER.

Example 9: Using a New Menu

The following example code shows how to use the functions **ProMenuProcess()** and **ProMenuCreate()**. The example builds on the previous examples.

```
int action, menuId;
.
.
.
ProMenuCreate (PROMENUTYPE_MAIN, "mymenu", &menuId);
ProMenuProcess ("", &action);
.
.
```

Creating a Menu for Selecting a Single Value

Function Introduced:

- **ProMenuDeleteWithStatus()**
- **ProMenubuttonActionSet()**
- **ProMenuProcess()**

Use of ProMenubuttonActionSet() Final Arguments

The two last arguments of **ProMenubuttonActionSet()** are *app_data*, of type *ProAppData* and *app_int*, of type integer. These arguments are passed directly to your callback function when it is invoked. Because Pro/TOOLKIT and Pro/ENGINEER do not look at these arguments, you can use them for any information that you want to pass to or from your function.

Example 10: Creating a Menu that Selects a Value uses the final argument of **ProMenubuttonActionSet()** to distinguish between several menu buttons that share the same command function. Inside the command function, this value appears as the second argument. It is used to determine which button was selected and then perform the appropriate action. The command function does not use the fourth argument of **ProMenubuttonActionSet()**, but includes a dummy first argument of type *ProAppData* to match it, so that the second argument is received correctly.

Returning a Value from ProMenuProcess()

The function **ProMenuDelete()** closes the current menu and causes control to return from the call to **ProMenuProcess()** that made that menu active. If you want to close the menu under more than one condition and react to that condition in the code that follows the return from **ProMenuProcess()**, use **ProMenuDeleteWithStatus()** instead of **ProMenuDelete()**. The **ProMenuDeleteWithStatus()** function takes a single integer argument, which is the value returned by **ProMenuProcess()**.

Example 10: Creating a Menu that Selects a Value

The following example code shows several new techniques for using the menu functions. This example shows how to use **ProMenuDeleteWithStatus()** and uses more of the arguments to **ProMenubuttonActionSet()**.

```
/*-----*\
    The three values from which to choose.
\*-----*/
```

```

#define EX3_QUIT          -1
#define EX3_VALUE1        1
#define EX3_VALUE2        2
#define EX3_VALUE3        3

ProUserValueMenuCreate (void *a, int b)
{
    int action, menuId;
    int ProUserValueGet (void *dummy, int value);

    int value;
/*-----*\
    Set up the value menu.
\*-----*/
    ProMenuFileRegister ("value", "value.mnu", &menuId);
    ProMenubuttonActionSet ("value", "Value 1",
        ProUserValueGet, NULL, EX3_VALUE1);
    ProMenubuttonActionSet ("value", "Value 2", ProUserValueGet,
        NULL, EX3_VALUE2);
    ProMenubuttonActionSet ("value", "Value 3", ProUserValueGet,
        NULL, EX3_VALUE3);
    ProMenubuttonActionSet ("value", "Quit",
        (ProMenubuttonAction)ProMenuDelete, NULL, EX3_QUIT);
    ProMenubuttonActionSet ("value", "value",
        (ProMenubuttonAction)ProMenuHold, NULL, 0);
/*-----*\
    Use the value menu.
\*-----*/
    ProMenuCreate (PROMENUTYPE_MAIN, "value", &menuId);
    value = ProMenuProcess ("", &action);
    if (value == EX3_QUIT)
        return(0);
}
/*=====*\
FUNCTION: ProUserValueGet
PURPOSE:  Close a menu and return the selected value.
\*=====*/
int ProUserValueGet (void *dummy, int value)
{
    ProMenuDeleteWithStatus (value);
    return (0);
}

```

Compound Menus

Function Introduced:

- **ProCompoundmenuCreate()**

The **ProCompoundmenuCreate()** function enables you to take an array of previously loaded menu names and append them together into one menu.

To Create a Compound Menu:

1. Specify which submenus to include in the compound menu, as follows:

```
static char **compound_menu = {"MENU_1", "MENU_2",  
"MENU_3", ""};
```

2. Load the actions on the buttons.
3. Set the button visibility and accessibility.
4. Generate the compound menu, as follows:

```
ProCompoundmenuCreate (compound_menu, n_submenus);
```

5. Get user input, as follows:

```
ProMenuProcess (compound_menu[0], action);
```

User Interface: Menus,
Commands, and
Popupmenus

Preempting Pro/ENGINEER Commands

Functions Introduced:

- **ProMenubuttonPreactionSet()**
- **ProMenubuttonPostactionSet()**

In addition to adding your own menus and menu buttons, it is sometimes useful to be able to modify the effect of an existing Pro/ENGINEER menu button. The function **ProMenubuttonPreactionSet()** enables you to call your function before the Pro/ENGINEER command is executed. If the operation is before the upcoming command execution, and you want to cancel the upcoming command execution, return 1. Otherwise, return zero.

You could also cancel the Pro/ENGINEER command, so only your function gets called. Similarly, the function **ProMenubuttonPostactionSet()** enables you to call your function after the Pro/ENGINEER command is executed.

You can use the **ProMenubuttonPreactionSet()** function to protect certain commands, so that the user can use them only under certain circumstances specified by your Pro/TOOLKIT application. For example, you may want to prevent the user from saving a model unless it has passed a certain validity check.

Calling ProMenubutton*actionSet()

The functions **ProMenubuttonPreactionSet()** and **ProMenubuttonPostactionSet()** have the same arguments as **ProMenubuttonActionSet()**. The function **ProMenubuttonPreactionSet()** inserts your function before an existing Pro/ENGINEER command instead of assigning it to a new button. The function **ProMenubuttonPostactionSet()** inserts your function after an existing Pro/ENGINEER command.

Because you are changing the definition of the menu in Pro/ENGINEER, you must make sure the menu is loaded into memory first, by calling **ProMenuFileRegister()**.

If the command function you load returns the value 0, the Pro/ENGINEER command for that menu button will be executed immediately. If your function returns any other value, the Pro/ENGINEER command will not be performed.

Example 11: Asking for Confirmation on Quit Window

The following example code shows how to use **ProMenubuttonPreactionSet()** to ask the user to confirm a selection. The example uses **ProMenubuttonPreactionSet()** to protect **Quit Window** from accidental selection.

```
int user_initialize()
{
    .
    .
    int menuId;
    int ProUserQuitWindowConfirm (ProAppData data, int num);
    .
    .
    ProMenuFileRegister ("main", "main.mnu", &menuId);
    ProMenubuttonPreactionSet ("main", "Quit Window",
        ProUserQuitWindowConfirm, NULL, 0);
    .
    .
    return (0);
}

ProUserQuitWindowConfirm (ProAppData data, int num)
{
```

```

wchar_t  w_answer[4];
char      answer[4];
wchar_t  msgfil[20];

ProStringToWstring (msgfil, "msg_ug2.txt");
ProMessageDisplay (msgfil,
    "USER Do you really want to Quit Window? [Y] :");
/*-----*\
    If the user just hit return, go ahead and quit.
\*-----*/
    if (ProMessageStringRead (4, w_answer))
        return (0);
/*-----*\
    If the answer starts with 'n' return TRUE to prevent
    the Quit Window.
\*-----*/
    ProWstringToString (answer, w_answer);
    return (tolower (answer[0]) == 'n');
}

```

Submenus

Function Introduced:

- **ProMenuCreate()**

All the menus described so far have been main menus. The other type of menu is called a submenu. A submenu differs from a main menu in the following ways:

- A submenu is active at the same time as the menu above it. Selecting from the menu above does not close the submenu.
- A submenu does not display its title.

In effect, a submenu acts as an extension to the menu above it. This enables you to display two active menus at the same time, such as if you want the user to choose two options from two exclusive groups of values.

Making a Menu a Submenu

To make a Main Menu a Submenu:

1. Display the menu above the submenu, using **ProMenuCreate()**.
2. Display the submenu, using **ProMenuCreate()**, but make the first argument PROMENUTYPE_SUB instead of PROMENUTYPE_MAIN.

3. Call **ProMenuProcess()** for the submenu only. Because it is a submenu, the menu above it will become active at the same time.
4. Close both menus, using either **ProMenuDelete()** or **ProMenuDeleteWithStatus()**.

Manipulating Menus

Function Introduced:

- **ProMenubuttonLocationSet()**

The function **ProMenubuttonLocationSet()** provides the ability to move a Pro/ENGINEER menu button to a different location on its menu, or to add new menu buttons to a Pro/ENGINEER menu somewhere other than at the bottom of the menu.

Before you call **ProMenubuttonLocationSet()**, you must make sure the menu you are modifying has been fully loaded into memory. Make sure **ProMenuFileRegister()** has been called, and, where appropriate, **ProMenuAuxfileRegister()**.

The first two arguments of the **ProMenubuttonLocationSet()** function identify the menu and the button, as in **ProMenubuttonActionSet()**.

The final argument is a switch that specifies where to move the button. The possible values are as follows:

- 0—The button becomes the first in the menu.
- 1—The button is inserted after the current first button.
- 2—The button is inserted after the current second button.
- -1—The button becomes the last button on the menu.

Data Menus

Functions Introduced:

- **ProMenuModeSet()**
- **ProMenuDatamodeSet()**

Menus can operate in two modes:

- **PROMENUMODE_OPERATIONAL**—The default mode. This mode is used in all the previous examples. On an operational menu, only one button is ever set (that is, displayed with a red background) while that command is in progress.

- **PROMENUMODE_DATA**—Each button remains set until you select it again. This is useful when the buttons do not represent commands, but, for example, a set of independently selectable options.

The function **ProMenuModeSet()** sets the menu mode. For a **PROMENUMODE_DATA** menu, you can choose to indicate the set buttons with a check mark instead of the usual red background by using the function **ProMenuDatamodeSet()**.

Calling ProMenuModeSet() and ProMenuDatamodeSet()

The function **ProMenuModeSet()** has two arguments:

- The menu title
- The menu mode (either **PROMENU_MODE_OPERATIONAL** or **PROMENUMODE_DATA**)

The function **ProMenuDatamodeSet()** has two arguments:

- The menu title.
- The set indicator, which indicates which buttons are set. This argument can have either of the following values:
 - **TRUE**—Use a check mark.
 - **FALSE**—Use a red background. This is the default value.

Both of these functions must be called after the menu has been loaded into memory (using **ProMenuFileRegister()**), and before the menu has been displayed (using **ProMenuCreate()**).

If you want to create a menu whose buttons are dependent on run-time data, use the function **ProMenuStringsSelect()**, described later in this chapter.

Setting Menu Buttons

Functions Introduced:

- **ProMenubuttonHighlight()**
- **ProMenubuttonUnhighlight()**

Sometimes it is useful to be able to set and unset menu buttons from the Pro/TOOLKIT application. For example, if you are using data menus, you can set the appropriate buttons when the menu is displayed to show the current options.

Calling ProMenubuttonHighlight() and ProMenubuttonUnhighlight()

Both **ProMenubuttonHighlight()** and **ProMenubuttonUnhighlight()** take two arguments—the menu title and button name. Both functions must be called after the menu has been displayed (using **ProMenuCreate()**), but before making the menu interactive (using **ProMenuProcess()**). Contrast these rules to the rules for using **ProMenuModeSet()** and **ProMenuDatamodeSet()**.

Controlling Accessibility of Menu Buttons

Functions Introduced:

- **ProMenubuttonActivate()**
- **ProMenubuttonDeactivate()**

A menu button that is inaccessible is one that, though currently displayed on a menu, is gray and has no effect when it is selected. Pro/ENGINEER uses this facility for options that are temporarily unavailable for some reason. For example, you cannot create a hole until you have created the first protrusion.

You can control the accessibility of your own menu buttons from Pro/TOOLKIT using **ProMenubuttonActivate()** and **ProMenubuttonDeactivate()**. Each function takes two arguments: the menu title and button name. These functions must be called when the menu is displayed (after calling **ProMenuCreate()**).

Pushing and Popping Menus

Functions Introduced:

- **ProMenuVisibilityGet()**
- **ProMenuPush()**
- **ProMenuPop()**

Sometimes Pro/ENGINEER temporarily hides certain menus, even though they are still in context, to make room for lower-level menus. An example of this is when you select **Make Datum** during feature creation. This process is called pushing menus, because they are put on a stack from which they can be popped to make them reappear.

The function **ProMenuVisibilityGet()** tells you whether the specified menu is currently displayed. It takes one input argument—the menu title.

The function **ProMenuPush()** pushes the current lowest menu. It takes no arguments.

The function **ProMenuPop()** pops the menu from the top of the stack. It takes no arguments.

Run-time Menus

Functions Introduced:

- **ProMenuStringsSelect()**
- **ProMenuFromStringsRegister()**

The **ProMenuStringsSelect()** function enables you to set up a menu at run time. You do not need to supply a menu file because the buttons are defined when you display the menu. You cannot attach command functions to the button; a run-time menu simply returns a list of the buttons selected.

A run-time menu is displayed together with a submenu that contains the following buttons:

- **Done Select**
- **Quit Select**
- **List**

The default option, **List**, causes the string menu itself to be displayed.

You can set the maximum number of items you want to be selectable. The function returns when the user has selected the maximum number of items you specified, or has selected **Done** or **Quit**. Pro/ENGINEER uses this type of menu to select a disk file to be retrieved after the user selects **Search/Retr**.

The maximum size of the string you assign to a button is `PRO_NAME_SIZE - 1`. `PRO_NAME_SIZE` is defined in file *ProSizeConst.h*.

The function **ProMenuFromStringsRegister()** creates menus at run time and attaches actions to the menu buttons. The function takes as arguments all the information required to create auxiliary (*.aux) and user-defined (*.mnu) menu files. The first argument is the default menu name. The next argument enables you to specify an alternate name for the menu if, for instance, your application

supports a foreign language. The list of button labels is passed to the function as an array of wide character strings. As with the menu name, you can provide alternate button labels for foreign language support. You can also provide one-line Help for each button.

After you have registered the menu with a call to the function **ProMenuFromStringsRegister()**, you can attach actions to the buttons by calling the function **ProMenuButtonActionSet()** for each button. You must also define an exit action for your run-time menu. To do this, call **ProMenuButtonActionSet()** and supply the name of the menu instead of a button name. Finally, create the menu by calling **ProMenuProcess()**, and then **ProMenuCreate()**.

Customizing the Pro/ENGINEER Navigation Area

The Pro/ENGINEER navigation area includes the **Model** and **Layer Tree** pane, **Folder browser** pane, and **Favorites** pane. The functions described in this section enable Pro/TOOLKIT applications to add custom panes to the Pro/ENGINEER navigation area. The custom panes can contain custom dialog box components or WEB pages.

Adding Custom Web Pages

To add custom web pages to the Navigation area the Pro/TOOLKIT application must:

1. Add a new pane to the navigation area.
2. Set an icon for this pane.
3. Set the URL of the location that will be displayed in the pane.

Functions Introduced:

- **ProNavigatorpaneBrowserAdd()**
- **ProNavigatorpaneBrowsericonSet()**
- **ProNavigatorpaneBrowserURLSet()**

The function **ProNavigatorpaneBrowserAdd()** adds a new pane that can display a web page to the navigation area. The input arguments are:

- *pane_name*—Specify a unique name for the pane. Use this name in subsequent calls to **ProNavigatorpaneBrowsericonSet()** and **ProNavigatorpaneBrowserURLSet()**.
- *icon_file_name*—Specify the name of the icon file, including the extension. A valid format for the icon file is .GIF, .JPG, or .PNG. The new pane is displayed with the icon image. If you specify the value as “NULL”, the default Pro/ENGINEER icon is used.
- *url*—Specify the URL of the location to be accessed from the pane.

Use the function **ProNavigatorpaneBrowsericonSet()** to set or change the icon of a specified browser pane in the navigation area.

Use the **ProNavigatorpaneBrowserURLSet()** to change the URL of the page displayed in the browser pane in the navigation area.

Adding Custom Dialog Box Components

To add a new pane to the navigation area based on Pro/TOOLKIT dialog box components:

1. Add a new pane to the navigation area.
2. Assign an icon for the pane.
3. Add the Pro/TOOLKIT dialog box components using one of the following methods:
 - Assign resource files that describe the overall structure of the new navigation pane. To customize the components after placement in the navigation pane, get the window device name and component name using the Pro/TOOLKIT functions described in this section and set the necessary values using the Pro/TOOLKIT functions described in the chapter “User Interface: Dialogs”.
 - To build the layout of the dialog box programmatically, obtain the window device name and layout name. Add each component to the layout as desired using Pro/TOOLKIT user interface functions described in the chapter “User Interface: Dialogs”.

Functions Introduced:

- **ProNavigatorpanePHolderAdd()**
- **ProNavigatorpanePHolderDevicenameGet()**
- **ProNavigatorpanePHolderLayoutGet()**
- **ProNavigatorpanePHolderComponentnameGet()**

The function **ProNavigatorpanePHolderAdd()** adds a layout that will be displayed in the new pane in the navigation area. The input arguments are:

- *pane_name*—Specify a unique name for the pane.
- *resource_name*—Specify the name of the resource file to use (whose top component must be a layout, not a dialog box). The contents of the layout from the specified resource file will be inserted into the custom pane.
- *icon_file_name*—Specify the name of the icon file, including the extension. A valid format for the icon file is .GIF, .JPG, or .PNG. The new pane is displayed with the icon image. If you specify the value as “NULL”, the default Pro/ENGINEER icon is used.

The function **ProNavigatorpanePHolderLayoutGet()** returns the layout name for the specified pane in the navigation area. You can create and place the user interface components within this layout.

Components added to the custom pane actually belong to the Pro/ENGINEER main window dialog. Pro/ENGINEER automatically modifies the names of components loaded from resource files to ensure that no name collisions occur when the components are added. The functions **ProNavigatorpanePHolderDevicenameGet()** and **ProNavigatorpanePHolderComponentnameGet()** allow you to locate the names of components that you need to access.

Use the function **ProNavigatorpanePHolderDevicenameGet()** to obtain name of the Pro/ENGINEER window owning the new pane in the navigation area.

Use the function **ProNavigatorpanePHolderComponentnameGet()** to obtain the complete name of the component in the navigation pane, if loaded from a layout.

Use the device name and the component name to add or update the placement of the components in the layout with the help of the ProUI* functions. Refer to the chapter User Interface: Dialogs for more information on the user interface functions.

Example 13: Customizing the Pro/ENGINEER Navigation Pane

The following sample code shows you how to customize the Pro/ENGINEER navigation pane.

```

/*****\
FILE      : UgNavigatorPane.c
PURPOSE   : Pro/TOOLKIT User Guide Example - User Defined Navigator Panes
\*****/

/*----- Pro/Toolkit includes -----*/
#include <ProToolkit.h>
#include <ProMdl.h>
#include <ProUtil.h>
#include <ProMessage.h>
#include <ProNavigator.h>
#include <ProArray.h>
/*----- Application includes -----*/
#include <TestError.h>
#include <UtilFiles.h>
#include <UtilString.h>
#include <PTApplsUnicodeUtils.h>

#define MSGFIL L##"msg_ugmenu.txt"

ProError UgAddNewPane();
ProError UgChangeNavPaneUrl();
ProError UgChangeNavPaneIcon();
ProError UgNavPaneInfoGet();
/*=====*\
FUNCTION   : UgAddNewPane()
PURPOSE    : Adds new Navigator Pane
\*=====*/
ProError status;
ProError UgAddNewPane()
{
    wchar_t w_pane_name[PRO_LINE_SIZE];
    ProCharName pane_name;
    wchar_t url1[PRO_LINE_SIZE];

    status = ProMessageDisplay (MSGFIL, "Please enter name for
                                   the new pane");
    status = ProMessageStringRead(PRO_LINE_SIZE, w_pane_name);

    if(status != PRO_TK_NO_ERROR)

```

```

        return (status);

ProWstringToString (pane_name, w_pane_name);

ProMessageDisplay (MSGFIL, "Please enter URL for the pane");
status = ProMessageStringRead(PRO_LINE_SIZE, url1);

if(status != PRO_TK_NO_ERROR)
    return (status);

status = ProNavigatorpaneBrowserAdd(pane_name, NULL, url1);
ERROR_CHECK("ProNavigatorpaneBrowserAdd",
            "ProNavigatorpaneBrowserAdd",status);
if(status == PRO_TK_NO_ERROR)
    ProMessageDisplay (MSGFIL, "Added new pane");
else
    ProMessageDisplay (MSGFIL, "Error in new pane creation");

return (status);
}

/*=====*\
FUNCTION   : UgAddNewDialogPane()
PURPOSE    : Creates the Navigator Pane with Dialog per resource file
\*=====*/

ProError UgAddNewDialogPane()
{
    wchar_t w_pane_name[PRO_LINE_SIZE];
    ProCharName pane_name;
    ProCharName resource_name;

    status = ProMessageDisplay (MSGFIL, "Please enter name for
                                the new pane");
    status = ProMessageStringRead(PRO_LINE_SIZE, w_pane_name);

    if(status != PRO_TK_NO_ERROR)
        return (status);

    ProWstringToString (pane_name, w_pane_name);

    strcpy(resource_name, "ug_navigatorpane.res");

    status = ProNavigatorpanePHolderAdd (pane_name, resource_name, NULL);
    ERROR_CHECK("UgAddNewDialogPane", "ProNavigatorpanePHolderAdd",
                status);

    if(status == PRO_TK_NO_ERROR)
        ProMessageDisplay (MSGFIL, "Added new pane");
    else
        ProMessageDisplay (MSGFIL, "Error in new pane creation");
}

```



```

    return (status);
}

/*=====*\
FUNCTION   : UgChangeNavPaneUrl()
PURPOSE    : Changes Navigator Panes URL
\*=====*/

ProError UgChangeNavPaneUrl()
{
    wchar_t w_pane_name[PRO_LINE_SIZE];
    ProCharName pane_name;
    wchar_t url1[PRO_LINE_SIZE];

    status = ProMessageDisplay (MSGFIL, "Please enter the pane name
                                     for changing icon");
    status = ProMessageStringRead(PRO_LINE_SIZE, w_pane_name);

    if(status != PRO_TK_NO_ERROR)
        return (status);

    ProWstringToString (pane_name, w_pane_name);

    status = ProMessageDisplay (MSGFIL, "Please enter the new URL");
    status = ProMessageStringRead(PRO_LINE_SIZE, url1);

    if(status != PRO_TK_NO_ERROR)
        return (status);

    status = ProNavigatorpaneBrowserURLSet(pane_name, url1);
    ERROR_CHECK("ProNavigatorpaneBrowserAdd",
               "ProNavigatorpaneBrowserURLSet",status);

    if (status != PRO_TK_NO_ERROR)
        ProMessageDisplay (MSGFIL, "Pane with entered name
                                   does not exist");

    else
        ProMessageDisplay (MSGFIL, "Pane URL Updated");

    return(status);
}

/*=====*\
FUNCTION   : UgChangeNavPaneIcon()
PURPOSE    : Changes Navigator Panes Icon
\*=====*/

ProError UgChangeNavPaneIcon()
{

```

```

wchar_t w_pane_name[PRO_LINE_SIZE];
ProCharName pane_name;
wchar_t w_icon_file[PRO_LINE_SIZE];
ProCharName icon_file_name;
ProLine buff;
ProPath *path_arr, sel_path, def_path;
ProName *path_lab_arr;

status = ProMessageDisplay (MSGFIL, "Please enter the pane name
                                for changing icon");
status = ProMessageStringRead(PRO_LINE_SIZE, w_pane_name);

if(status != PRO_TK_NO_ERROR)
    return (status);

ProWstringToString (pane_name, w_pane_name);
ProMessageDisplay(MSGFIL, "Choose new Icon file");

/*-----*\
    Prompt the user to select the file.
\*-----*/
ProStringToWstring(buff, "*.jpg,*.gif");
ProStringToWstring(def_path, ".");

ProArrayAlloc(0, sizeof(ProPath), 1, (ProArray*)&path_arr);
ProArrayAlloc(0, sizeof(ProPath), 1, (ProArray*)&path_lab_arr);
status = ProFileOpen(NULL, buff, path_arr, path_lab_arr, def_path,
                    NULL, sel_path);

if (status != PRO_TK_NO_ERROR)
    ProMessageDisplay (MSGFIL, "No file selected");

else
{
    ProWstringToString (icon_file_name, sel_path);
    status = ProNavigatorpaneBrowsericonSet(pane_name, icon_file_name);
    ERROR_CHECK("UgChangeNavPaneIcon", "ProNavigatorpaneBrowsericonSet",
                status);
    if (status != PRO_TK_NO_ERROR)
        ProMessageDisplay (MSGFIL, "Pane with entered name does not
                                exist");
    else
        ProMessageDisplay (MSGFIL, "Pane icon Updated");
}

return (status);
}

/*=====*\
FUNCTION   : UgNavPaneInfoGet()
PURPOSE    : Gives Navigator Pane information

```

```
\*=====*/
```

```
ProError UgNavPaneInfoGet()
{
    wchar_t w_pane_name[PRO_LINE_SIZE];
    ProCharName pane_name;
    ProCharName icon_file_name;
    wchar_t url1[PRO_LINE_SIZE];
    char* comp_name;
    char* layout_name;
    char* layout_name2;
    int win_id;
    char * device_name;
    char *nominal_name = "LocalName";
    FILE *fp;
    char name[PRO_NAME_SIZE];
    wchar_t wname[PRO_NAME_SIZE];

    fp = PTApplsUnicodeFopen("info.txt", "w");
    status = ProMessageDisplay (MSGFIL, "Please enter the pane name");
    status = ProMessageStringRead(PRO_LINE_SIZE, w_pane_name);

    if(status != PRO_TK_NO_ERROR)
        return (status);

    ProWstringToString (pane_name, w_pane_name);

    status = ProWindowCurrentGet (&win_id);
    ERROR_CHECK("UgNavPaneInfoGet", "ProWindowCurrentGet", status);
    if( status == PRO_TK_NO_ERROR)
    {
        status = ProNavigatorpanePHolderDevicenameGet (win_id, &device_name);
        ERROR_CHECK("UgNavPaneInfoGet", "ProNavigatorpanePHolderDevicenameGet",
                    status);

        if (status != PRO_TK_NO_ERROR)
            return (status);
        else
        {
            ProTKFprintf(fp, "Device name is: %s \n", device_name);
            ProStringFree(device_name);
        }
    }
    else
        return (status);

    status = ProNavigatorpanePHolderLayoutGet (pane_name, &layout_name);
    ERROR_CHECK("UgNavPaneInfoGet", "ProNavigatorpanePHolderLayoutGet",
                status);
}
```

```

if (status != PRO_TK_NO_ERROR)
{
    ProMessageDisplay (MSGFIL, "Pane with entered name does not
                           exist");
    return (status);
}
else
{
    ProTKFprintf(fp, "layout name is: %s \n", layout_name);
    ProStringFree(layout_name);
}

status = ProNavigatorpanePHolderComponentnameGet (pane_name,
                                                    nominal_name,&comp_name);
ERROR_CHECK("UgNavPaneInfoGet",
            "ProNavigatorpanePHolderComponentnameGet",status);
if (status != PRO_TK_NO_ERROR)
{
    ProMessageDisplay (MSGFIL, "Pane with entered name does not
                           exist");
    return (status);
}
else
{
    ProTKFprintf(fp, "Component name is: %s \n", comp_name);
    ProStringFree(comp_name);
}

fclose(fp);
ProStringToWstring(wname, "info.txt");
ProInfoWindowDisplay(wname, NULL, NULL);

return (status);
}

```

Registering Notifications to Add and Destroy Content to a New Pane

The navigation panes are available in every window within a Pro/ENGINEER session. Notifications are provided to the Pro/TOOLKIT application when a Pro/ENGINEER window populated with a model so that the application can add the necessary contents to the new pane upon this event. Similarly, notifications are provided before a model is removed from a Pro/ENGINEER window so that the application can cleanup resources related to added panes. For more information on using notifications see the chapter Event-driven Programming: Notifications.

Functions Introduced:

- **ProNotificationSet()**
- **ProWindowOccupyPostAction()**
- **ProWindowVacatePreAction()**

Specify the argument *type* of the function **ProNotificationSet()** to **PRO_WINDOW_OCCUPY_POST**, to call the callback function whose signature matches **ProWindowOccupyPostAction()**. This function is called when a new Pro/ENGINEER window is created, or when the base window is populated and is used by the application to add the necessary content to the new pane in the navigation area.

Specify the argument *type* of the function **ProNotificationSet()** to **PRO_WINDOW_VACATE_PRE**, to call the callback function whose signature matches **ProWindowVacatePreAction()**. This function is called when a Pro/ENGINEER window is closed, or when the base window gets cleared. Use this function to free memory allocated by the Pro/TOOLKIT application to add content in the navigation pane.

Entering Pro/ENGINEER Commands

Functions Introduced:

- **ProMacroLoad()**
- **ProMacroExecute()**
- **ProMenuCommandPush()**

The function **ProMacroLoad()** loads a macro string or a mapkey onto a stack of macros that are executed after control returns to Pro/ENGINEER. A Pro/TOOLKIT macro string is a string. The macro string is equivalent to a mapkey without the key sequence and the mapkey name.

Note: **ProMacroLoad()** fails if a macro string contains a backslash or if a command splits over two lines with or without a backslash. A macro string can contain multiple commands separated by semicolons. However, each command should entirely appear in a single line.

Create a mapkey in the Pro/ENGINEER user interface from the **Tools>Mapkeys** dialog box. Copy the value of the generated mapkey option from the **Tools>Options** dialog box. An example of a created mapkey is as follows:

```
$F2 @MAPKEY_LABELtest;  
~ Activate `main_dlg_cur` `ProCmdModelNew.file`;  
~ Activate `new` `OK`;
```

The key sequence is \$F2. The mapkey name is @MAPKEY_LABELtest. The remainder of the string after the first semicolon is the macro string. In this case, it is as follows:

```
~ Activate `main_dlg_cur` `ProCmdModelNew.file`;  
~ Activate `new` `OK`;
```

You can either pass the mapkey directly or the generated macro string to **ProMacroLoad()**. When passing the mapkey, pass it as '%mapkey_name'.

Note: Creating or editing the macro string manually is not supported, as mapkeys are not a supported scripting language. The syntax is not defined for users and may not remain constant across different datecodes of Pro/ENGINEER.

Execution Rules

Consider the following rules about the execution of macros:

- In asynchronous mode, macros are executed as soon as they are loaded with **ProMacroLoad()**.
- In synchronous mode, the mapkey or the macro strings are pushed onto a stack and are popped off and executed only when control returns to Pro/ENGINEER from the Pro/TOOLKIT program. Due to the last in, first out nature of the stack, macros that cannot be passed entirely in one **ProMacroLoad()** call should have the strings loaded in reverse order of desired execution.
- To execute a macro from within Pro/TOOLKIT, call the function **ProMacroExecute()**. The function runs the Pro/ENGINEER macro and returns the control to the Pro/TOOLKIT application. It executes the macros previously loaded using the function **ProMacroLoad()**. The function works only in the synchronous mode.
- Do not call the function **ProMacroExecute()** during the following operations:
 - Activating dialog boxes or setting the current model
 - Erasing the current model
 - Replaying a trail file

- Clicking the **OK** button on the dialog box to complete the command operation. In this case, the dialog box may be displayed momentarily without completing the command operation.

Note:

- You can execute only the dialog boxes with built-in exit confirmation as macros, by canceling the exit action.
- It is possible that a macro may not be executed because a command specified in the macro is currently inaccessible in the menus. The functional success of **ProMacroExecute()** depends on the priority of the executed command against the current context.
- If some of the commands ask for an input to be entered from the keyboard (such as a part name), the macro continues execution after you type the input and press ENTER. However, if you must select something with the mouse (such as selecting a sketching plane), the macro is interrupted and ignores the remaining commands in the string.

This allows the application to create object-independent macros for long sequences of repeating choices (as long as the user does not have to select any geometry).

A **ProStringToWstring()** call for defining the macro string must be followed by the following calls:

- **ProMacroLoad(macro wstring)** to load the macro strings or the mapkey
- **ProMacroExecute()** to execute the macro

Some sample macros in various scenarios are given below.

Menu bar and Tool bar Macros

The following single entry and exit type of interactions are supported by **ProMacroExecute()**.

- To switch the wireframe display of model, use the macro:

```
ProStringToWstring ( macro_wstring, "~ Select
`main_dlg_cur` `ProCmdEnvMdlDisp.mdisp` 1
`Wireframe`");
```

- To switch the shaded display of model, use the macro:

```
ProStringToWstring ( macro_wstring, "~ Select
`main_dlg_cur` `ProCmdEnvMdlDisp.mdisp` 1 `Shading`"
);
```

- You can switch the display for datum planes and datum axes using the following macros:

- **Datum Planes**

```
ProStringToWstring ( macro_wstring, "~ Activate
`main_dlg_cur` `ProCmdEnvDtmDisp.ddisp` 0 " );
```

- **Datum Axes**

```
ProStringToWstring ( macro_wstring, "~ Activate
`main_dlg_cur` `ProCmdEnvAxisDisp.ddisp` 0 " );
```

- To repaint a model, use the macro:

```
ProStringToWstring ( macro_wstring[0], "~ Activate
`main_dlg_cur` `ProCmdViewRepaint.view` " );
```

- To get the default model orientation, use the macro:

```
ProStringToWstring ( macro_wstring, "~ Activate
`main_dlg_cur` `ProCmdViewNamePick.view` 1; ~ Select
`nameviewlist` `nv_list` 1 `Default`");
```

- To get the model information, use the macro:

```
ProStringToWstring ( macro_wstring, "~ Select
`main_dlg_cur` `MenuBar1` 1 `Info` ~ Close
`main_dlg_cur` `MenuBar1` ~ Activate `main_dlg_cur`
`Info.psh_info_model` " );
```

Macros For Feature Creation Tool Bar

The following macros are used while creating the following features.

- To create a hole feature, use the macro:

```
ProStringToWstring ( macro_wstring, "~ Activate
`main_dlg_cur` `ProCmdHole.d_feat` " );
```

- To extrude a feature, use the macro:

```
ProStringToWstring ( macro_wstring, "~ Activate
`main_dlg_cur` `ProCmdFtExtrude.a_feat`");
```

- To create a datum plane, use the macro:

```
ProStringToWstring ( macro_wstring[0], "~ Activate
`main_dlg_cur` `ProCmdDatumPlane.datum`");
```


Pro/ENGINEER Wildfire Navigator Macros

The following macros are provided for Pro/ENGINEER Wildfire navigator:

- For folder navigator:

```
ProStringToWstring ( macro_wstring, "~ Select
`main_dlg_cur` `PHTLeft.ProExplorerTab` 1
`PHTLeft.Folders`");
```

- For Favorites navigator:

```
ProStringToWstring ( macro_wstring, "~ Select
`main_dlg_cur` `PHTLeft.ProExplorerTab` 1
`PHTLeft.FavLay`");
```

- For the Model Tree:

```
ProStringToWstring ( macro_wstring, "~ Select
`main_dlg_cur` `PHTLeft.ProExplorerTab` 1
`PHTLeft.MdlTreeLay`");
```

The function **ProMenuCommandPush()** places the name of a specific menu button in the command input buffer for Pro/ENGINEER. This command is executed after control returns to Pro/ENGINEER from the Pro/TOOLKIT application, as if you have selected that menu button. This menu button must be from the menu that is currently displayed in the Pro/ENGINEER user interface.

Specifying Keyboard Input

You can specify keyboard input within the command string. As previously specified, a macro must be preceded by a pound sign (#) and terminated by a semicolon. If the field after the semicolon does not start with a pound sign, the data up to the next semicolon is used as input at the next keyboard prompt. If the command currently being executed does not request keyboard input, the system ignores this keyboard data. Note that keyboard data is case-sensitive and spaces are not ignored. A carriage return is indicated when no data appears between the semicolons.

Note: Note that the correctness of the sequence is the responsibility of the user. PTC does not guarantee that a sequence will be valid from one version of Pro/ENGINEER to another.

16

User Interface: Dialogs

This chapter describes the User Interface (UI) components available in Pro/TOOLKIT for Pro/ENGINEER Wildfire 3.0 onwards. The following sections introduce each of the dialog component types, operations and callbacks available for each component, and the methods and techniques that can be used to instantiate and show customized user interface dialogs.

Topic	Page
Introduction	16 - 3
UI Components	16 - 4
Cascade Button	16 - 17
Checkbutton	16 - 18
Drawing Area	16 - 20
Input Panel	16 - 28
Label	16 - 31
Layout	16 - 32
List	16 - 35
Menubar	16 - 42
Menupane	16 - 43
Optionmenu	16 - 44
Progressbar	16 - 47
Pushbutton	16 - 48
Radiogroup	16 - 52
Separator	16 - 54

Slider	16 - 55
Spinbox	16 - 61
Tab	16 - 63
Table	16 - 66
Textarea	16 - 81
Thumbwheel	16 - 83
Tree	16 - 85
Master Table of Resource File Attributes	16 - 97
Using Resource Files	16 - 110

Introduction

This chapter includes documentation for each UI component. The documentation is divided into the following sections:

- **Attributes**—Defines the names and functions that affect attributes on the UI component. Each component supports its own set of unique attributes; however, some attributes are supported by more than one component. Because of the fact that attributes typically work on more than one component, detailed documentation for the attributes is included in a master table at the end of this chapter.
- **Operations**—Defines the component-specific functions that make more detailed modifications to the components.
- **Actions**—Defines the functions that register action callbacks on a component.

About Pro/TOOLKIT Support for User Interface

Pro/TOOLKIT allows applications to create dialogs and dashboards with the same look and feel as those in Pro/ENGINEER.

Pro/TOOLKIT users can accomplish this task by following the following steps:

- Establish the main UI container object. This is either a dialog or a dashboard. Optionally, this can be read from a resource file to prepopulate the container with specific components.
- Add components to the container (if they do not already exist)
- Set attributes on components in the container
- Execute operations on components in the container. Operations also modify the component, but typically make more detailed or sophisticated changes than setting an individual attribute
- Establish action function callbacks on components in the container. Action functions are called when the user interacts with the component in some way. They allow the application to react to user events.
- Show the dialog or dashboard.
- Based on user actions, eventually you will need to close the container.
- "Destroy" the container to free up the resources it uses.

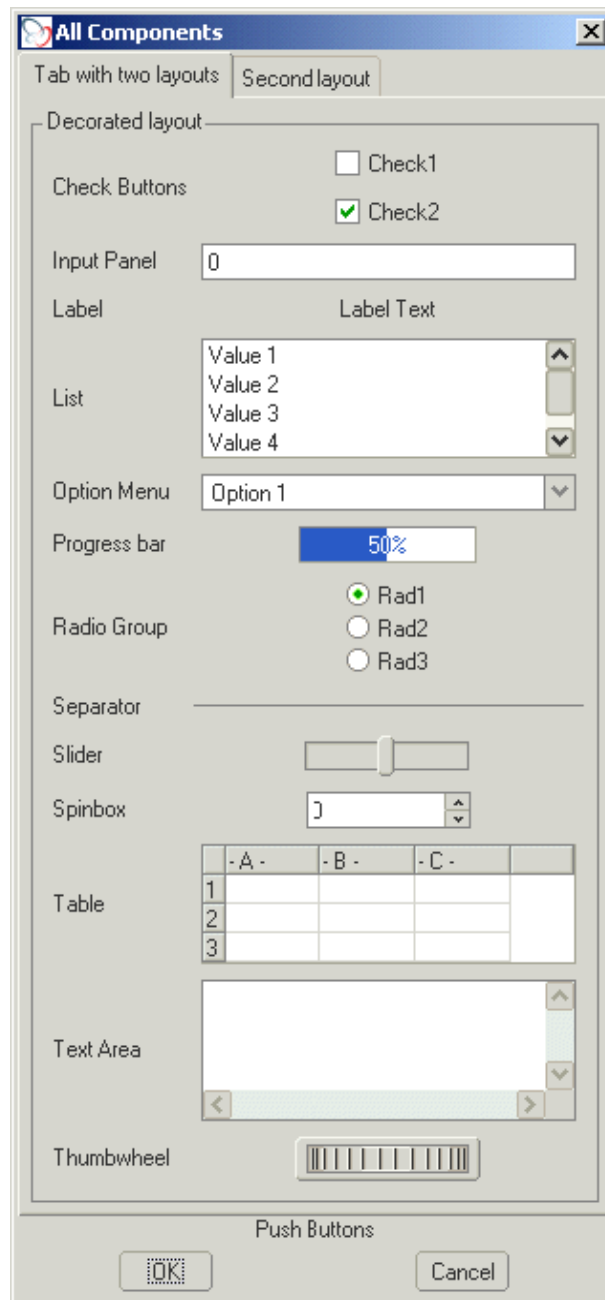
Note: The functions described in this section do not support using Pro/TOOLKIT to modify standard Pro/ENGINEER dialogs.

The UI function library is integral to Pro/ENGINEER, not the Pro/TOOLKIT libraries; thus, the library can only be used while a Pro/ENGINEER session is active. The library cannot be used to display user interfaces when the application is not connected to Pro/ENGINEER running interactively.

UI Components

Most of the UI components that Pro/TOOLKIT can access are shown in the following figure.

Figure 16-1: All Components Dialog



The behavior and uses of the different component types is introduced briefly below, and described in more detail in later sections. The component types are:

- **Tab**—part of a dialog that can contain several groups of components, formatted such that only one group is visible at a time. A Tab component must always contain a set of Layout components; each layout contains the components that must be displayed at one time. The Figure - ‘All Components Dialog’ shows a decorated Tab which displays a handle on each layout to allow the user to select which layout is visible.
- **Layout**—an area of a dialog which can contain any number of other dialog components. A Layout can be used to better control the relative position of components in a dialog, by allowing the grids in different parts of the dialog to adopt unaligned rows or columns. A layout can also be used inside a Tab component.
- **Check Button**—a button which toggles between a TRUE and FALSE state each time the user selects it.
- **Drawing Area**—a component which allows points, lines, shapes, images and text (including symbols) to be drawn in a variety of colors.
- **Input Panel**—a box containing a single line of text. The Input Panel may be set to expect text in different formats, for example a real number or an integer. The Input Panel may also be set to be read-only, when it is used by the application to show information to the user.
- **Label**—a text string used to label the other components.
- **List**—a box containing a list of text strings, which can be selected by the user. Users can set the List to allow selection of only one item at a time, or more than one.
- **Option Menu**—a single-line box which allows selection of a single text string from a list of options. The selection is done using a pull-down menu, which appears when a button next to the text box is selected.
- **Progress Bar**—a component which shows the progress of a time-consuming action.
- **Push Button**—a button which performs some action when it is selected. It does not contain any remembered state. Push Buttons appear on almost every dialog as OK and Cancel buttons.

- **Radio Group**—a set of buttons which individually act like check buttons, but which are connected to each other such that only one can be set to TRUE at any time. Selecting one button sets that button and unsets all others in the group.
- **Separator**—a separator is for cosmetic purposes only, and helps to visually divide components into logical groups.
- **Slider**—a device which allows the user to set a value in a predefined range by moving a handle with the mouse. Use sliders in situations where an exact value may not be needed. A slider should usually be tied programmatically with a read-only input panel to show the current value.
- **Spin-Box**—a box containing a single numerical value that can be directly edited. The spin box also has up- and down-arrow buttons for increasing or decreasing the value in steps. A single click increments or decrements by a single step. Holding a button down makes the value change in repeated steps, first small steps and then large steps. The step sizes can be set for each spin box.
- **Table**—a set of tabulated rows and columns containing text and other components.
- **Text Area**—a box containing unformatted text containing any number of lines. It may be set to be read-only and used by the application to output information to the user.
- **Thumbwheel**—a thumbwheel is similar to slider but provides finer control over a wider range of values. Unlike the slider, it does not provide a visual indication of the current value.
- **Tree**—a tree contains nodes which are structured and displayed in a hierarchical formation.

Naming Convention for UI Components

The valid characters for naming UI components are:

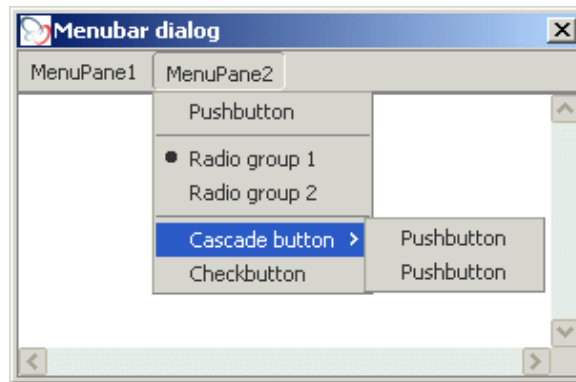
- A to Z (uppercase)
- a to z (lowercase)
- 0 to 9
- Underscore(_)
- Hyphen(-)

Using any other characters in UI component names may result in an error.

Menubars and Menubar Components

A dialog can also contain its own menubar. These menubars support cascading menus. The figure below shows an example of a simple dialog containing a menubar and a text area.

Figure 16-2: Menubar Dialog



See Example 15: Resource File for Dialog with Menubar for resource file code for this example.

The following components are used to define menu bars and their dependent menus:

- **Menubar**—The menubar itself is just a container for the menu panes. A dialog can contain only one menubar, and it must contain at least one other component at the top level.
- **MenuPane**—A menu pane describes a button on a menubar and also acts as a container for the components on the pull-down menu that appears when the user selects the menu pane button.
- **Cascade button**—A button on a pull-down menu that contains its own menupane. Selecting the cascade button pulls out the menu described by the menupane.

The following components described in the previous section can also be added to menu panes, but in this case their appearance is automatically modified to suit the style of pull-down menus:

- **Check Button**—This looks like a regular menu button, but in fact toggles its state. When `TRUE`, it shows a check mark next to the label.
- **Push Button**—When added to a menu pane a pushbutton represents a command that causes some action.

- **Radio Group**—A radio group on a menu pane behaves exactly as it would in the body of a dialog, although the appearance is rather different, as shown in the picture above.
- **Separator**—A separator can be used to group buttons on a menu pane.

Dialog Attributes

Attribute Name	Get Function	Set Function(s)
.AttachBottom	ProUIDialogIsAttachedBottom()	ProUIDialogAttachBottom() ProUIDialogUnattachBottom()
.AttachTop	ProUIDialogIsAttachedTop()	ProUIDialogAttachTop() ProUIDialogUnattachTop()
.AttachRight	ProUIDialogIsAttachedRight()	ProUIDialogAttachRight() ProUIDialogUnattachRight()
.AttachLeft	ProUIDialogIsAttachedLeft()	ProUIDialogAttachLeft() ProUIDialogUnattachLeft()
.BottomOffset	ProUIDialogBottomoffsetGet()	ProUIDialogBottomoffsetSet()
.Bitmap	ProUIDialogBitmapGet()	ProUIDialogBitmapSet()
.ChildNames	ProUIDialogChildnamesGet()	Not Applicable
.ClassName	ProUIComponentClassnameGet()	Not Applicable
.DefaultButton	ProUIDialogDefaultbuttonGet()	ProUIDialogDefaultbuttonSet()
.DialogStyle	ProUIDialogDialogstyleGet()	ProUIDialogDialogstyleSet()
.Focus	Not Applicable	ProUIDialogFocusSet()
.Height	ProUIDialogHeightGet()	ProUIDialogHeightSet()
.HorzAtPoint	ProUIDialogHorzatpointGet()	ProUIDialogHorzatpointSet()
.HorzDialog	ProUIDialogHorzdialogGet()	ProUIDialogHorzdialogSet()
.HorzMethod	ProUIDialogHorzmethodGet()	ProUIDialogHorzmethodSet()
.HorzPosition	ProUIDialogHorzpointGet()	ProUIDialogHorzpointSet()
.HorzPosOffset	ProUIDialogHorzposoffsetGet()	ProUIDialogHorzposoffsetSet()
.HorzSize	ProUIDialogHorzsizeGet()	ProUIDialogHorzsizeSet()
.Labels	ProUIDialogTitleGet()	ProUIDialogTitleSet()
.LeftOffset	ProUIDialogLeftoffsetGet()	ProUIDialogLeftoffsetSet()
.Mapped	ProUIDialogIsMapped()	ProUIDialogMappedSet() ProUIDialogMappedUnset()
.PopupMenu	ProUIDialogPopupmenuGet()	ProUIDialogPopupmenuSet()

Attribute Name	Get Function	Set Function(s)
.RememberPosition	ProUIDialogRemembersPosition()	ProUIDialogRememberPosition() ProUIDialogForgetPosition()
.RememberSize	ProUIDialogRemembersSize()	ProUIDialogRememberSize() ProUIDialogForgetSize()
.Resizable	ProUIDialogIsResizable()	ProUIDialogEnableResizing() ProUIDialogDisableResizing()
.RightOffset	ProUIDialogRightoffsetGet()	ProUIDialogRightoffsetSet()
.StartLocation	ProUIDialogStartlocationGet()	ProUIDialogStartlocationSet()
.TitleBarImage	Not Applicable	ProUIDialogTitlebarimageSet()
.TopOffset	ProUIDialogTopoffsetGet()	ProUIDialogTopoffsetSet()
.VertAtPoint	ProUIDialogVertatpointGet()	ProUIDialogVertatpointSet()
.VertDialog	ProUIDialogVertdialogGet()	ProUIDialogVertdialogSet()
.VertMethod	ProUIDialogVertmethodGet()	ProUIDialogVertmethodSet()
.VertPoint	ProUIDialogVertpointGet()	ProUIDialogVertpointSet()
.VertPosOffset	ProUIDialogVertposoffsetGet()	ProUIDialogVertposoffsetSet()
.VertSize	ProUIDialogVertsizeGet()	ProUIDialogVertsizeSet()
.Width	ProUIDialogWidthGet()	ProUIDialogWidthSet()

Dialog Operations

Functions Introduced

- **ProUIDialogCreate()**
- **ProUIDialogActivate()**
- **ProUIDialogComponentsCollect()**
- **ProUIDialogMinimumsizeGet()**
- **ProUIDialogSizeGet()**
- **ProUIDialogPositionReset()**
- **ProUIDialogReconfigure()**
- **ProUIDialogScreenpositionGet()**
- **ProUIDialogAboveactivewindowGet()**
- **ProUIDialogAboveactivewindowSet()**
- **ProUIDialogShow()**
- **ProUIDialogHide()**

- **ProUIDialogExit()**
- **ProUIDialogDestroy()**

The function **ProUIDialogCreate()** loads a dialog from a resource file into memory. It can also create an empty dialog by passing a NULL value for the resource file name.

Note:

- The function **ProUIDialogCreate()** requires that the resource file name input matches the case of the actual resource file.
- The dialog name specified in the resource file should match the dialog name input argument.

Use the function **ProUIDialogActivate()** to display the dialog on the screen and makes it active by setting the key input focus on it. This function returns only after the function **ProUIDialogExit()** has been called on the same dialog.

Use the function **ProUIDialogComponentsCollect()** to return the names of components in this dialog. This function can also filter components by their type.

Note: Refer to the section 'UI Components' for the predefined list of component types.

Use the function **ProUIDialogMinimumsizeGet()** to get the minimum size of the dialog in pixels.

Use the function **ProUIDialogSizeGet()** to get the size of the dialog.

Use the function **ProUIDialogPositionReset()** to reset the dialog in its previous screen position.

Use the function **ProUIDialogReconfigure()** to reset the size and position of the dialog.

Use the function **ProUIDialogScreenpositionGet()** get the screen position in pixels of the dialog component.

Use the function **ProUIDialogAboveactivewindowGet()** to checks if the dialog is always going to be above the Pro/ENGINEER dialogs currently active in the Pro/ENGINEER window.

Use the function **ProUIDialogAboveactivewindowSet()** to set focus of the dialog above any dialog in the currently active Pro/ENGINEER window.

Note: Using the **ProUIDialogAboveactivewindowSet()** allows Pro/TOOLKIT applications to always stay in focus in the Pro/ENGINEER window during opening and closing events of Pro/ENGINEER.

Use the **ProUIDialogDestroy()** to remove the dialog instance from memory as it is not automatically removed.

Use the function **ProUIDialogShow()** to show a secondary window when the primary window is being restored.

Use the function **ProUIDialogHide()** to iconify a secondary window when the primary window is being minimized.

Use the function **ProUIDialogExit()** to terminate the activation of the named dialog window. The function causes a return from the call to **ProUIDialogActivate()** that make it active.

Use the function **ProUIDialogDestroy()** to remove the dialog from the memory.

Example 1: Source for Dialog with Text Question, OK and Cancel Buttons

This example shows the source code that uses this dialog.

```
#define OK      1
#define CANCEL 0

int UsrConfirmAction(char *question, ProBoolean *confirm);
/*-----*\
    Example function to show how UsrConfirmAction() is used
\*-----*/
int UsrExample()
{
    ProBoolean confirm;
    UsrConfirmAction("Do you really want to delete the table?", &confirm);
}
/*=====*\
    FUNCTION : UsrOKAction() PURPOSE : Action function for the OK button
\*=====*/
void UsrOKAction(
    char *dialog,
    char *component,
    ProAppData data)
{
    ProUIDialogExit(dialog, OK);
}
/*=====*\
    FUNCTION : UsrCancelAction()
    PURPOSE : Action function for the Cancel button
```

```

\*=====*/
void UsrCancelAction(
    char *dialog,
    char *component,
    ProAppData data)
{
    ProUIDialogExit(dialog, CANCEL);
}
\*=====*\
FUNCTION : UsrConfirmAction()
PURPOSE  : Utility to prompt the user with question, and OK and Cancel
           buttons.
\*=====*/
int UsrConfirmAction(
    char *question,
    ProBoolean *confirm)
{
    ProLine wline;
    int status;

\*-----*\
    Load the dialog from the resource file
\*-----*\
    ProUIDialogCreate("confirm", "confirm");
\*-----*\
    Set the OK and Cancel button actions
\*-----*\
    ProUIPushbuttonActivateActionSet("confirm", "OK", UsrOKAction, NULL);
    ProUIPushbuttonActivateActionSet("confirm", "Cancel", UsrCancelAction
        NULL);
\*-----*\
    Set the Question text in the label
\*-----*\
    ProStringToWstring(wline, question);
    ProUILabelTextSet("confirm", "Question", wline);
\*-----*\
    Display and activate the dialog
\*-----*\
    ProUIDialogActivate("confirm", &status);
\*-----*\
    Set confirm according to which button was used to close the dialog
\*-----*\
    *confirm = (status == OK) ? PRO_B_TRUE : PRO_B_FALSE;
\*-----*\
    Remove the dialog from memory
\*-----*\
    ProUIDialogDestroy("confirm");
    return(1);
}

```

Figure 16-3: Confirm Dialog



Adding and Removing Components

Component Name	Add Function	Remove Function
Checkbutton	ProUIDialogCheckbuttonAdd()	ProUIDialogCheckbuttonRemove()
Drawingarea	ProUIDialogDrawingareaAdd()	ProUIDialogDrawingareaRemove()
Inputpanel	ProUIDialogInputpanelAdd()	ProUIDialogInputpanelRemove()
Label	ProUIDialogLabelAdd()	ProUIDialogLabelRemove()
Layout	ProUIDialogLayoutAdd()	ProUIDialogLayoutRemove()
List	ProUIDialogListAdd()	ProUIDialogListRemove()
Menubar	ProUIDialogMenubarAdd()	Not Applicable
Menupane	ProUIDialogMenupaneAdd()	ProUIDialogMenupaneRemove()
Optionmenu	ProUIDialogProgressbarAdd()	ProUIDialogOptionmenuRemove()
Progressbar	ProUIDialogProgressbarAdd()	ProUIDialogProgressbarRemove()
Pushbutton	ProUIDialogPushbuttonAdd()	ProUIDialogPushbuttonRemove()
Radiogroup	ProUIDialogRadiogroupAdd()	ProUIDialogRadiogroupRemove()
Separator	ProUIDialogSeparatorAdd()	ProUIDialogSeparatorRemove()
Slider	ProUIDialogSliderAdd()	ProUIDialogSliderRemove()
Spinbox	ProUIDialogSpinboxAdd()	ProUIDialogSpinboxRemove()
Tab	ProUIDialogTabAdd()	ProUIDialogTabRemove()
Table	ProUIDialogTableAdd()	ProUIDialogTableRemove()
Textarea	ProUIDialogTextareaAdd()	ProUIDialogTextareaRemove()
Thumbwheel	ProUIDialogThumbwheelAdd()	ProUIDialogThumbwheelRemove()
Tree	ProUIDialogTreeAdd()	ProUIDialogTreeRemove()

Pro/TOOLKIT provides functions to add components to a dialog. These functions accept an argument of type ProUIGridopts that determines the location and initial placement details of the component. In addition, a number of grid-specific attributes control the position and resizing of the newly created component within the grid of the Dialog.

These grid-specific attributes are listed as follows:

Attribute	Default Value	Description
column	PRO_UI_INSERT_NEW_COLUMN	The column of the grid into which the component should be added. A value of PRO_UI_INSERT_NEW_COLUMN indicates that the component should be added to a newly created column to the left of any existing columns.
row	PRO_UI_INSERT_NEW_ROW	The row of the grid into which the component should be added. A value of PRO_UI_INSERT_NEW_ROW indicates that the component should be added to a newly created row to the left of any existing rows.
horz_cells	1	The number of cells which the component should occupy in the existing grid in a horizontal direction.
vert_cells	1	The number of cells which the component should occupy in the existing grid in a vertical direction.
horz_resize	PRO_B_TRUE	A flag indicating whether the grid cell containing the component should resize horizontally.
vert_resize	PRO_B_TRUE	A flag indicating whether the grid cell containing the component should resize vertically.
attach_top	PRO_B_TRUE	Attach the item to the top neighbor
attach_bottom	PRO_B_TRUE	Attach the item to the bottom neighbor
attach_left	PRO_B_TRUE	Attach the item to the left neighbor
attach_right	PRO_B_TRUE	Attach the item to the right neighbor
top_offset	PRO_UI_USE_DEVICE_OFFSET	Offset to the top neighbor. The default value PRO_UI_USE_DEVICE_OFFSET inherits the offset from the owning dialog.
bottom_offset	PRO_UI_USE_DEVICE_OFFSET	Offset to the bottom neighbor. The default value PRO_UI_USE_DEVICE_OFFSET inherits the offset from the owning dialog.
left_offset	PRO_UI_USE_DEVICE_OFFSET	Offset to the left neighbor. The default value PRO_UI_USE_DEVICE_OFFSET inherits the offset from the owning dialog.
right_offset	PRO_UI_USE_DEVICE_OFFSET	Offset to the right neighbor. The default value PRO_UI_USE_DEVICE_OFFSET inherits the offset from the owning dialog.

Note: Components that are added to a dialog after it is displayed do not permit modification of all component attributes. When creating and displaying a dialog, it is preferable to use these functions to add the components before activating the dialog. If a component might be needed but should not be shown initially, add it before activation and set its `.Visible` attribute to false.

Dialog Action Callbacks

Functions Introduced

- **ProUIDialogPremanagenotifyActionSet()**
- **ProUIDialogPostmanagenotifyActionSet()**
- **ProUIDialogDestroynotifyActionSet()**
- **ProUIDialogCloseActionSet()**
- **ProUIDialogActivateActionSet()**

Use the function **ProUIDialogPremanagenotifyActionSet()** to set the function to be called when the dialog is about to be managed. For example, when a dialog box is displayed or redisplayed.

Use the function **ProUIDialogPostmanagenotifyActionSet()** to set the function to be called when the dialog has just been managed. For example, when a dialog box is displayed.

Use the function **ProUIDialogDestroynotifyActionSet()** to set the function to be called when the dialog is about to be destroyed.

Use the function **ProUIDialogCloseActionSet()** to set the action function to be called when the user attempts to close the dialog using the window close icon in the upper right corner of the dialog.

Use the function **ProUIDialogActivateActionSet()** to set the function to be called when the dialog has just been activated and made the current foreground window. The action function for a given dialog can be called only on a Windows platform under the following conditions:

- The dialog must not be the current foreground application.
- The dialog (when it is not the foreground application) is activated using one of the following methods:

- When the user clicks on the taskbar button for the given dialog.
- When the user switches to the given dialog using Alt+Tab.
- When the user clicks within the given dialog.

Cascade Button

Cascade Button Attributes

Attribute Name	Get Function	Set Function(s)
.AttachBottom	ProUICascadebuttonIsAttachedBottom()	ProUICascadebuttonAttachBottom() ProUICascadebuttonUnattachBottom()
.AttachTop	ProUICascadebuttonIsAttachedTop()	ProUICascadebuttonAttachTop() ProUICascadebuttonUnattachTop()
.AttachRight	ProUICascadebuttonIsAttachedRight()	ProUICascadebuttonAttachRight() ProUICascadebuttonUnattachRight()
.AttachLeft	ProUICascadebuttonIsAttachedLeft()	ProUICascadebuttonAttachLeft() ProUICascadebuttonUnattachLeft()
.BottomOffset	ProUICascadebuttonBottomoffsetGet()	ProUICascadebuttonBottomoffsetSet()
.Bitmap	ProUICascadebuttonBitmapGet()	ProUICascadebuttonBitmapSet()
.CascadeDirection	ProUICascadebuttonCascadedirectionGet()	ProUICascadebuttonCascadedirectionSet()
.ChildNames	ProUICascadebuttonChildnamesGet()	ProUICascadebuttonChildnamesSet()
.HelpText	ProUICascadebuttonHelpTextGet()	ProUICascadebuttonHelpTextSet()
.Label	ProUICascadebuttonTextGet()	ProUICascadebuttonTextSet()
.LeftOffset	ProUICascadebuttonLeftoffsetGet()	ProUICascadebuttonLeftoffsetSet()
.ParentName	ProUICascadebuttonParentnameGet()	ProUICascadebuttonParentnameSet()
.PopupMenu	ProUICascadebuttonPopupmenuGet()	ProUICascadebuttonPopupmenuSet()
.Resizable	ProUICascadebuttonIsResizable()	ProUICascadebuttonEnableResizing() ProUICascadebuttonDisableResizing()
.RightOffset	ProUICascadebuttonRightoffsetGet()	ProUICascadebuttonRightoffsetSet()
.Sensitive	ProUICascadebuttonIsEnabled()	ProUICascadebuttonEnable() ProUICascadebuttonDisable()
.TopOffset	ProUICascadebuttonTopoffsetGet()	ProUICascadebuttonTopoffsetSet()

Attribute Name	Get Function	Set Function(s)
.Visible	ProUICascadebuttonIsVisible()	ProUICascadebuttonShow() ProUICascadebuttonHide()

Checkbutton

Checkbutton Attributes

Attribute Name	Get Function	Set Function(s)
.AttachBottom	ProUICheckbuttonIsAttachedBottom()	ProUICheckbuttonAttachBottom() ProUICheckbuttonUnattachBottom
.AttachTop	ProUICheckbuttonIsAttachedTop()	ProUICheckbuttonAttachTop() ProUICheckbuttonUnattachTop()
.AttachRight	ProUICheckbuttonIsAttachedRight()	ProUICheckbuttonAttachRight() ProUICheckbuttonUnattachRight()
.AttachLeft	ProUICheckbuttonIsAttachedLeft()	ProUICheckbuttonAttachLeft() ProUICheckbuttonUnattachLeft()
.BottomOffset	ProUICheckbuttonBottomoffsetGet()	ProUICheckbuttonBottomoffsetSet()
.Bitmap	ProUICheckbuttonBitmapGet()	ProUICheckbuttonBitmapSet()
.ButtonStyle	ProUICheckbuttonButtonstyleGet()	ProUICheckbuttonButtonstyleSet()
.HelpText	ProUICheckbuttonHelptextGet()	ProUICheckbuttonHelptextSet()
.Label	ProUICheckbuttonTextGet()	ProUICheckbuttonTextSet()
.LeftOffset	ProUICheckbuttonLeftoffsetGet()	ProUICheckbuttonLeftoffsetSet()
.ParentName	ProUICheckbuttonParentnameGet()	Not Applicable
.PopupMenu	ProUICheckbuttonPopupmenuGet()	ProUICheckbuttonPopupmenuSet()
.Resizable	ProUICheckbuttonIsResizable()	ProUICheckbuttonEnableResizing() ProUICheckbuttonDisableResizing()
.RightOffset	ProUICheckbuttonRightoffsetGet()	ProUICheckbuttonRightoffsetSet()
.Set	ProUICheckbuttonGetState()	ProUICheckbuttonSet() ProUICheckbuttonUnset()
.Sensitive	ProUICheckbuttonIsEnabled()	ProUICheckbuttonEnable() ProUICheckbuttonDisable()
.TopOffset	ProUICheckbuttonTopoffsetGet()	ProUICheckbuttonTopoffsetSet()

Attribute Name	Get Function	Set Function(s)
.Visible	ProUICheckbuttonIsVisible()	ProUICheckbuttonShow() ProUICheckbuttonHide()

Checkbutton Operations

Functions Introduced

- **ProUICheckbuttonAnchorSet()**
- **ProUICheckbuttonSizeSet()**
- **ProUICheckbuttonPositionSet()**
- **ProUICheckbuttonPositionGet()**
- **ProUICheckbuttonSizeGet()**

Use the function **ProUICheckbuttonAnchorSet()** to set the position of the checkbutton with respect to a given anchor location. This function is applicable only if the parent of the checkbutton is a drawing area. The input argument *anchor* determines which part of the component is being positioned.

Use the function **ProUICheckbuttonSizeSet()** to set the size of the checkbutton in pixels. This operation is applicable only if the parent is a drawing area.

Use the function **ProUICheckbuttonPositionSet()** to set the position to the checkbutton with respect to its parent. This operation is applicable only if the parent is a drawing area.

Use the function **ProUICheckbuttonPositionGet()** to get the position of the checkbutton with respect to its parent. This operation is applicable only if the parent is a drawing area.

Use the function **ProUICheckbuttonSizeGet()** to get the size of the checkbutton. This operation is applicable only if the parent is a drawing area.

Checkbutton Action Callbacks

Functions Introduced

- **ProUICheckbuttonActivateActionSet()**

The function **ProUICheckbuttonActivateActionSet()** sets the callback action to be invoked when the user toggles the state of the checkbutton.

Drawing Area

Drawing Area Attributes

Attribute Name	Get Function	Set Function(s)
.ArcDirection	ProUIDrawingareaArcdirectionGet()	ProUIDrawingareaArcdirectionSet()
.ArcFillMode	ProUIDrawingareaArcfillmodeGet()	ProUIDrawingareaArcfillmodeSet()
.AttachBottom	ProUIDrawingarealsAttachedBottom()	ProUIDrawingareaAttachBottom() ProUIDrawingareaUnattachBottom()
.AttachTop	ProUIDrawingarealsAttachedTop()	ProUIDrawingareaAttachTop() ProUIDrawingareaUnattachTop()
.AttachRight	ProUIDrawingarealsAttachedRight()	ProUIDrawingareaAttachRight() ProUIDrawingareaUnattachRight()
.AttachLeft	ProUIDrawingarealsAttachedLeft()	ProUIDrawingareaAttachLeft() ProUIDrawingareaUnattachLeft()
.BackgroundColor	ProUIDrawingareaBackgroundcolorGet()	ProUIDrawingareaBackgroundcolorSet()
.BgColor	ProUIDrawingareaBgcolorGet()	ProUIDrawingareaBgcolorSet()
.BottomOffset	ProUIDrawingareaBottomoffsetGet()	ProUIDrawingareaBottomoffsetSet()
.ChildNames	ProUIDrawingareaChildnamesGet()	Not Applicable
.ClipChildren	ProUIDrawingareaClipchildrenGet()	ProUIDrawingareaClipchildrenSet()
.Decorated	ProUIDrawingarealsDecorated()	ProUIDrawingareaDecorate() ProUIDrawingareaUndecorate()
.DrawingHeight	ProUIDrawingareaDrawingheightGet()	ProUIDrawingareaDrawingheightSet()
.DrawingMode	ProUIDrawingareaDrawingmodeGet()	ProUIDrawingareaDrawingmodeSet()
.DrawingWidth	ProUIDrawingareaDrawingwidthGet()	ProUIDrawingareaDrawingwidthSet()
.FillMode	ProUIDrawingareaFillmodeGet()	ProUIDrawingareaFillmodeSet()
.FontClass	ProUIDrawingareaFontclassGet()	ProUIDrawingareaFontclassSet()
.FontSize	ProUIDrawingareaFontsizeGet()	ProUIDrawingareaFontsizeSet()
.FontStyle	ProUIDrawingareaFontstyleGet()	ProUIDrawingareaFontstyleSet()
.FgColor	ProUIDrawingareaFgcolorGet()	ProUIDrawingareaFgcolorSet()
.HelpText	ProUIDrawingareaHelptextGet()	ProUIDrawingareaHelptextSet()
.Images	ProUIDrawingarealmagesGet()	ProUIDrawingarealmagesSet()
.LeftOffset	ProUIDrawingareaLeftoffsetGet()	ProUIDrawingareaLeftoffsetSet()
.LineStyle	ProUIDrawingareaLinestyleGet()	ProUIDrawingareaLinestyleSet()
.ParentName	ProUIDrawingareaParentnameGet()	Not Applicable

Attribute Name	Get Function	Set Function(s)
.PolygonFillMode	ProUIDrawingareaPolygonfillmodeGet()	ProUIDrawingareaPolygonfillmodeSet()
.PopupMenu	ProUIDrawingareaPopupmenuGet()	ProUIDrawingareaPopupmenuSet()
.RightOffset	ProUIDrawingareaRightoffsetGet()	ProUIDrawingareaRightoffsetSet()
.Sensitive	ProUIDrawingarealsEnabled()	ProUIDrawingareaEnable() ProUIDrawingareaDisable()
.TopOffset	ProUIDrawingareaTopoffsetGet()	ProUIDrawingareaTopoffsetSet()
.Tracking	ProUIDrawingarealsTrackingEnabled()	ProUIDrawingareaEnableTracking() ProUIDrawingareaDisableTracking()
.Visible	ProUIDrawingarealsVisible()	ProUIDrawingareaShow() ProUIDrawingareaHide()

Adding and Removing Components

Component Name	Adding Functions	Removing Functions
Checkbutton	ProUIDrawingareaCheckbuttonAdd()	ProUIDrawingareaCheckbuttonRemove()
Drawingarea	ProUIDrawingareaDrawingareaAdd()	ProUIDrawingareaDrawingareaRemove()
Inputpanel	ProUIDrawingareaInputpanelAdd()	ProUIDrawingareaInputpanelRemove()
Label	ProUIDrawingareaLabelAdd()	ProUIDrawingareaLabelRemove()
Layout	ProUIDrawingareaLayoutAdd()	ProUIDrawingareaLayoutRemove()
List	ProUIDrawingareaListAdd()	ProUIDrawingareaListRemove()
Optionmenu	ProUIDrawingareaOptionmenuAdd()	ProUIDrawingareaOptionmenuRemove()
Progressbar	ProUIDrawingareaProgressbarAdd()	ProUIDrawingareaProgressbarRemove()
Pushbutton	ProUIDrawingareaPushbuttonAdd()	ProUIDrawingareaPushbuttonRemove()
Radiogroup	ProUIDrawingareaRadiogroupAdd()	ProUIDrawingareaRadiogroupRemove()
Slider	ProUIDrawingareaSliderAdd()	ProUIDrawingareaSliderRemove()
Spinbox	ProUIDrawingareaSpinboxAdd()	ProUIDrawingareaSpinboxRemove()
Tab	ProUIDrawingareaTabAdd()	ProUIDrawingareaTabRemove()
Table	ProUIDrawingareaTableAdd()	ProUIDrawingareaTableRemove()
Textarea	ProUIDrawingareaTextareaAdd()	ProUIDrawingareaTextareaRemove()
Thumbwheel	ProUIDrawingareaThumbwheelAdd()	ProUIDrawingareaThumbwheelRemove()
Tree	ProUIDrawingareaTreeAdd()	ProUIDrawingareaTreeRemove()

Components added to drawing areas are not managed in grids like components in most other containers. Instead, use the operations that set the component size and position to place the component how you wish in the drawing area.

Note: This means that components may possibly overlap each other in a drawing area depending on their individual placement.

Drawing Area Action Callbacks

Functions Introduced

- **ProUIDrawingareaEnterActionSet()**
- **ProUIDrawingareaExitActionSet()**
- **ProUIDrawingareaMoveActionSet()**
- **ProUIDrawingareaLbuttonarmActionSet()**
- **ProUIDrawingareaLbuttondisarmActionSet()**
- **ProUIDrawingareaLbuttonactivateActionSet()**
- **ProUIDrawingareaLbuttondblclkActionSet()**
- **ProUIDrawingareaMbuttonarmActionSet()**
- **ProUIDrawingareaMbuttondisarmActionSet()**
- **ProUIDrawingareaMbuttondisarmActionSet()**
- **ProUIDrawingareaMbuttonactivateActionSet()**
- **ProUIDrawingareaMbuttondblclkActionSet()**
- **ProUIDrawingareaRbuttonarmActionSet()**
- **ProUIDrawingareaRbuttondisarmActionSet()**
- **ProUIDrawingareaRbuttonactivateActionSet()**
- **ProUIDrawingareaRbuttondblclkActionSet()**
- **ProUIDrawingareaUpdateActionSet()**
- **ProUIDrawingareaResizeActionSet()**
- **ProUIDrawingareaPostmanagenotifyActionSet()**

Use the function **ProUIDrawingareaEnterActionSet()** to set the action function to be called when the user has moved the cursor into the drawing area. This action will be generated only if tracking is enabled for the drawing area.

Use the function **ProUIDrawingareaExitActionSet()** to set the function to be called when the user has moved the cursor out of the drawing area. This action will be generated only if tracking is enabled for the drawing area.

Use the function **ProUIDrawingareaMoveActionSet()** to set the function to be called when the cursor is moved over the drawing area. This action will be generated only if tracking is enabled for the drawing area.

Use the function **ProUIDrawingareaLbuttonarmActionSet()** to set the function to be called when the left mouse button is clicked in the drawing area.

Use the function **ProUIDrawingareaLbuttondisarmActionSet()** to set the function to be called when the left mouse button is released in the drawing area.

Use the function **ProUIDrawingareaLbuttonactivateActionSet()** to set the function to be called when the left mouse button is clicked and released in the drawing area.

Use the function **ProUIDrawingareaLbuttondblclkActionSet()** to set the function to be called when the left mouse button is double-clicked in the drawing area.

Use the function **ProUIDrawingareaMbuttonarmActionSet()** to set the function to be called when the middle mouse button is clicked in the drawing area.

Use the function **ProUIDrawingareaMbuttondisarmActionSet()** to set the function to be called when the middle mouse button is released in the drawing area.

Use the function **ProUIDrawingareaMbuttonactivateActionSet()** to set the function to be called when the middle mouse button is clicked and released in the drawing area.

Use the function **ProUIDrawingareaMbuttondblclkActionSet()** to set the function to be called when the middle mouse button is double-clicked in the drawing area.

Use the function **ProUIDrawingareaRbuttonarmActionSet()** to set the function to be called when the right mouse button is clicked in the drawing area.

Use the function

ProUIDrawingareaRbuttondisarmActionSet() to set the function to be called when the right mouse button is released in the drawing area.

Use the function

ProUIDrawingareaRbuttonactivateActionSet() to set the function to be called when the right mouse button is clicked and released in the drawing area.

Use the function

ProUIDrawingareaRbuttondblclkActionSet() to set the function to be called when the right mouse button is double-clicked in the drawing area.

Use the function **ProUIDrawingareaUpdateActionSet()** to set the function to be called when the drawing area needs to be updated due to a system-level color scheme change.

Use the function **ProUIDrawingareaResizeActionSet()** to set the function to be called when the drawing area is resized.

Note: Any graphics, text or images added to the drawing area is typically cleared after a resize.

Use the function

ProUIDrawingareaPostmanagenotifyActionSet() to set the function to be called when the drawing area has just been displayed. Use this callback to setup the initial graphics, text, and images in the drawing area.

Drawing Area Operations

Functions Introduced

- **ProUIDrawingareaAnchorSet()**
- **ProUIDrawingareaSizeGet()**
- **ProUIDrawingareaSizeSet()**
- **ProUIDrawingareaPositionGet()**
- **ProUIDrawingareaPositionSet()**
- **ProUIDrawingareaClear()**
- **ProUIDrawingareaCopyArea()**
- **ProUIDrawingareaPointDraw()**
- **ProUIDrawingareaPointsDraw()**

- **ProUIDrawingareaLineDraw()**
- **ProUIDrawingareaLinesDraw()**
- **ProUIDrawingareaPolylineDraw()**
- **ProUIDrawingareaRectDraw()**
- **ProUIDrawingareaRectsDraw()**
- **ProUIDrawingareaRectFill()**
- **ProUIDrawingareaRectsFill()**
- **ProUIDrawingareaShadowRectDraw()**
- **ProUIDrawingareaShadowRectsDraw()**
- **ProUIDrawingareaPolygonDraw()**
- **ProUIDrawingareaPolygonFill()**
- **ProUIDrawingareaArcDraw()**
- **ProUIDrawingareaArcsDraw()**
- **ProUIDrawingareaArcFill()**
- **ProUIDrawingareaArcsFill()**
- **ProUIDrawingareaEllipseDraw()**
- **ProUIDrawingareaEllipsesDraw()**
- **ProUIDrawingareaEllipseFill()**
- **ProUIDrawingareaEllipsesFill()**
- **ProUIDrawingarealmageDraw()**
- **ProUIDrawingarealmagesDraw()**
- **ProUIDrawingareaStringDraw()**
- **ProUIDrawingareaStringsDraw()**
- **ProUIDrawingareaStringsSizeGet()**
- **ProUIDrawingareaStringBaselineGet()**
- **ProUIDrawingarealmagesSizeGet()**
- **ProUIDrawingareaCursorposGet()**
- **ProUIDrawingareaCursorposSet()**

Use the function **ProUIDrawingareaAnchorSet()** to set the position of the drawing area with respect to a given anchor location. This function is applicable only if the parent of the drawing area is another drawing area.

Use the function **ProUIDrawingareaSizeGet()** to get the size of the drawing area. This operation is applicable only if the parent is a drawing area.

Use the function **ProUIDrawingareaSizeSet()** to set the size of the drawing area. This operation is applicable only if the parent is a drawing area.

Use the function **ProUIDrawingareaPositionGet()** to get the position of the drawing area with respect to its parent. This operation is applicable only if the parent is a drawing area.

Use the function **ProUIDrawingareaPositionSet()** to set the position to the drawing area with respect to its parent. This operation is applicable only if the parent is a drawing area.

Use the function **ProUIDrawingareaClear()** to clear the contents of the drawing area by painting it in the drawing background color.

Use the function **ProUIDrawingareaCopyArea()** to copy the contents within a given boundary at a location in the drawing area to another location.

Use the function **ProUIDrawingareaPointDraw()** to draw a point in the drawing area.

Use the function **ProUIDrawingareaPointsDraw()** to draw an array of points in the drawing area.

Use the function **ProUIDrawingareaLineDraw()** to draw a line in the drawing area.

Use the function **ProUIDrawingareaLinesDraw()** to draw a set of lines between in the drawing area. Each line will be drawn from the indicated start point in the array to the corresponding endpoint.

Use the function **ProUIDrawingareaPolylineDraw()** to draw a series of connected lines in the drawing area.

Use the function **ProUIDrawingareaRectDraw()** to draw a rectangle in the drawing area.

Use the function **ProUIDrawingareaRectsDraw()** to draw a set of rectangles in the drawing area.

Use the function **ProUIDrawingareaRectFill()** to draw a filled rectangle in the drawing area.

Use the function **ProUIDrawingareaRectsFill()** to draw a set of filled rectangles in the drawing area.

Use the function **ProUIDrawingareaShadowRectDraw()** to draw a rectangle with a shadow border.

Use the function **ProUIDrawingareaShadowRectsDraw()** to draw a set of rectangles with shadow borders.

Use the function **ProUIDrawingareaPolygonDraw()** to draw a polygon in the drawing area.

Use the function **ProUIDrawingareaPolygonFill()** to draw a filled polygon in the drawing area.

Use the function **ProUIDrawingareaArcDraw()** to draw an arc in the drawing area.

Use the function **ProUIDrawingareaArcsDraw()** to draw a set of arcs in the drawing area.

Use the function **ProUIDrawingareaArcFill()** to draw a filled arc in the drawing area.

Use the function **ProUIDrawingareaArcsFill()** to draw a set of filled arcs in the drawing.

Use the function **ProUIDrawingareaEllipseDraw()** to draw an ellipse in the drawing area.

Use the function **ProUIDrawingareaEllipsesDraw()** to draw a set of ellipses in the drawing area.

Use the function **ProUIDrawingareaEllipseFill()** to draw a filled ellipse in the drawing area.

Use the function **ProUIDrawingareaEllipsesFill()** to draw a set of filled ellipses in the drawing area.

Use the function **ProUIDrawingareaImageDraw()** to draw an image in the drawing area.

Use the function **ProUIDrawingareaImagesDraw()** to draw images at the given positions in the drawing area.

Use the function **ProUIDrawingareaStringDraw()** to draw a string at the given position in the drawing area.

Use the function **ProUIDrawingareaStringsDraw()** to draw strings at the given positions in the drawing.

Use the function **ProUIDrawingareaStringsSizeGet()** to get the size that the given text string will occupy in the drawing area, given the current drawing area font settings.

Use the function **ProUIDrawingareaStringbaselineGet()** to get the height from the top of the string border to the string baseline for the given text string in the drawing area, given the current drawing area font settings.

Use the function **ProUIDrawingareaImagesizeGet()** to get the size of the image in the drawing area.

Use the function **ProUIDrawingareaCursorposGet()** to get the position of the cursor in the drawing area.

Use the function **ProUIDrawingareaCursorposSet()** to set the cursor at the given location in the drawing area.

Input Panel

Input Panel Attributes

Attribute Name	Get Function	Set Function(s)
.AttachBottom	ProUIInputpanellsAttachedBottom()	ProUIInputpanelAttachBottom() ProUIInputpanelUnattachBottom()
.AttachTop	ProUIInputpanellsAttachedTop()	ProUIInputpanelAttachTop() ProUIInputpanelUnattachTop()
.AttachRight	ProUIInputpanellsAttachedRight()	ProUIInputpanelAttachRight() ProUIInputpanelUnattachRight()
.AttachLeft	ProUIInputpanellsAttachedLeft()	ProUIInputpanelAttachLeft() ProUIInputpanelUnattachLeft()
.Autohighlight	ProUIInputpanellsAutohighlightEnabled()	ProUIInputpanelAutohighlightEnable() ProUIInputpanelAutohighlightDisable()
.BottomOffset	ProUIInputpanelBottomoffsetGet()	ProUIInputpanelBottomoffsetSet()
.Columns	ProUIInputpanelColumnsGet()	ProUIInputpanelColumnsSet()
.Denominator	ProUIInputpanelDenominatorGet()	ProUIInputpanelDenominatorSet()
.Digits	ProUIInputpanelDigitsGet()	ProUIInputpanelDigitsSet()
.Double	ProUIInputpanelDoubleGet()	ProUIInputpanelDoubleSet()
.DoubleFormat	ProUIInputpanelDoubleformatGet()	ProUIInputpanelDoubleformatSet()
.Editable	ProUIInputpanellsEditable()	ProUIInputpanelEditable() ProUIInputpanelReadOnly()
.HelpText	ProUIInputpanelHelptextGet()	ProUIInputpanelHelptextSet()

Attribute Name	Get Function	Set Function(s)
.InputType	ProUIInputpanelInputtypeGet()	ProUIInputpanelInputtypeSet()
.Integer	ProUIInputpanelIntegerGet()	ProUIInputpanelIntegerSet()
.LeftOffset	ProUIInputpanelLeftoffsetGet()	ProUIInputpanelLeftoffsetSet()
.MinColumns	ProUIInputpanelMincolumnsGet()	ProUIInputpanelMincolumnsSet()
.MinDouble	ProUIInputpanelMindoubleGet()	ProUIInputpanelMindoubleSet()
.MaxDouble	ProUIInputpanelMaxdoubleGet()	ProUIInputpanelMaxdoubleSet()
.MinInteger	ProUIInputpanelMinintegerGet()	ProUIInputpanelMinintegerSet()
.MaxInteger	ProUIInputpanelMaxintegerGet()	ProUIInputpanelMaxintegerSet()
.MaxLen	ProUIInputpanelMaxlenGet()	ProUIInputpanelMaxlenSet()
.Numerator	ProUIInputpanelNumeratorGet()	ProUIInputpanelNumeratorSet()
.Ordinal	ProUIInputpanelOrdinalGet()	ProUIInputpanelOrdinalSet()
.ParentName	ProUIInputpanelParentnameGet()	Not Applicable
.Password	ProUIInputpanelPasswordcharGet()	ProUIInputpanelUsePasswordchars() ProUIInputpanelUseNormalchars() ProUIInputpanelContainsPassword() ProUIInputpanelPasswordcharSet()
.PopupMenu	ProUIInputpanelPopupmenuGet()	ProUIInputpanelPopupmenuSet()
.RightOffset	ProUIInputpanelRightoffsetGet()	ProUIInputpanelRightoffsetSet()
.Sensitive	ProUIInputpanelIsEnabled()	ProUIInputpanelEnable() ProUIInputpanelDisable()
.String	ProUIInputpanelStringGet()	ProUIInputpanelStringSet()
.TabCharsAlloww	ProUIInputpanelTabcharsAllow()	ProUIInputpanelTabcharsDisallow() ProUIInputpanelAllowsTabchars()
.TopOffset	ProUIInputpanelTopoffsetGet()	ProUIInputpanelTopoffsetSet()
.Value	ProUIInputpanelValueGet()	ProUIInputpanelValueSet()
.Visible	ProUIInputpanelIsVisible()	ProUIInputpanelShow() ProUIInputpanelHide()
.WideString	ProUIInputpanelWidestringGet()	ProUIInputpanelWidestringSet()

Input Panel Action Callbacks

Functions Introduced

- **ProUIInputpanelActivateActionSet()**
- **ProUIInputpanelFocusinActionSet()**
- **ProUIInputpanelFocusoutActionSet()**
- **ProUIInputpanelInputActionSet()**

Use the function **ProUIInputpanelActivateActionSet()** to set the action callback to be called when the user hits return in an input panel.

Use the function **ProUIInputpanelFocusinActionSet()** to set the focus in action for an input panel. This function is called when the user moves the cursor onto the input panel using the mouse or <TAB> key.

Use the function **ProUIInputpanelFocusoutActionSet()** to set the focus out action for an input panel. This function is called when the user moves the cursor off of the input panel using the mouse or <TAB> key.

Use the function **ProUIInputpanelInputActionSet()** to set the action callback to be called when the user enters a key in an input panel.

Input Panel Operations

Functions Introduced

- **ProUIInputpanelAnchorSet()**
- **ProUIInputpanelSizeGet()**
- **ProUIInputpanelSizeSet()**
- **ProUIInputpanelPositionGet()**
- **ProUIInputpanelPositionSet()**

Use the function **ProUIInputpanelAnchorSet()** to set the position of the input panel with respect to a given anchor location. This function is applicable only if the parent of the input panel is a drawing area.

Use the function **ProUIInputpanelSizeGet()** to get the size of the input panel. This operation is applicable only if the parent is a drawing area.

Use the function **ProUIInputpanelSizeSet()** to set the size of the input panel. This operation is applicable only if the parent is a drawing area.

Use the function **ProUIInputpanelPositionGet()** to get the position of the input panel with respect to its parent. This operation is applicable only if the parent is a drawing area.

Use the function **ProUIInputpanelPositionSet()** to set the position to the input panel with respect to its parent. This operation is applicable only if the parent is a drawing area.

Label

Label Attributes

Attribute Name	Get Function	Set Function(s)
.AttachBottom	ProUILabelsAttachedBottom()	ProUILabelAttachBottom() ProUILabelUnattachBottom()
.AttachTop	ProUILabelsAttachedTop()	ProUILabelAttachTop() ProUILabelUnattachTop()
.AttachRight	ProUILabelsAttachedRight()	ProUILabelAttachRight() ProUILabelUnattachRight()
.AttachLeft	ProUILabelsAttachedLeft()	ProUILabelAttachLeft() ProUILabelUnattachLeft()
.Bitmap	ProUILabelBitmapGet()	ProUILabelBitmapSet()
.BottomOffset	ProUILabelBottomoffsetGet()	ProUILabelBottomoffsetSet()
.Columns	ProUILabelColumnsGet()	ProUILabelColumnsSet()
.HelpText	ProUILabelHelptextGet()	ProUILabelHelptextSet()
.Label	ProUILabelTextGet()	ProUILabelTextSet()
.LeftOffset	ProUILabelLeftoffsetGet()	ProUILabelLeftoffsetSet()
.ParentName	ProUILabelParentnameGet()	Not Applicable
.PopupMenu	ProUILabelPopupmenuGet()	ProUILabelPopupmenuSet()
.Resizable	ProUILabelsResizable()	ProUILabelEnableResizing() ProUILabelDisableResizing()
.RightOffset	ProUILabelRightoffsetGet()	ProUILabelRightoffsetSet()
.Sensitive	ProUILabelsEnabled()	ProUILabelEnable() ProUILabelDisable()
.TopOffset	ProUILabelTopoffsetGet()	ProUILabelTopoffsetSet()
.Visible	ProUILabelsVisible()	ProUILabelShow() ProUILabelHide()

User Interface:
Dialogs

Label Operations

Functions Introduced

- **ProUILabelAnchorSet()**
- **ProUILabelSizeSet()**
- **ProUILabelPositionGet()**
- **ProUILabelPositionSet()**
- **ProUILabelSizeGet()**

Use the function **ProUILabelAnchorSet()** to set the location to position the label with respect to its parent. This field is used only if the parent is a drawing area.

Use the function **ProUILabelSizeSet()** to set the size of the label. This field is used only if the parent is a drawing area.

Use the function **ProUILabelPositionGet()** to get the position of the label with respect to its parent. This field is used only if the parent is a drawing area.

Use the function **ProUILabelPositionSet()** to set the position to the label with respect to its parent. This field is used only if the parent is a drawing area.

Use the function **ProUILabelSizeGet()** to get the size of the label. This field is used only if the parent is a drawing area.

Layout

Layout Attributes

Attribute Name	Get Function	Set Function(s)
.AttachBottom	ProUILayoutIsAttachedBottom()	ProUILayoutAttachBottom() ProUILayoutUnattachBottom()
.AttachTop	ProUILayoutIsAttachedTop()	ProUILayoutAttachTop() ProUILayoutUnattachTop()
.AttachRight	ProUILayoutIsAttachedRight()	ProUILayoutAttachRight() ProUILayoutUnattachRight()

Attribute Name	Get Function	Set Function(s)
.AttachLeft	ProUILayoutIsAttachedLeft()	ProUILayoutAttachLeft() ProUILayoutUnattachLeft()
.Bitmap	ProUILayoutBitmapGet()	ProUILayoutBitmapSet()
.BottomOffset	ProUILayoutBottomoffsetGet()	ProUILayoutBottomoffsetSet()
.ChildNames	ProUILayoutChildnamesGet()	Not Applicable
.HelpText	ProUILayoutHelpTextGet()	ProUILayoutHelpTextSet()
.Label	ProUILayoutTextGet()	ProUILayoutTextSet()
.LeftOffset	ProUILayoutLeftoffsetGet()	ProUILayoutLeftoffsetSet()
.Mapped	ProUILayoutIsMapped()	ProUILayoutMappedSet() ProUILayoutMappedUnset()
.ParentName	ProUILayoutParentnameGet()	Not Applicable
.PopupMenu	ProUILayoutPopupMenuGet()	ProUILayoutPopupMenuSet()
.RightOffset	ProUILayoutRightoffsetGet()	ProUILayoutRightoffsetSet()
.Sensitive	ProUILayoutIsEnabled()	ProUILayoutEnable() ProUILayoutDisable()
.TopOffset	ProUILayoutTopoffsetGet()	ProUILayoutTopoffsetSet()
.Visible	ProUILayoutIsVisible()	ProUILayoutShow() ProUILayoutHide()

Adding and Removing Components

Component Name	Adding Functions	Removing Functions
Checkbutton	ProUILayoutCheckbuttonAdd()	ProUILayoutCheckbuttonRemove()
Drawingarea	ProUILayoutDrawingareaAdd()	ProUILayoutDrawingareaRemove()
Inputpanel	ProUILayoutInputpanelAdd()	ProUILayoutInputpanelRemove()
Label	ProUILayoutLabelAdd()	ProUILayoutLabelRemove()
Layout	ProUILayoutLayoutAdd()	ProUILayoutLayoutRemove()
List	ProUILayoutListAdd()	ProUILayoutListRemove()
Optionmenu	ProUILayoutOptionmenuAdd()	ProUILayoutOptionmenuRemove()
Progressbar	ProUILayoutProgressbarAdd()	ProUILayoutProgressbarRemove()
Pushbutton	ProUILayoutPushbuttonAdd()	ProUILayoutPushbuttonRemove()
Radiogroup	ProUILayoutRadiogroupAdd()	ProUILayoutRadiogroupRemove()
Separator	ProUILayoutSeparatorAdd()	ProUILayoutSeparatorRemove()
Slider	ProUILayoutSliderAdd()	ProUILayoutSliderRemove()

Component Name	Adding Functions	Removing Functions
Spinbox	ProUILayoutSpinboxAdd()	ProUILayoutSpinboxRemove()
Tab	ProUILayoutTabAdd()	ProUILayoutTabRemove()
Table	ProUILayoutTableAdd()	ProUILayoutTableRemove()
Textarea	ProUILayoutTextareaAdd()	ProUILayoutTextareaRemove()
Thumbwheel	ProUILayoutThumbwheelAdd()	ProUILayoutThumbwheelRemove()
Tree	ProUILayoutTreeAdd()	ProUILayoutTreeRemove()

See the section on Adding and Removing Components for description of how to use the *ProUIGridopts* arguments when adding components to a layout.

Layout Operations

Functions Introduced

- **ProUILayoutAnchorSet()**
- **ProUILayoutSizeSet()**
- **ProUILayoutPositionGet()**
- **ProUILayoutPositionSet()**
- **ProUILayoutSizeGet()**
- **ProUILayoutIsMapped()**
- **ProUILayoutMappedSet()**
- **ProUILayoutMappedUnset()**

Use the function **ProUILayoutAnchorSet()** to set the position of the layout with respect to a given anchor location. This function is applicable only if the parent of the layout is a drawing area.

Use the function **ProUILayoutSizeSet()** to set the size of the layout. This field is used only if the parent is a drawing area.

Use the function **ProUILayoutPositionGet()** to get the position of the layout with respect to its parent. This field is used only if the parent is a drawing area.

Use the function **ProUILayoutPositionSet()** to set the position to the layout with respect to its parent. This field is used only if the parent is a drawing area.

Use the function **ProUILayoutSizeGet()** to get the size of the layout. This field is used only if the parent is a drawing area.

The function **ProUILayoutIsMapped()** specifies if the given layout component is mapped. The value of the output flag is **PRO_B_TRUE** if the layout is mapped, else it is **PRO_B_FALSE**.

The functions **ProUILayoutShow()** and **ProUILayoutHide()** show and hide the layout component respectively, along with its contents. Use the function **ProUILayoutMappedSet()** to keep the size of the layout unchanged while working with these functions. To collapse or expand the layout to its nominal size, use the function **ProUILayoutMappedUnset()**.

List

List Attributes

Attribute Name	Get Function	Set Function(s)
.AttachBottom	ProUIListIsAttachedBottom()	ProUIListAttachBottom() ProUIListUnattachBottom()
.AttachTop	ProUIListIsAttachedTop()	ProUIListAttachTop() ProUIListUnattachTop()
.AttachRight	ProUIListIsAttachedRight()	ProUIListAttachRight() ProUIListUnattachRight()
.AttachLeft	ProUIListIsAttachedLeft()	ProUIListAttachLeft() ProUIListUnattachLeft()
.BottomOffset	ProUIListBottomoffsetGet()	ProUIListBottomoffsetSet()
.Columns	ProUIListColumnsGet()	ProUIListColumnsSet()
.ColumnLabel	ProUIListColumnlabelGet()	ProUIListColumnlabelSet()
.HelpText	ProUIListHelpertextGet()	ProUIListHelpertextSet()
.ItemHelpText	ProUIListItemhelptextGet()	ProUIListItemhelptextSet()
.ItemImage	ProUIListItemimageGet()	ProUIListItemimageSet()
.Label	ProUIListLabelsGet()	ProUIListLabelsSet()
.Lastentereditem	ProUIListLastentereditemGet()	Not Applicable
.LeftOffset	ProUIListLeftoffsetGet()	ProUIListLeftoffsetSet()
.ListState	ProUIListStateGet()	ProUIListStateSet()
.ListType	ProUIListListtypeGet()	ProUIListListtypeSet()
.ParentName	ProUIListParentnameGet()	Not Applicable
.MinColumns	ProUIListMincolumnsSet()	ProUIListMincolumnsGet()

Attribute Name	Get Function	Set Function(s)
.MinRows	ProUListMinrowsGet()	ProUListMinrowsSet()
.Names	ProUListNamesGet()	ProUListNamesSet()
.PopupMenu	ProUListPopupmenuGet()	ProUListPopupmenuSet()
.RightOffset	ProUListRightoffsetGet()	ProUListRightoffsetSet()
.SelectedNames	ProUListSelectednamesGet()	ProUListSelectednamesSet()
.SelectionPolicy	ProUListSelectionpolicyGet()	ProUListSelectionpolicySet()
.Sensitive	ProUListIsEnabled()	ProUListEnable() ProUListDisable()
.TopOffset	ProUListTopoffsetGet()	ProUListTopoffsetSet()
.Visible	ProUListIsVisible()	ProUListShow() ProUListHide()
.VisibleRows	ProUListVisiblerowsGet()	ProUListVisiblerowsSet()

List Action Callbacks

Functions Introduced

- **ProUListActivateActionSet()**
- **ProUListSelectActionSet()**
- **ProUListTriggerhighlightActionSet()**
- **ProUListFocusinActionSet()**
- **ProUListFocusoutActionSet()**

Use the function **ProUListActivateActionSet()** to set the activate action for a list. This function is called when the return key is pressed or the mouse is double-clicked in the list.

Use the function **ProUListSelectActionSet()** to set the select action for a list component.

Use the function **ProUListTriggerhighlightActionSet()** to set the trigger highlight action for a list. This function is called when the user moves the mouse over an item on the list.

Use the function **ProUListFocusinActionSet()** to set the focus in action for a list. This function is called when the user moves the cursor onto of the list using the mouse or <TAB> key.

Use the function **ProUListFocusoutActionSet()** to set the focus out action for a list. This function is called when the user moves the cursor off of the list using the mouse or <TAB> key.

List Operations

Functions Introduced

- **ProUListAnchorSet()**
- **ProUListSizeSet()**
- **ProUListPositionGet()**
- **ProUListPositionSet()**
- **ProUListSizeGet()**
- **ProUListStateGet()**
- **ProUListStateSet()**

Use the function **ProUListAnchorSet()** to set the position of the list with respect to a given anchor location. This function is applicable only if the parent of the list is a drawing area.

Use the function **ProUListSizeSet()** to set the size of the list. This field is used only if the parent is a drawing area.

Use the function **ProUListPositionSet()** to set the position to the list with respect to its parent. This field is used only if the parent is a drawing area.

Use the function **ProUListPositionGet()** to get the position of the list with respect to its parent. This field is used only if the parent is a drawing area.

Use the function **ProUListSizeGet()** to get the size of the list. This field is used only if the parent is a drawing area.

Use the function **ProUListStateGet()** to get the state of the item in the list. The state is applicable only for a "check" type of list and refers to the checked or unchecked status of the item.

Use the function **ProUListStateSet()** to set the state of the item in the list. The state is applicable only for a "check" type of list and refers to the checked or unchecked status of the item.

Example 2: To use UI List Functions

The following example shows the source code for UI List functions. The application gets the names of the parts that constitute any given drawing and then populates the list area in a newly created dialog with those names.

```

#include <ProToolkit.h>
#include <ProDrawing.h>
#include <ProMessage.h>
#include <ProMdl.h>
#include <ProModelitem.h>
#include <ProWindows.h>
#include <ProDwgtable.h>
#include <ProUtil.h>
#include <ProUIDialog.h>
#include <ProUIList.h>
#include <ProUIPushbutton.h>
#include <ProUITextarea.h>
#include "UtilString.h"
#include "UtilFiles.h"
#include "TestError.h"
#include "UtilNames.h"

#define OK 1
#define CANCEL 0

void list_OKAction(char *dialog, char *component, ProAppData data);
void list_CancelAction(char *dialog, char *component, ProAppData data);
ProError UserUIListImplement ();

/*=====*\
    FUNCTION : list_OKAction()
    PURPOSE  : Action function for the "OK" button
\*=====*/
void list_OKAction(char *dialog, char *component, ProAppData data)
{
    ProError status;

    status = ProUIDialogExit(dialog, OK);
    TEST_CALL_REPORT ("ProUIDialogExit()", "list_OKAction()",
        status, status!=PRO_TK_NO_ERROR);
}
/*=====*\
    FUNCTION : list_CancelAction()
    PURPOSE  : Action function for the "Cancel" button
\*=====*/
void list_CancelAction(char *dialog, char *component, ProAppData data)
{
    ProError status;

    status = ProUIDialogExit(dialog, CANCEL);
    TEST_CALL_REPORT ("ProUIDialogExit()", "list_CancelAction()",
        status, status!=PRO_TK_NO_ERROR);
}

/*=====*\
    FUNCTION: UserUIListImplement

```



```

PURPOSE:  demonstrates how to use the UI List Names and Label Set
functions
\*=====*/
ProError UserUIListImplement ()
{
    ProError          status;
    ProDrawing         drawing;
    ProSolid           *solids;
    ProLine            wline;
    wchar_t           **model_names;
    int                i, exit_status, num;
    char               **strings, type;
    char               *list_dialog = "ug_uilist";
    char               *list_comp = "ug_uilist_comp";
    char               *list_txtarea = "ug_uilist_txtarea";
    char               *list_pbok = "ug_uilist_ok";
    char               *list_pbcancel = "ug_uilist_cancel";

/*-----*\
    Get current drawing
\*-----*/
    status = ProMdlCurrentGet ((ProMdl*)&drawing);
    TEST_CALL_REPORT ("ProMdlCurrentGet()", "UserUIListImplement()",
        status, status!=PRO_TK_NO_ERROR);

/*-----*\
    Get the models that make the drawing
\*-----*/
    status = ProDrawingSolidsCollect(drawing, &solids);
    TEST_CALL_REPORT ("ProDrawingTablesCollect()", "UserUIListImplement()",
        status, status!=PRO_TK_NO_ERROR);

/*-----*\
    Allocate memory for the char and wchar string arrays
\*-----*/
    if (status == PRO_TK_NO_ERROR)
    {
        status = ProArraySizeGet ((ProArray)solids, &num);
        TEST_CALL_REPORT ("ProArraySizeGet()", "UserUIListImplement()",
            status, status!=PRO_TK_NO_ERROR);
        status = ProArrayAlloc (num, sizeof(wchar_t *), 1,
            (ProArray *)&model_names);
        TEST_CALL_REPORT ("ProArrayAlloc()", "UserUIListImplement()",
            status, status!=PRO_TK_NO_ERROR);

        status = ProArrayAlloc(num, sizeof(char *), 1,
            (ProArray *)&strings);
        TEST_CALL_REPORT ("ProArrayAlloc()", "UserUIListImplement()",
            status, status!=PRO_TK_NO_ERROR);

/*-----*\

```

```

    Allocate memory for the char and wchar strings
/*-----*/
for (i=0; i<num; i++)
{
    model_names[i] = (wchar_t *)calloc(PRO_NAME_SIZE, sizeof(wchar_t));
    strings[i] = (char *)calloc(PRO_NAME_SIZE, sizeof(char));

/*-----*\
    Get the names of the models that makeup the drawing
/*-----*/
    status = ProMdlNameGet ((ProMdl)solids[i], model_names[i]);
    TEST_CALL_REPORT ("ProMdlNameGet()", "UserUIListImplement()",
        status, status!=PRO_TK_NO_ERROR);

/*-----*\
    Convert widestrings to strings
/*-----*/
    ProWstringToString(strings[i], model_names[i]);
}

/*-----*\
    Create and set the pushbutton activate actions
/*-----*/
    status = ProUIDialogCreate (list_dialog, list_dialog);
    TEST_CALL_REPORT ("ProUIDialogCreate()", "UserUIListImplement()",
        status, status!=PRO_TK_NO_ERROR );

    status = ProUIPushbuttonActivateActionSet (list_dialog, list_pbok,
        list_OKAction, NULL);
    TEST_CALL_REPORT ("ProUIPushbuttonActivateSet()",
        "UserUIListImplement()", status,
        status!=PRO_TK_NO_ERROR );

    status = ProUIPushbuttonActivateActionSet (list_dialog, list_pbcancel,
        list_CancelAction, NULL);
    TEST_CALL_REPORT ("ProUIPushbuttonActivateSet()",
        "UserUIListImplement()", status,
        status!=PRO_TK_NO_ERROR );

/*-----*\
    Populate the list with the model names after activating
    the dialog and destroy when done
/*-----*/

    status = ProUIListNamesSet (list_dialog, list_comp, num, strings);
    TEST_CALL_REPORT ("ProUIListNamesSet()", "UserUIListImplement()",
        status, status!=PRO_TK_NO_ERROR );

    status = ProUIListLabelsSet (list_dialog, list_comp, num, model_names);
    TEST_CALL_REPORT ("ProUIListLabelsSet()", "UserUIListImplement()",
        status, status!=PRO_TK_NO_ERROR );

```

```

ProStringToWstring(wline,
"These are the names of the parts\nthat makeup the drawing");

status = ProUITextareaValueSet(list_dialog, list_txtarea, wline);
TEST_CALL_REPORT ("ProUIListLabelsSet()", "UserUIListImplement()",
status, status!=PRO_TK_NO_ERROR );

status = ProUIDialogActivate (list_dialog, &exit_status);
TEST_CALL_REPORT ("ProUIDialogActivate()", "UserUIListImplement()",
status, status!=PRO_TK_NO_ERROR );

status = ProUIDialogDestroy (list_dialog);
TEST_CALL_REPORT ("ProUIDialogDestroy()", "UserUIListImplement()",
status, status!=PRO_TK_NO_ERROR );

/*-----*\
Free memory
\*-----*/

status = ProArrayFree ((ProArray*)&solids);
TEST_CALL_REPORT ("ProArrayFree()", "UserUIListImplement()",
status, status!=PRO_TK_NO_ERROR);

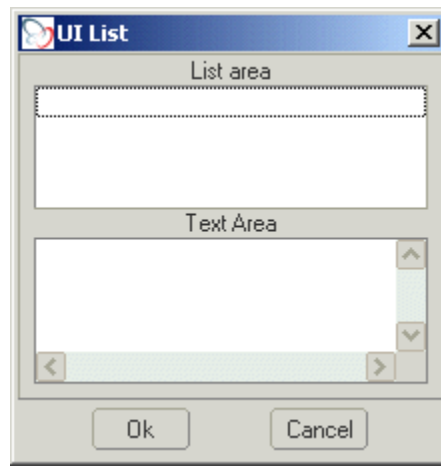
status = ProArrayFree((ProArray *)&model_names);
TEST_CALL_REPORT ("ProWstringproarrayFree()", "UserUIListImplement()",
status, status!=PRO_TK_NO_ERROR);

status = ProArrayFree((ProArray *)&strings);
TEST_CALL_REPORT ("ProStringproarrayFree()",
"UserUIListImplement()",
status, status!=PRO_TK_NO_ERROR);
}

return(PRO_TK_NO_ERROR);
}

```

Figure 16-4: UI List



Menubar

Menubar Attributes

Attribute Name	Get Function	Set Function(s)
.AttachBottom	ProUIMenubarIsAttachedBottom()	ProUIMenubarAttachBottom() ProUIMenubarUnattachBottom()
.AttachTop	ProUIMenubarIsAttachedTop()	ProUIMenubarAttachTop() ProUIMenubarUnattachTop()
.AttachRight	ProUIMenubarIsAttachedRight()	ProUIMenubarAttachRight() ProUIMenubarUnattachRight()
.AttachLeft	ProUIMenubarIsAttachedLeft()	ProUIMenubarAttachLeft() ProUIMenubarUnattachLeft()
.BottomOffset	ProUIMenubarBottomoffsetGet()	ProUIMenubarBottomoffsetSet()
.HelpText	ProUIMenubarHelptextGet()	ProUIMenubarHelptextSet()
.ItemHelpText	ProUIMenubarItemhelptextGet()	ProUIMenubarItemhelptextSet()
.Label	ProUIMenupaneTextGet()	ProUIMenupaneTextSet()
.LeftOffset	ProUIMenubarLeftoffsetGet()	ProUIMenubarLeftoffsetSet()
.Names	ProUIMenubarNamesGet()	ProUIMenubarNamesSet()

Attribute Name	Get Function	Set Function(s)
.PopupMenu	ProUIMenubarPopupMenuGet()	ProUIMenubarPopupMenuSet()
.RightOffset	ProUIMenubarRightoffsetGet()	ProUIMenubarRightoffsetSet()
.SelectableNames	ProUIMenubarSelectablenamesGet()	ProUIMenubarSelectablenamesSet()
.Sensitive	ProUIMenubarIsEnabled()	ProUIMenubarEnable() ProUIMenubarDisable()
.TopOffset	ProUIMenubarTopoffsetGet()	ProUIMenubarTopoffsetSet()
.Visible	ProUIMenubarIsVisible()	ProUIMenubarShow() ProUIMenubarHide()
.VisibleNames	ProUIMenubarVisiblenamesGet()	ProUIMenubarVisiblenamesSet()

Menupane

Menupane Attributes

Attribute Name	Get Function	Set Function(s)
.AttachBottom	ProUIMenupaneAttachedBottom()	ProUIMenupaneAttachBottom() ProUIMenupaneUnattachBottom()
.AttachTop	ProUIMenupaneAttachedTop()	ProUIMenupaneAttachTop() ProUIMenupaneUnattachTop()
.AttachRight	ProUIMenupaneAttachedRight()	ProUIMenupaneAttachRight() ProUIMenupaneUnattachRight()
.AttachLeft	ProUIMenupaneAttachedLeft()	ProUIMenupaneAttachLeft() ProUIMenupaneUnattachLeft()
.Bitmap	ProUIMenupaneBitmapGet()	ProUIMenupaneBitmapSet()
.BottomOffset	ProUIMenupaneBottomoffsetGet()	ProUIMenupaneBottomoffsetSet()
.ChildNames	ProUIMenupaneChildnamesGet()	Not Applicable
.LeftOffset	ProUIMenupaneLeftoffsetGet()	ProUIMenupaneLeftoffsetSet()
.PopupMenu	ProUIMenupanePopupMenuGet()	ProUIMenupanePopupMenuSet()
.RightOffset	ProUIMenupaneRightoffsetGet()	ProUIMenupaneRightoffsetSet()
.TopOffset	ProUIMenupaneTopoffsetGet()	ProUIMenupaneTopoffsetSet()
.Visible	ProUIMenupaneIsVisible()	ProUIMenupaneShow() ProUIMenupaneHide()

Adding and Removing Components

Component Name	Adding Functions	Removing Functions
Cascadebutton	ProUIMenupaneCascadebuttonAdd() ProUIMenupaneCascadebuttonInsert()	ProUIMenupaneCascadebuttonRemove()
Checkbutton	ProUIMenupaneCheckbuttonAdd() ProUIMenupaneCheckbuttonInsert()	ProUIMenupaneCheckbuttonRemove()
Pushbutton	ProUIMenupanePushbuttonAdd() ProUIMenupanePushbuttonInsert()	ProUIMenupanePushbuttonRemove()
Radiogroup	ProUIMenupaneRadiogroupAdd() ProUIMenupaneRadiogroupInsert()	ProUIMenupaneRadiogroupRemove()
Separator	ProUIMenupaneSeparatorAdd() ProUIMenupaneSeparatorInsert()	ProUIMenupaneSeparatorRemove()

Optionmenu

Optionmenu Attributes

Attribute Name	Get Function	Set Function(s)
.AttachBottom	ProUIOptionmenulsAttachedBottom()	ProUIOptionmenuAttachBottom() ProUIOptionmenuUnattachBottom()
.AttachTop	ProUIOptionmenulsAttachedTop()	ProUIOptionmenuAttachTop() ProUIOptionmenuUnattachTop()
.AttachRight	ProUIOptionmenulsAttachedRight()	ProUIOptionmenuAttachRight() ProUIOptionmenuUnattachRight()
.AttachLeft	ProUIOptionmenulsAttachedLeft()	ProUIOptionmenuAttachLeft() ProUIOptionmenuUnattachLeft()
.BottomOffset	ProUIOptionmenuBottomoffsetGet()	ProUIOptionmenuBottomoffsetSet()
.Columns	ProUIOptionmenuColumnsGet()	ProUIOptionmenuColumnsSet()
.Editable	ProUIOptionmenulsEditable()	ProUIOptionmenuEditable() ProUIOptionmenuReadOnly()
.HelpText	ProUIOptionmenuHelptextGet()	ProUIOptionmenuHelptextSet()
.ItemHelpText	ProUIOptionmenultemhelpertextGet()	ProUIOptionmenultemhelpertextSet()
.ItemImage	ProUIOptionmenultemimageGet()	ProUIOptionmenultemimageSet()

Attribute Name	Get Function	Set Function(s)
.Label	ProUIOptionmenuLabelsGet()	ProUIOptionmenuLabelsSet()
.Lastentereditem	ProUIOptionmenuLastentereditemGet()	Not Applicable
.LeftOffset	ProUIOptionmenuLeftoffsetGet()	ProUIOptionmenuLeftoffsetSet()
.MinColumns	ProUIOptionmenuMincolumnsGet()	ProUIOptionmenuMincolumnsSet()
.Names	ProUIOptionmenuNamesGet()	ProUIOptionmenuNamesSet()
.ParentName	ProUIOptionmenuParentnameGet()	Not Applicable
.PopupMenu	ProUIOptionmenuPopupMenuGet()	ProUIOptionmenuPopupMenuSet()
.RightOffset	ProUIOptionmenuRightoffsetGet()	ProUIOptionmenuRightoffsetSet()
.SelectedNames	ProUIOptionmenuSelectednamesGet()	ProUIOptionmenuSelectednamesSet()
.Sensitive	ProUIOptionmenulsEnabled()	ProUIOptionmenuEnable() ProUIOptionmenuDisable()
.TopOffset	ProUIOptionmenuTopoffsetGet()	ProUIOptionmenuTopoffsetSet()
.Value	ProUIOptionmenuValueGet()	ProUIOptionmenuValueSet()
.Visible	ProUIOptionmenulsVisible()	ProUIOptionmenuShow() ProUIOptionmenuHide()
.VisibleRows	ProUIOptionmenuVisiblerowsGet()	ProUIOptionmenuVisiblerowsSet()

Optionmenu Action Callbacks

Functions Introduced

- **ProUIOptionmenuActivateActionSet()**
- **ProUIOptionmenuSelectActionSet()**
- **ProUIOptionmenuInputActionSet()**
- **ProUIOptionmenuTriggerhighlightActionSet()**
- **ProUIOptionmenuFocusinActionSet()**
- **ProUIOptionmenuFocusoutActionSet()**

Use the function **ProUIOptionmenuActivateActionSet()** to set the activate action for an option menu. This function is called when the user modifies the contents of the option menu.

Use the function **ProUIOptionmenuSelectActionSet()** to set the select action for a option menu component.

Use the function **ProUIOptionmenuInputActionSet()** to set the input action for an optionmenu. This function is called when the user changes the contents of the option menu. This is only valid for editable optionmenus.

Use the function

ProUIOptionmenuTriggerhighlightActionSet() to set the trigger highlight action for an optionmenu. This function is called when the user moves the mouse on an item in the drop down list of the optionmenu.

Use the function **ProUIOptionmenuFocusinActionSet()** to set the focus in action for an optionmenu.

Use the function **ProUIOptionmenuFocusoutActionSet()** to set the focus out action for an optionmenu.

Optionmenu Operations

Functions Introduced

- **ProUIOptionmenuAnchorSet()**
- **ProUIOptionmenuSizeSet()**
- **ProUIOptionmenuPositionSet()**
- **ProUIOptionmenuPositionGet()**
- **ProUIOptionmenuSizeGet()**

Use the function **ProUIOptionmenuAnchorSet()** to the position of the option menu with respect to a given anchor location. This function is applicable only if the parent of the option menu is a drawing area.

Use the function **ProUIOptionmenuSizeSet()** to set the size of the optionmenu. This field is used only if the parent is a drawing area.

Use the function **ProUIOptionmenuPositionSet()** to set the position to the optionmenu with respect to its parent. This field is used only if the parent is a drawing area.

Use the function **ProUIOptionmenuPositionGet()** to get the position of the optionmenu with respect to its parent. This field is used only if the parent is a drawing area.

Use the function **ProUIOptionmenuSizeGet()** to get the size of the optionmenu. This field is used only if the parent is a drawing area.

Progressbar

Progressbar Attributes

Attribute Name	Get Function	Set Function(s)
.AttachBottom	ProUIProgressbarIsAttachedBottom()	ProUIProgressbarAttachBottom() ProUIProgressbarUnattachBottom()
.AttachTop	ProUIProgressbarIsAttachedTop()	ProUIProgressbarAttachTop() ProUIProgressbarUnattachTop()
.AttachRight	ProUIProgressbarIsAttachedRight()	ProUIProgressbarAttachRight() ProUIProgressbarUnattachRight()
.AttachLeft	ProUIProgressbarIsAttachedLeft()	ProUIProgressbarAttachLeft() ProUIProgressbarUnattachLeft()
.BottomOffset	ProUIProgressbarBottomoffsetGet()	ProUIProgressbarBottomoffsetSet()
.HelpText	ProUIProgressbarHelpTextGet()	ProUIProgressbarHelpTextSet()
.Integer	ProUIProgressbarIntegerGet()	ProUIProgressbarIntegerSet()
.LeftOffset	ProUIProgressbarLeftoffsetGet()	ProUIProgressbarLeftoffsetSet()
.Length	ProUIProgressbarLengthGet()	ProUIProgressbarLengthSet()
.MaxInteger	ProUIProgressbarMaxintegerGet()	ProUIProgressbarMaxintegerSet()
.MinInteger	ProUIProgressbarMinintegerGet()	ProUIProgressbarMinintegerSet()
.Orientation	ProUIProgressbarOrientationGet()	ProUIProgressbarOrientationSet()
.ParentName	ProUIProgressbarParentnameGet()	Not Applicable
.PopupMenu	ProUIProgressbarPopupMenuGet()	ProUIProgressbarPopupMenuSet()
.ProgressStyle	ProUIProgressbarProgressstyleGet()	ProUIProgressbarProgressstyleSet()
.RightOffset	ProUIProgressbarRightoffsetGet()	ProUIProgressbarRightoffsetSet()
.TopOffset	ProUIProgressbarTopoffsetGet()	ProUIProgressbarTopoffsetSet()
.Visible	ProUIProgressbarIsVisible()	ProUIProgressbarShow() ProUIProgressbarHide()

Progressbar Operations

Functions Introduced

- **ProUIProgressbarAnchorSet()**
- **ProUIProgressbarSizeSet()**
- **ProUIProgressbarPositionSet()**
- **ProUIProgressbarPositionGet()**
- **ProUIProgressbarSizeGet()**

Use the function **ProUIProgressbarAnchorSet()** to set the position of the progressbar with respect to a given anchor location. This function is applicable only if the parent of the progressbar is a drawing area.

Use the function **ProUIProgressbarSizeSet()** to set the size of the progressbar. This field is used only if the parent is a drawing area.

Use the function **ProUIProgressbarPositionSet()** to set the position to the progressbar with respect to its parent. This field is used only if the parent is a drawing area.

Use the function **ProUIProgressbarPositionGet()** to get the position to the progressbar with respect to its parent. This field is used only if the parent is a drawing area.

Use the function **ProUIProgressbarSizeGet()** to get the size of the progressbar. This field is used only if the parent is a drawing area.

Pushbutton

Pushbutton Attributes

Attribute Name	Get Function	Set Function(s)
.AttachBottom	ProUIPushbuttonIsAttachedBottom()	ProUIPushbuttonAttachBottom() ProUIPushbuttonUnattachBottom
.AttachTop	ProUIPushbuttonIsAttachedTop()	ProUIPushbuttonAttachTop() ProUIPushbuttonUnattachTop()

Attribute Name	Get Function	Set Function(s)
.AttachRight	ProUIPushbuttonIsAttachedRight()	ProUIPushbuttonAttachRight() ProUIPushbuttonUnattachRight()
.AttachLeft	ProUIPushbuttonIsAttachedLeft()	ProUIPushbuttonAttachLeft() ProUIPushbuttonUnattachLeft()
.BottomOffset	ProUIPushbuttonBottomoffsetGet()	ProUIPushbuttonBottomoffsetSet()
.Bitmap	ProUIPushbuttonBitmapGet()	ProUIPushbuttonBitmapSet()
.ButtonStyle	ProUIPushbuttonButtonstyleGet()	ProUIPushbuttonButtonstyleSet()
.HelpText	ProUIPushbuttonHelpTextGet()	ProUIPushbuttonHelpTextSet()
.Label	ProUIPushbuttonTextGet()	ProUIPushbuttonTextSet()
.LeftOffset	ProUIPushbuttonLeftoffsetGet()	ProUIPushbuttonLeftoffsetSet()
.ParentName	ProUIPushbuttonParentnameGet()	Not Applicable
.PopupMenu	ProUIPushbuttonPopupMenuGet()	ProUIPushbuttonPopupMenuSet()
.Resizable	ProUIPushbuttonIsResizable()	ProUIPushbuttonEnableResizing() ProUIPushbuttonDisableResizing()
.RightOffset	ProUIPushbuttonRightoffsetGet()	ProUIPushbuttonRightoffsetSet()
.Sensitive	ProUIPushbuttonIsEnabled()	ProUIPushbuttonEnable() ProUIPushbuttonDisable()
.TopOffset	ProUIPushbuttonTopoffsetGet()	ProUIPushbuttonTopoffsetSet()
.Visible	ProUIPushbuttonIsVisible()	ProUIPushbuttonShow() ProUIPushbuttonHide()

Pushbutton Operations

Functions Introduced

- **ProUIPushbuttonAnchorSet()**
- **ProUIPushbuttonSizeSet()**
- **ProUIPushbuttonPositionSet()**
- **ProUIPushbuttonPositionGet()**
- **ProUIPushbuttonSizeGet()**

Use the function **ProUIPushbuttonAnchorSet()** to set the position of the pushbutton with respect to a given anchor location. This function is applicable only if the parent of the Pushbutton is a drawing area.

Use the function **ProUIPushbuttonSizeSet()** to set the size of the pushbutton. This field is used only if the parent is a drawing area.

Use the function **ProUIPushbuttonPositionSet()** to set the position to the pushbutton with respect to its parent. This field is used only if the parent is a drawing area.

Use the function **ProUIPushbuttonPositionGet()** to get the position to the pushbutton with respect to its parent. This field is used only if the parent is a drawing area.

Use the function **ProUIPushbuttonSizeGet()** to get the size of the pushbutton. This field is used only if the parent is a drawing area.

Pushbutton Action Callbacks

Function Introduced

- **ProUIPushbuttonActivateActionSet()**

Use the function **ProUIPushbuttonActivateActionSet()** to set the activate action for a pushbutton. This function is called when the user selects the pushbutton.

Example 3: Controlling Component Visibility or Sensitivity at Runtime

```
#include <ProToolkit.h>
#include <ProUIDialog.h>
#include <ProUIPushbutton.h>

#define UGUIVISIBILITY "uguivisibility"

/*=====*\
FUNCTION: UserCloseAction
PURPOSE: Action function for the Close button in this example
\*=====*/
static void UserCloseAction (char* dialog, char* button, ProAppData data)
{
    ProUIDialogExit (UGUIVISIBILITY, PRO_TK_NO_ERROR);
}

/*=====*\
FUNCTION: UserVisibilityToggle
PURPOSE: Action function for the "Visible" checkbox
\*=====*/
static void UserVisibilityToggle (char* dialog, char* button, ProAppData
data)
{
    ProBoolean checked;

    ProUICheckbuttonGetState (UGUIVISIBILITY, "VisibleCheck", &checked);
    if (checked)
        ProUIPushbuttonShow (UGUIVISIBILITY, "TargetBtn");
}
```

```

else
ProUIPushbuttonHide (UGUIVISIBILITY, "TargetBtn");
}

/*=====*\
FUNCTION: UserSensitivityToggle
PURPOSE: Action function for the "Sensitive" checkbox
\*=====*/
static void UserSensitivityToggle (char* dialog, char* button, ProAppData
data)
{
ProBoolean checked;

ProUICheckbuttonGetState (UGUIVISIBILITY, "SensitiveCheck", &checked);
if (checked)
ProUIPushbuttonEnable (UGUIVISIBILITY, "TargetBtn");
else
ProUIPushbuttonDisable (UGUIVISIBILITY, "TargetBtn");
}

/*=====*\
FUNCTION: UserRelabelAction
PURPOSE: Action function for the "Label" input panel
\*=====*/
static void UserRelabelAction (char* dialog, char* button,
ProAppData data)
{
wchar_t* label;

ProUIInputpanelValueGet (UGUIVISIBILITY, "ButtonLabel", &label);
ProUIPushbuttonTextSet (UGUIVISIBILITY, "TargetBtn", label);
ProWstringFree (label);
}

/*=====*\
FUNCTION: UserUIVisibilityExample
PURPOSE: Shows example of using runtime visibility and sensitivity of
components
\*=====*/
int UserUIVisibilityExample ()
{
ProLine wline;
int status;

/*-----*\
Load the dialog from the resource file
\*-----*/
ProUIDialogCreate(UGUIVISIBILITY, UGUIVISIBILITY);

/*-----*\
Set the OK and Cancel button actions

```

```

\*-----*/
ProUIPushbuttonActivateActionSet(UGUIVISIBILITY,"CloseButton",
                                UserCloseAction, NULL);

ProUICheckbuttonActivateActionSet(UGUIVISIBILITY,"VisibleCheck",
                                UserVisibilityToggle, NULL);
ProUICheckbuttonActivateActionSet(UGUIVISIBILITY,"SensitiveCheck",
                                UserSensitivityToggle, NULL);
ProUIInputpanelInputActionSet(UGUIVISIBILITY,"ButtonLabel",
                              UserRelabelAction, NULL);
/*-----*\
    Display and activate the dialog
\*-----*/
ProUIDialogActivate(UGUIVISIBILITY, &status);

/*-----*\
    Remove the dialog from memory
\*-----*/
ProUIDialogDestroy(UGUIVISIBILITY);

return(1);
}

```

Figure 16-5: Component Visibility



Radiogroup

Radiogroup Attributes

Attribute Name	Get Function	Set Function(s)
.AttachBottom	ProUIRadiogroupsAttachedBottom()	ProUIRadiogroupAttachBottom() ProUIRadiogroupUnattachBottom
.AttachTop	ProUIRadiogroupsAttachedTop()	ProUIRadiogroupAttachTop() ProUIRadiogroupUnattachTop()
.AttachRight	ProUIRadiogroupsAttachedRight()	ProUIRadiogroupAttachRight() ProUIRadiogroupUnattachRight()

Attribute Name	Get Function	Set Function(s)
.AttachLeft	ProUIRadiogroupIsAttachedLeft()	ProUIRadiogroupAttachLeft() ProUIRadiogroupUnattachLeft()
.BottomOffset	ProUIRadiogroupBottomoffsetGet()	ProUIRadiogroupBottomoffsetSet()
.ButtonStyle	ProUIRadiogroupButtonstyleGet()	ProUIRadiogroupButtonstyleSet()
.HelpText	ProUIRadiogroupHelpTextGet()	ProUIRadiogroupHelpTextSet()
.ItemHelpText	ProUIRadiogroupItemhelpTextGet()	ProUIRadiogroupItemhelpTextSet()
.ItemImage	ProUIRadiogroupItemImageGet()	ProUIRadiogroupItemImageSet()
.Labels	ProUIRadiogroupLabelsGet()	ProUIRadiogroupLabelsSet()
.LeftOffset	ProUIRadiogroupLeftoffsetGet()	ProUIRadiogroupLeftoffsetSet()
.Names	ProUIRadiogroupNamesGet()	ProUIRadiogroupNamesSet()
.Orientation	ProUIRadiogroupOrientationGet()	ProUIRadiogroupOrientationSet()
.ParentName	ProUIRadiogroupParentnameGet()	Not Applicable
.PopupMenu	ProUIRadiogroupPopupMenuGet()	ProUIRadiogroupPopupMenuSet()
.Resizable	ProUIRadiogroupIsResizable()	ProUIRadiogroupEnableResizing() ProUIRadiogroupDisableResizing()
.RightOffset	ProUIRadiogroupRightoffsetGet()	ProUIRadiogroupRightoffsetSet()
.SelectedNames	ProUIRadiogroupSelectednamesGet()	ProUIRadiogroupSelectednamesSet()
.Sensitive	ProUIRadiogroupIsEnabled()	ProUIRadiogroupEnable() ProUIRadiogroupDisable()
.TopOffset	ProUIRadiogroupTopoffsetGet()	ProUIRadiogroupTopoffsetSet()
.Visible	ProUIRadiogroupIsVisible()	ProUIRadiogroupShow() ProUIRadiogroupHide()

Radiogroup Operations

Functions Introduced

- **ProUIRadiogroupAnchorSet()**
- **ProUIRadiogroupSizeSet()**
- **ProUIRadiogroupPositionSet()**
- **ProUIRadiogroupPositionGet()**
- **ProUIRadiogroupSizeGet()**

Use the function **ProUIRadiogroupAnchorSet()** to set the position of the radiogroup with respect to a given anchor location. This function is applicable only if the parent of the Radiogroup is a drawing area.

Use the function **ProUIRadiogroupSizeSet()** to set the size of the radiogroup. This field is used only if the parent is a drawing area.

Use the function **ProUIRadiogroupPositionSet()** to set the position to the radiogroup with respect to its parent. This field is used only if the parent is a drawing area.

Use the function **ProUIRadiogroupPositionGet()** to get the position to the radiogroup with respect to its parent. This field is used only if the parent is a drawing area.

Use the function **ProUIRadiogroupSizeGet()** to get the size of the radiogroup. This field is used only if the parent is a drawing area.

Radiogroup Action Callback

Function Introduced:

- **ProUIRadiogroupSelectActionSet()**

Use the function **ProUIRadiogroupSelectActionSet()** to set the select action for a radio group. This function is called when the user selects one of the buttons in the radio group.

Separator

Separator Attributes

Attribute Name	Get Function	Set Function(s)
.AttachBottom	ProUISeparatorIsAttachedBottom()	ProUISeparatorAttachBottom() ProUISeparatorUnattachBottom
.AttachTop	ProUISeparatorIsAttachedTop()	ProUISeparatorAttachTop() ProUISeparatorUnattachTop()
.AttachRight	ProUISeparatorIsAttachedRight()	ProUISeparatorAttachRight() ProUISeparatorUnattachRight()
.AttachLeft	ProUISeparatorIsAttachedLeft()	ProUISeparatorAttachLeft() ProUISeparatorUnattachLeft()
.BottomOffset	ProUISeparatorBottomoffsetGet()	ProUISeparatorBottomoffsetSet()
.LeftOffset	ProUISeparatorLeftoffsetGet()	ProUISeparatorLeftoffsetSet()
.PopupMenu	ProUISeparatorPopupMenuGet()	ProUISeparatorPopupMenuSet()

Attribute Name	Get Function	Set Function(s)
.RightOffset	ProUISeparatorRightoffsetGet()	ProUISeparatorRightoffsetSet()
.TopOffset	ProUISeparatorTopoffsetGet()	ProUISeparatorTopoffsetSet()
.Visible	ProUISeparatorIsVisible()	ProUISeparatorShow() ProUISeparatorHide()

Slider

Slider Attributes

Attribute Name	Get Function	Set Function(s)
.AttachBottom	ProUISliderIsAttachedBottom()	ProUISliderAttachBottom() ProUISliderUnattachBottom
.AttachTop	ProUISliderIsAttachedTop()	ProUISliderAttachTop() ProUISliderUnattachTop()
.AttachRight	ProUISliderIsAttachedRight()	ProUISliderAttachRight() ProUISliderUnattachRight()
.AttachLeft	ProUISliderIsAttachedLeft()	ProUISliderAttachLeft() ProUISliderUnattachLeft()
.BottomOffset	ProUISliderBottomoffsetGet()	ProUISliderBottomoffsetSet()
.HelpText	ProUISliderHelpTextGet()	ProUISliderHelpTextSet()
.Integer	ProUISliderIntegerGet()	ProUISliderIntegerSet()
.LeftOffset	ProUISliderLeftoffsetGet()	ProUISliderLeftoffsetSet()
.Length	ProUISliderLengthGet()	ProUISliderLengthSet()
.MaxInteger	ProUISliderMaxintegerGet()	ProUISliderMaxintegerSet()
.MinInteger	ProUISliderMinintegerGet()	ProUISliderMinintegerSet()
.Orientation	ProUISliderOrientationGet()	ProUISliderOrientationSet()
.ParentName	ProUISliderParentnameGet()	Not Applicable
.PopupMenu	ProUISliderPopupmenuGet()	ProUISliderPopupmenuSet()
.RightOffset	ProUISliderRightoffsetGet()	ProUISliderRightoffsetSet()
.Sensitive	ProUISliderIsEnabled()	ProUISliderEnable() ProUISliderDisable()
.TopOffset	ProUISliderTopoffsetGet()	ProUISliderTopoffsetSet()

Attribute Name	Get Function	Set Function(s)
.Tracking	ProUISliderIsTrackingEnabled()	ProUISliderEnableTracking() ProUISliderDisableTracking()
.Visible	ProUISliderIsVisible()	ProUISliderShow() ProUISliderHide()

Slider Operations

Functions Introduced

- **ProUISliderAnchorSet()**
- **ProUISliderSizeSet()**
- **ProUISliderPositionSet()**
- **ProUISliderPositionGet()**
- **ProUISliderSizeGet()**

Use the function **ProUISliderAnchorSet()** to set the position of the slider with respect to a given anchor location. This function is applicable only if the parent of the Slider is a drawing area.

Use the function **ProUISliderSizeSet()** to set the size of the slider. This field is used only if the parent is a drawing area.

Use the function **ProUISliderPositionSet()** to set the position to the slider with respect to its parent. This field is used only if the parent is a drawing area.

Use the function **ProUISliderPositionGet()** to get the position to the slider with respect to its parent. This field is used only if the parent is a drawing area.

Use the function **ProUISliderSizeGet()** to get the size of the slider. This field is used only if the parent is a drawing area.

Slider Action Callbacks

Function Introduced

- **ProUISliderUpdateActionSet()**

Use the function **ProUISliderUpdateActionSet()** to set the update action for the slider.

Example 4: Source of Dialog with Slider and Linked InputPanel

This example shows code for use of a dialog with Slider and Linked InputPanel.

```
#include <ProToolkit.h>
#include <ProDimension.h>
#include <ProFeature.h>
#include <ProSelection.h>
#include <ProUICheckbutton.h>
#include <ProUIPushbutton.h>
#include <ProUIDialog.h>
#include <ProUIInputpanel.h>
#include <ProUILabel.h>
#include <ProUISlider.h>
#include <PTAppUnicodeUtils.h>
#include <ProUIOptionmenu.h>
#define OK 1
#define CANCEL 0

static ProName msgfil;
int UsrAngleGet();

/*=====*\
FUNCTION : UsrModAngle()
PURPOSE : Command to modify an angular dimension using a slider
\*=====*/
void UsrModAngle()
{
    ProError status;
    ProSelection *sel;
    int n_sel, angle;
    double value;
    ProName wname;
    ProCharName name;
    ProCharLine prompt;
    ProDimension dimension;
    ProDimensiontype dtype;

    while(1)
    {

/*-----*\
        Select a feature or a dimension
\*-----*/
        status = ProSelect("feature,dimension",1,
                           NULL,NULL,NULL,NULL,&sel,&n_sel);
        if(status != PRO_TK_NO_ERROR || n_sel < 1)
            break;
    }
}
```

```

        ProSelectionModelitemGet(sel[0], &dimension);

/*-----*\
    If it's a feature, display the dimensions
\*-----*/
    if(dimension.type == PRO_FEATURE)
        ProFeatureParamsDisplay(sel[0], PRO_DIM_PARAM);
    else
    {
/*-----*\
        Check that it's angular
\*-----*/
        ProDimensionTypeGet(&dimension, &dtype);
        if(dtype != PRODIMTYPE_ANGLE)
            continue;

/*-----*\
        Get the current value
\*-----*/
        ProDimensionValueGet(&dimension, &value);

        angle = (int)value;

/*-----*\
        Get the name and form a prompt string
\*-----*/
        ProDimensionSymbolGet(&dimension, wname);
        ProWstringToString(name, wname);
        ProTKSprintf(prompt, "Enter the new value of %s", name);

/*-----*\
        Input the new angle using a dialog box with a slider
\*-----*/
        if(!UsrAngleGet(prompt, &angle))
            continue;

/*-----*\
        Set the new value and update the dim display
\*-----*/
        value = (double)angle;
        if(ProDimensionValueSet(&dimension, value) != PRO_TK_NO_ERROR)
        {
            ProMessageDisplay(msgfil, "USER Could not modify dimension");
            continue;
        }

        ProDimensionDisplayUpdate(&dimension);
    }
}
}
/*=====*\

```

```

FUNCTION : UstrOKAction()
PURPOSE  : Action function for the OK button
\*=====*/
static void UstrSliderOKAction(
    char *dialog,
    char *component,
    ProAppData data)
{
    ProUIDialogExit(dialog, OK);
}
/*=====*\
FUNCTION : UstrCancelAction()
PURPOSE  : Action function for the CANCEL button
\*=====*/
static void UstrSliderCancelAction(
    char *dialog,
    char *component,
    ProAppData data)
{
    ProUIDialogExit(dialog, CANCEL);
}
/*=====*\
FUNCTION : UstrSliderAction() PURPOSE  : Callback for movement of the
slider
\*=====*/
void UstrSliderAction(
    char *dialog,
    char *component,
    ProAppData data)
{
    ProName wstr;
    ProCharName str;
    int *angle = (int*)data;
/*-----*\
    Get the new value
\*-----*/
    ProUISliderIntegerGet(dialog,component,angle);

/*-----*\
    Set the input panel to the same value
\*-----*/
    ProTKSprintf(str,"%d",*angle);
    ProStringToWstring(wstr, str);
    ProUIInputpanelValueSet(dialog,"InputPanel",wstr);
}
/*=====*\
FUNCTION : UstrInputpanelAction()
PURPOSE  : Callback for input to the input panel
\*=====*/
void UstrInputpanelAction(
    char *dialog,

```

```

        char *component,
        ProAppData data)
    {
        wchar_t *wstr;
        ProCharName str;
        int *angle = (int*)data;
/*-----*\
    Get the current value as an integer
\*-----*/
        ProUIInputpanelValueGet(dialog,component,&wstr);
        ProWstringToString(str, wstr);
        *angle = atoi(str);

/*-----*\
    Set the slider to the same value
\*-----*/
        ProUISliderIntegerSet(dialog,"Slider",*angle);
    }
/*=====*\
FUNCTION : UsrAngleGet()
PURPOSE : Input an angle using a dialog with a slider
\*=====*/
int UsrAngleGet(
    char *text, /* Text of the prompt displayed on the dialog */
    int *angle)
{
    int status;
    ProName wstr;
    ProCharName str;
    ProLine wline;
/*-----*\
    Load the dialog from the resource file
\*-----*/
        ProUIDialogCreate("angle","angle");

/*-----*\
    Set the prompt as the text of the Label component
\*-----*/
        ProStringToWstring(wline,text);
        ProUILabelTextSet("angle","Prompt",wline);

/*-----*\
    Set the OK and Cancel callbacks
\*-----*/
        ProUIPushbuttonActivateActionSet("angle","OK",
            UsrSliderOKAction,NULL);
        ProUIPushbuttonActivateActionSet("angle","Cancel",
            UsrSliderCancelAction,NULL);
/*-----*\
    Set the slider action and initial value
\*-----*/

```

```

        ProUISliderUpdateActionSet("angle","Slider",UsrSliderAction, angle);
        ProUISliderIntegerSet("angle","Slider",*angle);

/*-----*\
    Set the input panel action and initial value
\*-----*/
    ProUIInputpanelActivateActionSet("angle","InputPanel",
                                     UsrInputpanelAction,angle);

    ProTKSprintf(str,"%d",*angle);
    ProStringToWstring(wstr, str);
    ProUIInputpanelValueSet("angle","InputPanel",wstr);

/*-----*\
    Display and activate the dialog
\*-----*/
    ProUIDialogActivate("angle",&status);

/*-----*\
    Dispose of the dialog
\*-----*/
    ProUIDialogDestroy("angle");

    return(status==OK?1:0);
}

```

Spinbox

Spinbox Attributes

Attribute Name	Get Function	Set Function(s)
.AttachBottom	ProUISpinboxIsAttachedBottom()	ProUISpinboxAttachBottom() ProUISpinboxUnattachBottom
.AttachTop	ProUISpinboxIsAttachedTop()	ProUISpinboxAttachTop() ProUISpinboxUnattachTop()
.AttachRight	ProUISpinboxIsAttachedRight()	ProUISpinboxAttachRight() ProUISpinboxUnattachRight()
.AttachLeft	ProUISpinboxIsAttachedLeft()	ProUISpinboxAttachLeft() ProUISpinboxUnattachLeft()
.BottomOffset	ProUISpinboxBottomoffsetGet()	ProUISpinboxBottomoffsetSet()
.Digits	ProUISpinboxDigitsGet()	ProUISpinboxDigitsSet()
.Double	ProUISpinboxDoubleGet()	ProUISpinboxDoubleSet()
.DoubleFormat	ProUISpinboxDoubleformatGet()	ProUISpinboxDoubleformatSet()

Attribute Name	Get Function	Set Function(s)
.DoubleIncrement	ProUISpinboxDoubleincrementGet()	ProUISpinboxDoubleincrementSet()
.Editable	ProUISpinboxIsEditable()	ProUISpinboxEditable() ProUISpinboxReadOnly()
.FastDoubleIncrement	ProUISpinboxFastdoubleincrementGet()	ProUISpinboxFastdoubleincrementSet()
.FastIncrement	ProUISpinboxFastincrementGet()	ProUISpinboxFastincrementSet()
.HelpText	ProUISpinboxHelpTextGet()	ProUISpinboxHelpTextSet()
.Increment	ProUISpinboxIncrementGet()	ProUISpinboxIncrementSet()
.InputType	ProUISpinboxInputtypeGet()	ProUISpinboxInputtypeSet()
.Integer	ProUISpinboxIntegerGet()	ProUISpinboxIntegerSet()
.LeftOffset	ProUISpinboxLeftoffsetGet()	ProUISpinboxLeftoffsetSet()
.MaxDouble	ProUISpinboxMaxdoubleGet()	ProUISpinboxMaxdoubleSet()
.MaxInteger	ProUISpinboxMaxintegerGet()	ProUISpinboxMaxintegerSet()
.MinDouble	ProUISpinboxMindoubleGet()	ProUISpinboxMindoubleSet()
.MinInteger	ProUISpinboxMinintegerGet()	ProUISpinboxMinintegerSet()
.ParentName	ProUISpinboxParentnameGet()	Not Applicable
.PopupMenu	ProUISpinboxPopupMenuGet()	ProUISpinboxPopupMenuSet()
.RightOffset	ProUISpinboxRightoffsetGet()	ProUISpinboxRightoffsetSet()
.Sensitive	ProUISpinboxIsEnabled()	ProUISpinboxEnable() ProUISpinboxDisable()
.TopOffset	ProUISpinboxTopoffsetGet()	ProUISpinboxTopoffsetSet()
.Visible	ProUISpinboxIsVisible()	ProUISpinboxShow() ProUISpinboxHide()

Spinbox Action Callbacks

Functions Introduced:

- **ProUISpinboxUpdateActionSet()**
- **ProUISpinboxActivateActionSet()**

Use the function **ProUISpinboxUpdateActionSet()** and **ProUISpinboxActivateActionSet()** to set the update and activate action for a spin box respectively.

Spinbox Operations

Functions Introduced

- **ProUISpinboxAnchorSet()**
- **ProUISpinboxSizeSet()**
- **ProUISpinboxPositionSet()**
- **ProUISpinboxPositionGet()**
- **ProUISpinboxSizeGet()**

Use the function **ProUISpinboxAnchorSet()** to set the position of the spinbox with respect to a given anchor location. This function is applicable only if the parent of the Spinbox is a drawing area.

Use the function **ProUISpinboxSizeSet()** to set the size of the spinbox. This field is used only if the parent is a drawing area.

Use the function **ProUISpinboxPositionSet()** to set the position to the spinbox with respect to its parent. This field is used only if the parent is a drawing area.

Use the function **ProUISpinboxPositionGet()** to get the position to the spinbox with respect to its parent. This field is used only if the parent is a drawing area.

Use the function **ProUISpinboxSizeGet()** to get the size of the spinbox. This field is used only if the parent is a drawing area.

Tab

Tab Attributes

Attribute Name	Get Function	Set Function(s)
.AttachBottom	ProUITabIsAttachedBottom()	ProUITabAttachBottom() ProUITabUnattachBottom
.AttachTop	ProUITabIsAttachedTop()	ProUITabAttachTop() ProUITabUnattachTop()
.AttachRight	ProUITabIsAttachedRight()	ProUITabAttachRight() ProUITabUnattachRight()

Attribute Name	Get Function	Set Function(s)
.AttachLeft	ProUITabIsAttachedLeft()	ProUITabAttachLeft() ProUITabUnattachLeft()
.BottomOffset	ProUITabBottomoffsetGet()	ProUITabBottomoffsetSet()
.Decorated	ProUITabIsDecorated()	ProUITabDecorate() ProUITabUndecorate()
.HelpText	ProUITabHelpTextGet()	ProUITabHelpTextSet()
.ItemHelpText	ProUITabItemhelpTextGet()	ProUITabItemhelpTextSet()
.Label	ProUITabLabelsGet()	ProUITabLabelsSet()
.LeftOffset	ProUITabLeftoffsetGet()	ProUITabLeftoffsetSet()
.ParentName	ProUITabParentNameGet()	Not Applicable
.PopupMenu	ProUITabPopupMenuGet()	ProUITabPopupMenuSet()
.RightOffset	ProUITabRightoffsetGet()	ProUITabRightoffsetSet()
.SelectedNames	ProUITabSelectedNamesGet()	ProUITabSelectedNamesSet()
.Sensitive	ProUITabIsEnabled()	ProUITabEnable() ProUITabDisable()
.TopOffset	ProUITabTopoffsetGet()	ProUITabTopoffsetSet()
.Visible	ProUITabIsVisible()	ProUITabShow() ProUITabHide()

Tab Operations

Functions Introduced

- **ProUITabAnchorSet()**
- **ProUITabSizeSet()**
- **ProUITabPositionSet()**
- **ProUITabPositionGet()**
- **ProUITabSizeGet()**
- **ProUITabLayoutAdd()**
- **ProUITabLayoutsInsert()**
- **ProUITabItemNameSet()**
- **ProUITabItemLabelSet()**
- **ProUITabItemImageSet()**
- **ProUITabItemHelpTextStringSet()**
- **ProUITabItemLabelGet()**

- **ProUITabItemImageGet()**
- **ProUITabItemHelptextStringGet()**
- **ProUITabItemExtentsGet()**

Use the function **ProUITabAnchorSet()** to set the position of the Tab with respect to a given anchor location. This function is applicable only if the parent of the Tab is a drawing area.

Use the function **ProUITabSizeSet()** to set the size of the Tab. This field is used only if the parent is a drawing area.

Use the function **ProUITabPositionSet()** to set the position to the Tab with respect to its parent. This field is used only if the parent is a drawing area.

Use the function **ProUITabPositionGet()** to get the position to the Tab with respect to its parent. This field is used only if the parent is a drawing area.

Use the function **ProUITabSizeGet()** to get the size of the Tab. This field is used only if the parent is a drawing area.

Use the function **ProUITabLayoutAdd()** to add a new layout to the tab. Even if the layout has a label assigned, you must set the tab labels with **ProUITabLabelsSet()** for the decorated tab to show the appropriate labels.

Use the function **ProUITabLayoutsInsert()** to insert a new layout after an existing component in the tab. Even if the layouts have labels assigned, you must set the tab labels with **ProUITabLabelsSet()** for the decorated tab to show the appropriate labels.

Use the function **ProUITabItemNameSet()** to set a new name to the item in the tab.

Use the function **ProUITabItemLabelSet()** to set the label of the item in the tab.

Use the function **ProUITabItemImageSet()** to set the image of the item in the tab.

Use the function **ProUITabItemHelptextStringSet()** to set the text that should be displayed when the cursor is over the item in the tab.

Use the function **ProUITabItemLabelGet()** to get the label of the item in the tab.

Use the function **ProUITabItemImageGet()** to get the image of the item in the tab.

Use the function **ProUITabItemHelptextStringGet()** to get the text that is displayed when the cursor is over the item in the tab.

Use the function **ProUITabItemExtentsGet()** to get the boundary of the item that is in the tab relative to the top left corner of the dialog.

Tab Action Callbacks

Function Introduced

- **ProUITabSelectActionSet()**

Use the function **ProUITabSelectActionSet()** sets the select action for a tab. This function is called when the user selects the tab.

Table

Table Attributes

Attribute Name	Get Function	Set Function(s)
.ActivateOnReturn	ProUITableIsActivateonreturnEnabled()	ProUITableActivateonreturnEnable() ProUITableActivateonreturnDisable()
.Alignment	ProUITableAlignmentGet()	ProUITableAlignmentSet()
.AttachBottom	ProUITableIsAttachedBottom()	ProUITableAttachBottom() ProUITableUnattachBottom
.AttachTop	ProUITableIsAttachedTop()	ProUITableAttachTop() ProUITableUnattachTop()
.AttachRight	ProUITableIsAttachedRight()	ProUITableAttachRight() ProUITableUnattachRight()
.AttachLeft	ProUITableIsAttachedLeft()	ProUITableAttachLeft() ProUITableUnattachLeft()
.Autohighlight	ProUITableIsAutohighlightEnabled()	ProUITableAutohighlightEnable() ProUITableAutohighlightDisable()
.BottomOffset	ProUITableBottomoffsetGet()	ProUITableBottomoffsetSet()
.ChildNames	ProUITableChildnamesGet()	Not Applicable

Attribute Name	Get Function	Set Function(s)
.ColumnLabels	ProUITableColumnlabelsGet()	ProUITableColumnlabelsSet()
.ColumnNames	ProUITableColumnnamesGet()	ProUITableColumnnamesSet()
.ColumnWidths	ProUITableColumnwidthsGet()	ProUITableColumnwidthsSet()
.ColumnResizings	ProUITableColumnresizingsGet()	ProUITableColumnresizingsSet()
.ColumnSelectionPolicy	ProUITableColumnselectionpolicyGet()	ProUITableColumnselectionpolicySet()
.Columns	ProUITableColumnsGet()	ProUITableColumnsSet()
.DefaultColumnWidth	ProUITableDefaultcolumnwidthGet()	ProUITableDefaultcolumnwidthSet()
.HelpText	ProUITableHelpertextGet()	ProUITableHelpertextSet()
.LeftOffset	ProUITableLeftoffsetGet()	ProUITableLeftoffsetSet()
.LockedColumns	ProUITableLockedcolumnsGet()	ProUITableLockedcolumnsSet()
.LockedRows	ProUITableLockedrowsGet()	ProUITableLockedrowsSet()
.MinColumns	ProUITableMincolumnsGet()	ProUITableMincolumnsSet()
.MinRows	ProUITableMinrowsGet()	ProUITableMinrowsSet()
.ParentName	ProUITableParentnameGet()	Not Applicable
.PopupMenu	ProUITablePopupmenuGet()	ProUITablePopupmenuSet()
.RightOffset	ProUITableRightoffsetGet()	ProUITableRightoffsetSet()
.RowLabels	ProUITableRowlabelsGet()	ProUITableRowlabelsSet()
.RowNames	ProUITableRownamesGet()	ProUITableRownamesSet()
.ScrollBarsWhenNeeded	ProUITableUsesScrollbarswhenneeded()	ProUITableUseScrollbarswhenNeeded() ProUITableAlwaysUsesscrollBars()
.SelectedNames	ProUITableSelectednamesGet()	ProUITableSelectednamesSet()
.SelectedColumnName	ProUITableSelectedcolumnnamesGet()	ProUITableSelectedcolumnnamesSet()
.SelectedRowNames	ProUITableSelectedrownamesGet()	ProUITableSelectedrownamesSet()
.Sensitive	ProUITableIsEnabled()	ProUITableEnable() ProUITableDisable()
.SelectionPolicy	ProUITableSelectionpolicyGet()	ProUITableSelectionpolicySet()
.ShowGrid	ProUITableShowgridGet()	ProUITableShowgridSet()
.TopOffset	ProUITableTopoffsetGet()	ProUITableTopoffsetSet()
.TruncateLabel	ProUITableTruncatelabelGet()	ProUITableTruncatelabelSet()
.Visible	ProUITableIsVisible()	ProUITableShow() ProUITableHide()
.VisibleRows	ProUITableVisiblerowsGet()	ProUITableVisiblerowsSet()

Adding and Removing Components

Component Name	Adding Functions	Removing Functions
Checkbutton	ProUITableCheckbuttonAdd()	ProUITableCheckbuttonRemove()
Drawingarea	ProUITableDrawingareaAdd()	ProUITableDrawingareaRemove()
Inputpanel	ProUITableInputpanelAdd()	ProUITableInputpanelRemove()
Label	ProUITableLabelAdd()	ProUITableLabelRemove()
Layout	ProUITableLayoutAdd()	ProUITableLayoutRemove()
List	ProUITableListAdd()	ProUITableListRemove()
Optionmenu	ProUITableOptionmenuAdd()	ProUITableOptionmenuRemove()
Progressbar	ProUITableProgressbarAdd()	ProUITableProgressbarRemove()
Pushbutton	ProUITablePushbuttonAdd()	ProUITablePushbuttonRemove()
Radiogroup	ProUITableRadiogroupAdd()	ProUITableRadiogroupRemove()
Slider	ProUITableSliderAdd()	ProUITableSliderRemove()
Spinbox	ProUITableSpinboxAdd()	ProUITableSpinboxRemove()
Tab	ProUITableTabAdd()	ProUITableTabRemove()
Table	ProUITableTableAdd()	ProUITableTableRemove()
Textarea	ProUITableTextareaAdd()	ProUITableTextareaRemove()
Thumbwheel	ProUITableThumbwheelAdd()	ProUITableThumbwheelRemove()
Tree	ProUITableTreeAdd()	ProUITableTreeRemove()

Table Cell Functions

Functions Introduced

- **ProUITableCellLabelSet()**
- **ProUITableCellLabelGet()**
- **ProUITableCellHelpTextSet()**
- **ProUITableCellSensitive()**
- **ProUITableCellEnable()**
- **ProUITableCellDisable()**
- **ProUITableCellComponentNameSet()**
- **ProUITableCellComponentNameGet()**
- **ProUITableCellComponentCopy()**
- **ProUITableCellComponentDelete()**

- **ProUITableAnchorCellSet()**
- **ProUITableAnchorCellGet()**
- **ProUITableFocusCellSet()**
- **ProUITableFocusCellGet()**
- **ProUITableCellHelptextStringGet()**
- **ProUITableCellHelptextStringSet()**
- **ProUITableCellForegroundColorGet()**
- **ProUITableCellForegroundColorSet()**
- **ProUITableCellBackgroundColorGet()**
- **ProUITableCellBackgroundColorSet()**

Use the function **ProUITableCellLabelSet()** to set the text contained in the table cell.

Use the function **ProUITableCellLabelGet()** to get the text contained in the table cell.

Use the function **ProUITableCellHelpTextSet()** to set the help text for the table cell.

Use the function **ProUITableIsCellSensitive()** to determine if the table cell is sensitive to user input.

Use the function **ProUITableCellEnable()** to set the table cell to be sensitive to user input.

Use the function **ProUITableCellDisable()** to set the table cell to be insensitive to user input.

Use the function **ProUITableCellComponentNameSet()** to set the component name contained in the table cell.

Use the function **ProUITableCellComponentNameGet()** to get the component name contained in the table cell.

Use the function **ProUITableCellComponentCopy()** to copy a predefined component and places it in the table in the designated cell. The component will be displayed in this cell.

Use the function **ProUITableCellComponentDelete()** to remove the component contained in the table cell.

Use the function **ProUITableAnchorCellSet()** to set the coordinates of the table selection anchor cell.

Use the function **ProUITableAnchorCellGet()** to get the coordinates of the table selection anchor cell.

Use the function **ProUITableFocusCellSet()** to set the coordinates of the table selection focus cell.

Use the function **ProUITableFocusCellGet()** to get the coordinates of the table selection focus cell.

Use the function **ProUITableCellHelptextStringGet()** to get the help-text that is displayed while the cursor is over the table cell.

Use the function **ProUITableCellHelptextStringSet()** to set the help-text to be displayed when the cursor is over the table cell.

Use the function **ProUITableCellForegroundColorGet()** to get the foreground color of the table cell.

Use the function **ProUITableCellForegroundColorSet()** to set the foreground color of the table.

Use the function **ProUITableCellBackgroundColorGet()** to get the background color of the table cell.

Use the function **ProUITableCellBackgroundColorSet()** to set the background color of the table cell.

Example 5: To Assign Components into Table Cells

```
#include <ProToolkit.h>
#include <ProUI.h>
#include <ProUIDialog.h>
#include <ProUIPushbutton.h>
#include <ProUITable.h>

#define UGUITABLEDLG "uguitableexample"

/*=====*\
FUNCTION: UserTableDialogCloseAction
PURPOSE: Close action for "Close" button in table examples
\*=====*/
static void UserTableDialogCloseAction (char* dialog, char* button,
                                         ProAppData data)
{
    ProUIDialogExit (UGUITABLEDLG, PRO_TK_NO_ERROR);
}

/*=====*\
FUNCTION: UserUITableComponentExample
PURPOSE: Add components to the table cells
\*=====*/
int UserUITableComponentExample ()
```



```
{
    int status;

/*-----*\
    Load the dialog from the resource file
/*-----*/
    ProUIDialogCreate(UGUITABLEDLG, UGUITABLEDLG);

/*-----*\
    Set the Close button actions
/*-----*/
    ProUIPushbuttonActivateActionSet(UGUITABLEDLG, "CloseButton",
                                      UserTableDialogCloseAction, NULL);

/*-----*\
    Assign a predefined table components from the table layout
/*-----*/
    ProUITableCellComponentNameSet (UGUITABLEDLG, "LargeTable",
                                    "A", "1", "BaseButton");

/*-----*\
    Copy a table component into a table cell
/*-----*/
    ProUITableCellComponentCopy (UGUITABLEDLG, "LargeTable",
                                "C", "3",
                                UGUITABLEDLG, "BaseCheckButton",
                                "CheckButtonC3");

    ProUICheckbuttonSet (UGUITABLEDLG, "CheckButtonC3");

/*-----*\
    Copy a non-table component into a table & assign it to a table cell
/*-----*/
    ProUITableComponentCopy (UGUITABLEDLG, "LargeTable", UGUITABLEDLG,
                             "ToCopy", "CopyOfOptionsMenu");

    ProUITableCellComponentNameSet (UGUITABLEDLG, "LargeTable",
                                    "B", "2",
                                    "CopyOfOptionsMenu");

    ProUIOptionmenuShow (UGUITABLEDLG, "CopyOfOptionsMenu");

/*-----*\
    Copy a component from another dialog and assign it to a table cell
/*-----*/
    ProUIDialogCreate ("uguitablecomponents", "uguitablecomponents");

    ProUITableCellComponentCopy (UGUITABLEDLG, "LargeTable",
                                "D", "4",
                                "uguitablecomponents",
                                "ExternalButtonToCopy",
```

```

        "CopyOfExternalButton");

ProUIDialogDestroy ("uguitablecomponents");

/*-----*\
    Display and activate the dialog
\*-----*/
ProUIDialogActivate(UGUITABLEDLG, &status);

/*-----*\
    Remove the dialog from memory
\*-----*/
ProUIDialogDestroy(UGUITABLEDLG);

return(1);
}

```

Figure 16-6: UG Tables

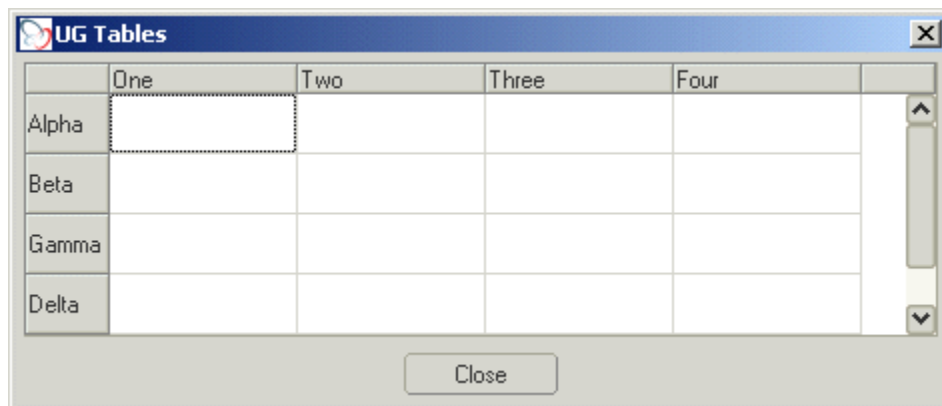


Table Row Functions

Functions Introduced

- **ProUITableRowIndexGet()**
- **ProUITableRowRename()**
- **ProUITableRowLabelSet()**
- **ProUITableRowLabelGet()**
- **ProUITableRowsInsert()**
- **ProUITableRowsDelete()**
- **ProUITableRowCellsSelect()**
- **ProUITableRowCellsDeselect()**
- **ProUITableRowCellLabelsSet()**

Use the function **ProUITableRowIndexGet()** to get the index of the table row with the given name.

Use the function **ProUITableRowRename()** to rename the table row.

Use the function **ProUITableRowLabelSet()** to set the user-visible label for the table row.

Use the function **ProUITableRowLabelGet()** to get the user-visible label for the table row.

Use the function **ProUITableRowsInsert()** to insert one or more new rows into the table.

Use the function **ProUITableRowsDelete()** to delete one or more rows from the table.

Use the function **ProUITableRowCellsSelect()** to select the cells of the given rows of the table. The table selection policy must be either Multiple or Extended.

Use the function **ProUITableRowCellsDeselect()** to deselect the cells of the given rows of the table.

Use the function **ProUITableRowCellLabelsSet()** to set the contents of the cells of the rows using a single string.

Table Column Functions

Functions Introduced

- **ProUITableColumnIndexGet()**
- **ProUITableColumnRename()**
- **ProUITableColumnLabelSet()**
- **ProUITableColumnLabelGet()**
- **ProUITableColumnWidthSet()**
- **ProUITableColumnWidthGet()**
- **ProUITableColumnResizingFactorSet()**
- **ProUITableColumnResizingFactorGet()**
- **ProUITableColumnsInsert()**
- **ProUITableColumnsDelete()**
- **ProUITableColumnCellsSelect()**

- **ProUITableColumnCellsDeselect()**
- **ProUITableResetColumnWidth()**

Use the function **ProUITableColumnIndexGet()** to get the column index for a given column.

Use the function **ProUITableColumnRename()** to rename a table column.

Use the function **ProUITableColumnLabelSet()** to set the user visible label for the column.

Use the function **ProUITableColumnLabelGet()** to get the user-visible label for the column.

Use the function **ProUITableColumnWidthSet()** to set the width of the column in the table.

Use the function **ProUITableColumnWidthGet()** to get the width of the column in the table.

Use the function **ProUITableColumnResizingFactorSet()** to set the resizing factor of the column in the table.

Use the function **ProUITableColumnResizingFactorGet()** to get the resizing factor of the column in the table.

Use the function **ProUITableColumnsInsert()** to insert one or more columns into the table.

Use the function **ProUITableColumnsDelete()** to delete one or more columns from the table.

Use the function **ProUITableColumnCellsSelect()** to select the cells of the given columns in the table.

Use the function **ProUITableColumnCellsDeselect()** to deselect the cells of the given columns in the table.

Use the function **ProUITableResetColumnWidth()** to set the column width to the default.

Example 6: To Access and Modify Names and Labels for the Table Rows and Columns

The following example shows how to access and modify names and labels for the Table rows and columns and also to access, allocate and free the string and wide string arrays.

```
#include <ProToolkit.h>
#include <ProUI.h>
#include <ProUIDialog.h>
#include <ProUIPushbutton.h>
#include <ProUITable.h>

#define UGUITABLEDLG "uguitableexample"

/*=====*\
FUNCTION: UserTableDialogCloseAction
PURPOSE: Close action for "Close" button in table examples
\*=====*/
static void UserTableDialogCloseAction (char* dialog, char* button,
                                         ProAppData data)
{
    ProUIDialogExit (UGUITABLEDLG, PRO_TK_NO_ERROR);
}

/*=====*\
FUNCTION: UserUITableRedesignExample
PURPOSE: Change the table columns and rows before displaying the dialog
\*=====*/
int UserUITableRedesignExample ()
{
    int status;
    int column_count;
    wchar_t** column_labels;
    char** column_names;

    int new_column_count = 3;
    char* new_column_names [] = {"Column A", "Column B", "Column C"};
    char_t* new_column_labels [] = {L"First", L"Second", L"Third"};

    /*-----*\
        Load the dialog from the resource file
    \*-----*/
    ProUIDialogCreate(UGUITABLEDLG, UGUITABLEDLG);

    /*-----*\
        Set the Close button actions
    \*-----*/
    ProUIPushbuttonActivateActionSet(UGUITABLEDLG, "CloseButton",
                                      UserTableDialogCloseAction, NULL);

    /*-----*\
        Get the list of column names and labels
    \*-----*/
    ProUITableColumnnamesGet (UGUITABLEDLG, "LargeTable", &column_count,
                              &column_names);
    ProUITableColumnlabelsGet (UGUITABLEDLG, "LargeTable", &column_count,
                              &column_labels);
}
```

```

/*-----*\
    Assign the column names to the rows
\*-----*/
ProUITableRownamesSet (UGUITABLEDLG, "LargeTable", column_count,
                      column_names);
ProUITableRowlabelsSet (UGUITABLEDLG, "LargeTable", column_count,
                      column_labels);

/*-----*\
    Free the retrieved arrays
\*-----*/
ProStringarrayFree (column_names, column_count);
ProWstringarrayFree (column_labels, column_count);

/*-----*\
    Modify the column names and labels
\*-----*/
ProUITableColumnnamesSet (UGUITABLEDLG, "LargeTable",
                        new_column_count,
                        new_column_names);
ProUITableColumnlabelsSet (UGUITABLEDLG, "LargeTable",
                        new_column_count,
                        new_column_labels);

/*-----*\
    Display and activate the dialog
\*-----*/
ProUIDialogActivate(UGUITABLEDLG, &status);

/*-----*\
    Remove the dialog from memory
\*-----*/
ProUIDialogDestroy(UGUITABLEDLG);

return(1);
}

```

Table Operations

Functions Introduced

- **ProUITableAnchorSet()**
- **ProUITableSizeSet()**
- **ProUITablePositionSet()**
- **ProUITablePositionGet()**
- **ProUITableSizeGet()**
- **ProUITableComponentCopy()**
- **ProUITableComponentDelete()**

Use the function **ProUITableAnchorSet()** to set the position of the table with respect to a given anchor location. This function is applicable only if the parent of the table is a drawing area.

Use the function **ProUITableSizeSet()** to set the size of the table. This field is used only if the parent is a drawing area.

Use the function **ProUITablePositionSet()** to set the position to the table with respect to its parent. This field is used only if the parent is a drawing area.

Use the function **ProUITablePositionGet()** to get the position to the table with respect to its parent. This field is used only if the parent is a drawing area.

Use the function **ProUITableSizeGet()** to get the size of the table. This field is used only if the parent is a drawing area.

Use the function **ProUITableComponentCopy()** to copy a predefined component and place it in the table. The component is not displayed until it is assigned to a table cell using **ProUITableCellComponentNameSet()**. However, you can update the component's properties as needed and display it at a later time.

Use the function **ProUITableComponentDelete()** to delete a specified component from the table.

Table Action Callbacks

Functions Introduced

- **ProUITableArmActionSet()**
- **ProUITableDisarmActionSet()**
- **ProUITableSelectActionSet()**
- **ProUITableActivateActionSet()**
- **ProUITableFocusinActionSet()**
- **ProUITableFocusoutActionSet()**
- **ProUITableColumnselectActionSet()**

Use the function **ProUITableArmActionSet()** to set the arm action for a table. This function is called when the user changes the selection anchor cell and focus cell in the table.

Use the function **ProUITableDisarmActionSet()** to set the disarm action for a table. This function is called when the user changes the selection focus cell in the table.

Use the function **ProUITableSelectActionSet()** to set the select action for a table. This function is called when the user changes the selected cells in the table.

Use the function **ProUITableActivateActionSet()** to set the activate action for a table. This function is called when the user presses the return key or double-clicks the left mouse button in the table.

Use the function **ProUITableFocusinActionSet()** to set the focus in action for a table.

Use the function **ProUITableFocusoutActionSet()** to set the focus out action for a table.

Use the function **ProUITableColumnselectActionSet()** to set the column selection action for the table. This function is called when the user changes the currently selected table columns.

Example 7: To Access Selected Names Array from Tables

```
<ProToolkit.h>
#include <ProUI.h>
#include <ProUIDialog.h>
#include <ProUIPushbutton.h>
#include <ProUITable.h>

#define UGUITABLEDLG "uguitableexample"

/*=====*\
FUNCTION: UserTableDialogCloseAction
PURPOSE: Close action for "Close" button in table examples
\*=====*/
static void UserTableDialogCloseAction (char* dialog, char* button,
                                         ProAppData data)
{
    ProUIDialogExit (UGUITABLEDLG, PRO_TK_NO_ERROR);
}

/*=====*\
FUNCTION: UserTableDialogSelectionsToggle
PURPOSE: UI callback function to toggle contents of table cells
\*=====*/
static void UserTableDialogSelectionsToggle (char* dialog, char* button,
                                              ProAppData data)
{
    char** selections;
    int selection_count;
    wchar_t* label;
    int i;

    ProUITableSelectednamesGet (UGUITABLEDLG, "LargeTable",
                                &selection_count, &selections);

    /*-----*\
       Selected names are arranged in sets of 2 ("Row", "Column")
    \*-----*/
    for (i = 0; i < selection_count; i = i + 2)
    {
        ProUITableCellLabelGet (UGUITABLEDLG, "LargeTable",
                                selections [i],
                                selections [i+1], &label);

        if (label != NULL && ProUtilWstrcmp (label, L"ON") == 0)
        {
            ProUITableCellLabelSet (UGUITABLEDLG, "LargeTable",
                                    selections [i],
                                    selections [i+1], L"OFF");
        }
        else
    }
```

```

{
    ProUITableCellLabelSet (UGUITABLEDLG, "LargeTable",
        selections [i],
        selections [i+1], L"ON");
}
if (label != NULL)

    ProWstringFree (label);
{
    ProStringarrayFree (selections, selection_count);
}

/*=====*\
FUNCTION: UserUITableSelectionExample
PURPOSE:  Change the table columns and rows before displaying the dialog
\*=====*/
int UserUITableSelectionExample ()
{
    int status;

/*-----*\
    Load the dialog from the resource file
\*-----*/
ProUIDialogCreate(UGUITABLEDLG, UGUITABLEDLG);

/*-----*\
    Set the Close button actions
\*-----*/
ProUIPushbuttonActivateActionSet(UGUITABLEDLG, "CloseButton",
                                UserTableDialogCloseAction,
                                NULL);

/*-----*\
    Set up the Modify Select button
\*-----*/
ProUIPushbuttonShow (UGUITABLEDLG, "ModifySelectButton");
ProUIPushbuttonActivateActionSet (UGUITABLEDLG, "ModifySelectButton",
    UserTableDialogSelectionsToggle,
    NULL);

/*-----*\
    Set the selection policy to multiple
\*-----*/
ProUITableSelectionpolicySet (UGUITABLEDLG, "LargeTable",
    PROUISELPOLICY_MULTIPLE);

/*-----*\
    Display and activate the dialog
\*-----*/
ProUIDialogActivate(UGUITABLEDLG, &status);

```

```

/*-----*
    Remove the dialog from memory
/*-----*/
ProUIDialogDestroy(UGUITABLEDLG);

return(1);
}

```

Textarea

Textarea Attributes

Attribute Name	Get Function	Set Function(s)
.AttachBottom	ProUITextarealsAttachedBottom()	ProUITextareaAttachBottom() ProUITextareaUnattachBottom()
.AttachTop	ProUITextarealsAttachedTop()	ProUITextareaAttachTop() ProUITextareaUnattachTop()
.AttachRight	ProUITextarealsAttachedRight()	ProUITextareaAttachRight() ProUITextareaUnattachRight()
.AttachLeft	ProUITextarealsAttachedLeft()	ProUITextareaAttachLeft() ProUITextareaUnattachLeft()
.BottomOffset	ProUITextareaBottomoffsetGet()	ProUITextareaBottomoffsetSet()
.Columns	ProUITextareaColumnsGet()	ProUITextareaColumnsSet()
.Editable	ProUITextarealsEditable()	ProUITextareaEditable() ProUITextareaReadOnly()
.HelpText	ProUITextareaHelpTextGet()	ProUITextareaHelpTextSet()
.LeftOffset	ProUITextareaLeftoffsetGet()	ProUITextareaLeftoffsetSet()
.MinRows	ProUITextareaMinrowsGet()	ProUITextareaMinrowsSet()
.MaxLen	ProUITextareaMaxlenGet()	ProUITextareaMaxlenSet()
.ParentName	ProUITextareaParentnameGet()	Not Applicable
.PopupMenu	ProUITextareaPopupmenuGet()	ProUITextareaPopupmenuSet()
.RightOffset	ProUITextareaRightoffsetGet()	ProUITextareaRightoffsetSet()
.Rows	ProUITextareaRowsGet()	ProUITextareaRowsSet()
.Sensitive	ProUITextarealsEnabled()	ProUITextareaEnable() ProUITextareaDisable()
.TopOffset	ProUITextareaTopoffsetGet()	ProUITextareaTopoffsetSet()
.Value	ProUITextareaValueGet()	ProUITextareaValueSet()

Attribute Name	Get Function	Set Function(s)
.Visible	ProUITextarealsVisible()	ProUITextareaShow() ProUITextareaHide()

Textarea Operations

Functions Introduced

- **ProUITextareaAnchorSet()**
- **ProUITextareaSizeSet()**
- **ProUITextareaPositionGet()**
- **ProUITextareaPositionSet()**
- **ProUITextareaSizeGet()**

Use the function **ProUITextareaAnchorSet()** to set the position of the Textarea with respect to a given anchor location. This function is applicable only if the parent of the Textarea is a drawing area.

Use the function **ProUITextareaSizeSet()** to set the size of the Textarea. This operation is applicable only if the parent is a drawing area.

Use the function **ProUITextareaPositionGet()** to get the position of the Textarea with respect to its parent. This operation is applicable only if the parent is a drawing area.

Use the function **ProUITextareaPositionSet()** to set the position to the Textarea with respect to its parent. This operation is applicable only if the parent is a drawing area.

Use the function **ProUITextareaSizeGet()** to get the size of the Textarea. This operation is applicable only if the parent is a drawing area.

Textarea Action Callbacks

Functions Introduced

- **ProUITextareaActivateActionSet()**
- **ProUITextareaFocusinActionSet()**
- **ProUITextareaFocusoutActionSet()**
- **ProUITextarealInputActionSet()**

Use the function **ProUITextareaActivateActionSet()** to set the action callback to be called when the user hits return in an Textarea.

Use the function **ProUITextareaFocusinActionSet()** to set the focus in action for an Textarea. This function is called when the user moves the cursor onto the Textarea using the mouse or <TAB> key.

Use the function **ProUITextareaFocusoutActionSet()** to set the focus out action for an Textarea. This function is called when the user moves the cursor off of the Textarea using the mouse or <TAB> key.

Use the function **ProUITextareaInputActionSet()** to set the action callback to be called when the user enters a key in an Textarea.

Thumbwheel

Thumbwheel Attributes

Attribute Name	Get Function	Set Function(s)
.AttachBottom	ProUIThumbwheelsAttachedBottom()	ProUIThumbwheelAttachBottom() ProUIThumbwheelUnattachBottom
.AttachTop	ProUIThumbwheelsAttachedTop()	ProUIThumbwheelAttachTop() ProUIThumbwheelUnattachTop()
.AttachRight	ProUIThumbwheelsAttachedRight()	ProUIThumbwheelAttachRight() ProUIThumbwheelUnattachRight()
.AttachLeft	ProUIThumbwheelsAttachedLeft()	ProUIThumbwheelAttachLeft() ProUIThumbwheelUnattachLeft()
.BottomOffset	ProUIThumbwheelBottomoffsetGet()	ProUIThumbwheelBottomoffsetSet()
.HelpText	ProUIThumbwheelHelpTextGet()	ProUIThumbwheelHelpTextSet()
.Integer	ProUIThumbwheelIntegerGet()	ProUIThumbwheelIntegerSet()
.LeftOffset	ProUIThumbwheelLeftoffsetGet()	ProUIThumbwheelLeftoffsetSet()
.MaxInteger	ProUIThumbwheelMaxintegerGet()	ProUIThumbwheelMaxintegerSet()
.MinInteger	ProUIThumbwheelMinintegerGet()	ProUIThumbwheelMinintegerSet()

Attribute Name	Get Function	Set Function(s)
.ParentName	ProUIThumbwheelParentnameGet()	Not Applicable
.PopupMenu	ProUIThumbwheelPopupmenuGet()	ProUIThumbwheelPopupmenuSet()
.RightOffset	ProUIThumbwheelRightoffsetGet()	ProUIThumbwheelRightoffsetSet()
.Sensitive	ProUIThumbwheelIsEnabled()	ProUIThumbwheelEnable() ProUIThumbwheelDisable()
.TopOffset	ProUIThumbwheelTopoffsetGet()	ProUIThumbwheelTopoffsetSet()
.UnitsPerRotation	ProUIThumbwheelUnitsperrotationGet()	ProUIThumbwheelUnitsperrotationSet()
.Visible	ProUIThumbwheelIsVisible()	ProUIThumbwheelShow() ProUIThumbwheelHide()

Thumbwheel Operations

Functions Introduced

- **ProUIThumbwheelAnchorSet()**
- **ProUIThumbwheelSizeSet()**
- **ProUIThumbwheelPositionSet()**
- **ProUIThumbwheelPositionGet()**
- **ProUIThumbwheelSizeGet()**

Use the function **ProUIThumbwheelAnchorSet()** to set the position of the Thumbwheel with respect to a given anchor location. This function is applicable only if the parent of the Thumbwheel is a drawing area.

Use the function **ProUIThumbwheelSizeSet()** to set the size of the Thumbwheel. This field is used only if the parent is a drawing area.

Use the function **ProUIThumbwheelPositionSet()** to set the position to the Thumbwheel with respect to its parent. This field is used only if the parent is a drawing area.

Use the function **ProUIThumbwheelPositionGet()** to get the position to the Thumbwheel with respect to its parent. This field is used only if the parent is a drawing area.

Use the function **ProUIThumbwheelSizeGet()** to get the size of the Thumbwheel. This field is used only if the parent is a drawing area.

Thumbwheel Action Callbacks

Functions Introduced

- **ProUIThumbwheelUpdateActionSet()**

Use the function **ProUIThumbwheelUpdateActionSet()** to set the update action for the Thumbwheel.

Tree

Tree Attributes

Attribute Name	Get Function	Set Function(s)
.ActivateOnReturn	ProUITreelsActivateonreturnEnabled()	ProUITreeActivateonreturnEnable() ProUITreeActivateonreturnDisable()
.AttachBottom	ProUITreelsAttachedBottom()	ProUITreeAttachBottom() ProUITreeUnattachBottom()
.AttachLeft	ProUITreelsAttachedLeft()	ProUITreeAttachLeft() ProUITreeUnattachLeft()
.AttachRight	ProUITreelsAttachedRight()	ProUITreeAttachRight() ProUITreeUnattachRight()
.AttachTop	ProUITreelsAttachedTop()	ProUITreeAttachTop() ProUITreeUnattachTop()
.BackgroundColor	ProUITreeBackgroundcolorGet()	ProUITreeBackgroundcolorSet()
.BottomOffset	ProUITreeBottomoffsetGet()	ProUITreeBottomoffsetSet()
.LeftOffset	ProUITreeLeftoffsetGet()	ProUITreeLeftoffsetSet()
.RightOffset	ProUITreeRightoffsetGet()	ProUITreeRightoffsetSet()
.TopOffset	ProUITreeTopoffsetGet()	ProUITreeTopoffsetSet()
.Columns	Not Applicable	Not Applicable
.ForegroundColor	ProUITreeForegroundColorGet()	ProUITreeForegroundColorSet()
.HelpText	ProUITreeHelptextGet()	ProUITreeHelptextSet()
.Label	Not Applicable	Not Applicable
.LabelAlignment	Not Applicable	Not Applicable
.Mapped	ProUITreelsMapped()	ProUITreeMappedSet() ProUITreeMappedUnset()
.MinColumns	Not Applicable	Not Applicable
.MinRows	Not Applicable	Not Applicable

Attribute Name	Get Function	Set Function(s)
.ParentName	ProUITreeParentnameGet()	Not Applicable
.PopupMenu	ProUITreePopupmenuGet()	ProUITreePopupmenuSet()
.PopupWhenInsen	ProUITreePopupwheninsensitiveEnabled()	ProUITreeEnablePopupwheninsensitive() ProUITreeDisablePopupwheninsensitive()
.ResizableCols	Not Applicable	Not Applicable
.Rows	ProUITreeRowsGet()	ProUITreeRowsSet()
.ScrollBarsWhenNeeded	ProUITreeUsesScrollbarswhenneeded()	ProUITreeUseScrollbarswhenNeeded() ProUITreeAlwaysUsesscrollBars()
.SelectedNames	ProUITreeSelectednamesGet()	ProUITreeSelectednamesSet()
.SelectionPolicy	ProUITreeSelectionpolicyGet()	ProUITreeSelectionpolicySet()
.SelectByCell	Not Applicable	Not Applicable
.SelectCBRegardless	ProUITreeSelectcbregardlessGet()	ProUITreeSelectcbregardlessSet()
.Sensitive	ProUITreeEnabled()	ProUITreeEnable() ProUITreeDisable()
.TreeAttributeWindow	Not Applicable	Not Applicable
.TreeBoxNodes	Not Applicable	Not Applicable
.TreeCellInput	ProUITreeTreecellinputGet()	Not Applicable
.TreeCellSelCol	ProUITreeTreecellselcolGet()	ProUITreeTreecellselcolSet()
.TreeCellSelNode	ProUITreeTreecellselnodeGet()	ProUITreeTreecellselnodeSet()
.TreeChildOrder	Not Applicable	Not Applicable
.TreeColumnsJusts	Not Applicable	Not Applicable
.TreeColumnsNames	Not Applicable	Not Applicable
.TreeColumnsTitles	Not Applicable	Not Applicable
.TreeColumnsWidths	Not Applicable	Not Applicable
.TreeColumnOrder	ProUITreeTreecolumnorderGet()	ProUITreeTreecolumnorderSet()
.TreeCurrent	ProUITreeTreecurrentnodeGet()	ProUITreeTreecurrentnodeSet()
.TreeDisplayRoot	ProUITreeRootnodeVisible()	ProUITreeShowRootnode() ProUITreeHideRootnode()
.TreeExpColNode	ProUITreeTreeexpcolnodeGet()	Not Applicable
.TreeIndicateChildren	Not Applicable	Not Applicable
.TreeInitialSashPercent	Not Applicable	Not Applicable
.TreeKeyboardInput	ProUITreeCellkeyboardinputEnabled()	ProUITreeEnableCellkeyboardinput() ProUITreeDisableCellkeyboardinput()

Attribute Name	Get Function	Set Function(s)
.TreeLinkStyle	Not Applicable	Not Applicable
.TreeNodeTypeAppends	ProUITreeTreenodetypeappendsGet()	ProUITreeTreenodetypeappendsSet()
.TreeNodeTypeCollapsedImages	ProUITreeTreenodetypecollapsedimagesGet()	ProUITreeTreenodetypecollapsedimagesSet()
.TreeNodeTypeExpandedImages	ProUITreeTreenodetypeexpandedimagesGet()	ProUITreeTreenodetypeexpandedimagesSet()
.TreeNodeTypeHelpTexts	ProUITreeTreenodetypehelptextsGet()	ProUITreeTreenodetypehelptextsSet()
.TreeNodeTypePrefixs	ProUITreeTreenodetypeprefixesGet()	ProUITreeTreenodetypeprefixesSet()
.TreeNodeTypeNames	ProUITreeTreenodetypenamesGet()	ProUITreeTreenodetypenamesSet()
.TreeRedraw	ProUITreeTreeredrawGet()	ProUITreeTreeredrawSet()
.TreeRootNode	ProUITreeTreerootnodeGet()	ProUITreeTreerootnodeSet()
.TreeVerticalSBPosition	Not Applicable	Not Applicable
.Visible	ProUITreeIsVisible()	ProUITreeShow() ProUITreeHide()

Note: Many of the properties of trees are currently only supported as a part of the resource file. Only very basic trees can be created using programmatic means.

Adding and Removing Components

Component Name	Adding Functions	Removing Functions
Checkbox	ProUITreeCheckboxAdd()	ProUITreeCheckboxRemove()
Drawingarea	ProUITreeDrawingareaAdd()	ProUITreeDrawingareaRemove()
Inputpanel	ProUITreeInputpanelAdd()	ProUITreeInputpanelRemove()
Label	ProUITreeLabelAdd()	ProUITreeLabelRemove()
Layout	ProUITreeLayoutAdd()	ProUITreeLayoutRemove()
List	ProUITreeListAdd()	ProUITreeListRemove()
Optionmenu	ProUITreeOptionmenuAdd()	ProUITreeOptionmenuRemove()
Progressbar	ProUITreeProgressbarAdd()	ProUITreeProgressbarRemove()
Pushbutton	ProUITreePushbuttonAdd()	ProUITreePushbuttonRemove()
Radiogroup	ProUITreeRadiogroupAdd()	ProUITreeRadiogroupRemove()
Slider	ProUITreeSliderAdd()	ProUITreeSliderRemove()

Component Name	Adding Functions	Removing Functions
Spinbox	ProUITreeSpinboxAdd()	ProUITreeSpinboxRemove()
Tab	ProUITreeTabAdd()	ProUITreeTabRemove()
Table	ProUITreeTableAdd()	ProUITreeTableRemove()
Tree	ProUITreeTreeAdd()	ProUITreeTreeRemove()
Textarea	ProUITreeTextareaAdd()	ProUITreeTextareaRemove()
Thumbwheel	ProUITreeThumbwheelAdd()	ProUITreeThumbwheelRemove()

Tree Column Functions

Functions Introduced

- **ProUITreeColumnCreate()**
- **ProUITreeColumnTitleGet()**
- **ProUITreeColumnWidthGet()**
- **ProUITreeColumnVisibilityGet()**
- **ProUITreeColumnJustificationGet()**
- **ProUITreeColumnTitleSet()**
- **ProUITreeColumnWidthSet()**
- **ProUITreeColumnVisibilitySet()**
- **ProUITreeColumnJustificationSet()**

Use the function **ProUITreeColumnCreate()** to create a column of the given name in the attribute window of the tree. The name must be unique within the scope of the columns of the tree.

Use the function **ProUITreeColumnTitleGet()** to obtain the title of a column in the tree.

Use the function **ProUITreeColumnWidthGet()** to obtain the width of a column in the tree.

Use the function **ProUITreeColumnVisibilityGet()** to obtain the visibility of a column in the tree.

Use the function **ProUITreeColumnJustificationGet()** to obtain the justification of a column in the tree.

Use the function **ProUITreeColumnTitleSet()** to set a title to the column in the tree.

Use the function **ProUITreeColumnWidthSet()** to set a width to the column in the tree.

Use the function **ProUITreeColumnVisibilitySet()** to set visibility of the column in the tree.

Use the function **ProUITreeColumnJustificationSet()** to set justification to the column in the tree.

Tree Node Functions

Functions Introduced

- **ProUITreeNodeComponentSet()**
- **ProUITreeNodeChildrenGet()**
- **ProUITreeNodeLabelGet()**
- **ProUITreeNodeLabelSet()**
- **ProUITreeNodeParentGet()**
- **ProUITreeNodeTypeGet()**
- **ProUITreeNodeTypeSet()**
- **ProUITreeNodeExtentsGet()**
- **ProUITreeNodesOfTypeGet()**
- **ProUITreeNodeAdd()**
- **ProUITreeNodeInsert()**
- **ProUITreeNodeCollapse()**
- **ProUITreeNodeDelete()**
- **ProUITreeNodeExists()**
- **ProUITreeNodeExpand()**
- **ProUITreeNodeHelptextGet()**
- **ProUITreeNodesVisible()**
- **ProUITreeNodesExpanded()**
- **ProUITreeNodesSelected()**
- **ProUITreeNodeFontstyleGet()**
- **ProUITreeNodeHelptextSet()**
- **ProUITreeNodeShow()**
- **ProUITreeNodeHide()**
- **ProUITreeNodeSelect()**
- **ProUITreeNodeDeselect()**

- **ProUITreeNodeFontstyleSet()**
- **ProUITreeNodeMove()**
- **ProUITreeNodeRename()**
- **ProUITreeNodeTypeAdd()**
- **ProUITreeNodeTypeDelete()**
- **ProUITreeNodeColumnTextSet()**
- **ProUITreeNodesSelect()**
- **ProUITreeNodeTypeSelect()**
- **ProUITreeAllnodesSelect()**
- **ProUITreeNodesDeselect()**
- **ProUITreeNodeTypeDeselect()**
- **ProUITreeAllnodesDeselect()**
- **ProUITreeNodeVisitAction()**
- **ProUITreeNodeFilterAction()**
- **ProUITreeNodesVisit()**

Use the function **ProUITreeNodeComponentSet()** to set the component to be displayed in the given column corresponding to the given node name of the tree.

Use the function **ProUITreeNodeChildrenGet()** to get the child nodes for the node in the tree.

Use the function **ProUITreeNodeLabelGet()** to get the label of the node in the tree.

Use the function **ProUITreeNodeLabelSet()** to set the label of the node in the tree.

Use the function **ProUITreeNodeParentGet()** to get the parent of the node in the tree.

Use the function **ProUITreeNodeTypeGet()** to get the name of the node type associated with the node.

Use the function **ProUITreeNodeTypeSet()** to set the name of the node type associated with the node.

Use the function **ProUITreeNodeExtentsGet()** to get the boundary of the node of the tree relative to the top-left corner of the dialog.

Use the function **ProUITreeNodesOfTypeGet()** to get the name of all nodes of the tree associated with the node type.

Use the function **ProUITreeNodeAdd()** to add a new node to the tree.

Use the function **ProUITreeNodeInsert()** to insert a new node before a node in the tree.

Use the function **ProUITreeNodeCollapse()** to collapse a node in the tree making its children invisible.

Use the function **ProUITreeNodeDelete()** to delete the node in the tree.

Use the function **ProUITreeNodeExists()** to checks if the given node exists or not.

Use the function **ProUITreeNodeExpand()** to expand the node of the tree making all of its children visible.

Use the function **ProUITreeNodeHelptextGet()** to get the help text of the node in the tree.

Use the function **ProUITreeNodeIsVisible()** to checks if the node is shown or hidden.

Use the function **ProUITreeNodeIsExpanded()** to checks if the node is expanded or collapsed.

Use the function **ProUITreeNodeIsSelected()** to checks if the node is selected or not.

Use the function **ProUITreeNodeFontstyleGet()** to get the fontstyle of the node.

Use the function **ProUITreeNodeHelptextSet()** to set the helptext of the node in tree.

Use the function **ProUITreeNodeShow()** to display the node in a tree.

Use the function **ProUITreeNodeHide()** to hide the node in the tree.

Use the function **ProUITreeNodeSelect()** to select the node in the tree.

Use the function **ProUITreeNodeDeselect()** to unselect the node in the tree.

Use the function **ProUITreeNodeFontstyleSet()** to set fontstyle to the node in the tree.

Use the function **ProUITreeNodeMove()** to move the node in the tree to a child of the given parent node.

Use the function **ProUITreeNodeRename()** to rename the given node of the tree using the given name.

Use the function **ProUITreeNodeTypeAdd()** to add a new node type to the tree using the information supplied in the given data structure. Use **ProUITreeNodeTypeAlloc()** to fill the **ProUITreeNodeType()**.

Use the function **ProUITreeNodeTypeDelete()** to delete a node type from the tree using the information supplied in the given data structure.

Use the function **ProUITreeNodeColumnTextSet()** to set the given text to be displayed in the given column corresponding to the given node of the tree.

Use the function **ProUITreeNodesSelect()** to select an array of nodes in the tree.

Use the function **ProUITreeNodeTypeSelect()** to select all the nodes corresponding to a given node type.

Use the function **ProUITreeAllnodesSelect()** to select all nodes in the tree.

Use the function **ProUITreeNodesDeselect()** to unselect an array of nodes in the tree.

Use the function **ProUITreeNodeTypeDeselect()** to unselect all the nodes corresponding to a given node type.

Use the function **ProUITreeAllnodesDeselect()** to unselect all the nodes in the tree.

Use the function **ProUITreeParentnameGet()** to get the name of the parent to the tree component.

Use the function **ProUITreeChildnamesGet()** to get the name of the children to the tree component.

Use the function **ProUITreeNodeVisitAction()** for visiting nodes.

Use the function **ProUITreeNodeFilterAction()** for filtering nodes.

Use the function **ProUITreeNodesVisit()** to visits all the descendent nodes of the given node in the tree.

Tree NodeType Functions

Functions Introduced

- **ProUITreeNodeTypeAlloc()**
- **ProUITreeNodeTypeFree()**
- **ProUITreeNodeTypeExpandImageSet()**
- **ProUITreeNodeTypeExpandImageGet()**
- **ProUITreeNodeTypeCollapseImageSet()**
- **ProUITreeNodeTypeCollapseImageGet()**
- **ProUITreeNodeTypePrefixSet()**
- **ProUITreeNodeTypePrefixGet()**
- **ProUITreeNodeTypeAppendStringSet()**
- **ProUITreeNodeTypeAppendStringGet()**
- **ProUITreeNodeTypeNodesSet()**
- **ProUITreeNodeTypeNodesGet()**

Use the function **ProUITreeNodeTypeAlloc()** to allocate *ProUITreeNodeType* data.

Use the function **ProUITreeNodeTypeFree()** to free *ProUITreeNodeType* data.

Use the function **ProUITreeNodeTypeExpandImageSet()** to set the image to appear when the nodetype is expanded.

Use the function **ProUITreeNodeTypeExpandImageGet()** to get the image that appears when the nodetype is expanded.

Use the function **ProUITreeNodeTypeCollapseImageSet()** to set the image to appear when the nodetype is collapsed.

Use the function **ProUITreeNodeTypeCollapseImageGet()** to get the image that appears when the nodetype is collapsed.

Use the function **ProUITreeNodeTypePrefixSet()** to set prefix to the nodetype.

Use the function **ProUITreeNodeTypePrefixGet()** to get prefix of the nodetype.

Use the function **ProUITreeNodeTypeAppendStringSet()** to append the string to the nodetype.

Use the function **ProUITreeNodeTypeAppendStringGet()** to get the appended string to the nodetype.

Use the function **ProUITreeNodeTypeNodesSet()** to set the nodes to the nodetype.

Use the function **ProUITreeNodeTypeNodesGet()** to get the nodes to the nodetype.

Tree Operations

Functions Introduced

- **ProUITreeAnchorSet()**
- **ProUITreeSizeSet()**
- **ProUITreePositionSet()**
- **ProUITreePositionGet()**
- **ProUITreeSizeGet()**
- **ProUITreeSashPositionGet()**

Use the function **ProUITreeAnchorSet()** to set the position of the Tree with respect to a given anchor location. This function is applicable only if the parent of the Tree is a drawing area.

Use the function **ProUITreeSizeSet()** to set the size of the Tree. This field is used only if the parent is a drawing area.

Use the function **ProUITreePositionSet()** to set the position to the Tree with respect to its parent. This field is used only if the parent is a drawing area.

Use the function **ProUITreePositionGet()** to get the position to the Tree with respect to its parent. This field is used only if the parent is a drawing area.

Use the function **ProUITreeSizeGet()** to get the size of the Tree. This field is used only if the parent is a drawing area.

Use the function **ProUITreeSashPositionGet()** to get the position of the sash between the tree hierarchy and the attribute window.

Tree Action Callbacks

Functions Introduced

- **ProUITreeSelectActionSet()**
- **ProUITreeActivateActionSet()**
- **ProUITreeFocusinActionSet()**
- **ProUITreeFocusoutActionSet()**
- **ProUITreeExpandActionSet()**
- **ProUITreeCollapseActionSet()**
- **ProUITreeTreecellselectActionSet()**
- **ProUITreeTreecellactivateActionSet()**
- **ProUITreeTreecellinputActionSet()**
- **ProUITreeTreecelldeleteActionSet()**
- **ProUITreeMoveActionSet()**

Use the function **ProUITreeSelectActionSet()** to set the select action for a Tree. This function is called when the user changes the selected cells in the Tree.

Use the function **ProUITreeActivateActionSet()** to set the activate action for a Tree. This function is called when the user presses the return key or double-clicks the left mouse button in the Tree.

Use the function **ProUITreeFocusinActionSet()** to set the focus in action for a Tree.

Use the function **ProUITreeFocusoutActionSet()** to set the focus out action for a Tree.

Use the function **ProUITreeSelectActionSet()** to set the action function to be called when the tree is selected. The left mouse button, the SPACE key and the navigation keys (if the selection policy is PROUISELPOLICY_BROWSE or PROUISELPOLICY_EXTENDED) can be used to make a selection in the tree.

Use the function **ProUITreeActivateActionSet()** to set the action function to be called when the tree is activated. The tree is activated by the press of RETURN key or the double click of LEFT mouse button over the node in the tree.

Use the function **ProUITreeExpandActionSet()** to set the action function to be called when the user attempts to expand a tree node by clicking on the '+' sign.

Note: You must make this callback call **ProUITreeNodeExpand()** to actually expand the node.

Use the function **ProUITreeCollapseActionSet()** to set the action function to be called when the user attempts to collapse a tree node by clicking on the '-' sign.

Note: You must make this callback call **ProUITreeNodeCollapse()** to actually collapse the node.

Use the function **ProUITreeFocusinActionSet()** to set the action function to be called when the tree has got keyboard input focus.

Use the function **ProUITreeFocusoutActionSet()** to set the action function to be called when the tree has lost keyboard input focus.

Use the function **ProUITreeTreecellselectActionSet()** to set the action function to be called when the cell of the attribute window of the tree has been selected.

Use the function **ProUITreeTreecellactivateActionSet()** to set the action function to be called when the cell of the attribute window of the tree has been activated.

Use the function **ProUITreeTreecellinputActionSet()** to set the action function to be called when text has been entered into a cell of the attribute window of the tree.

Use the function **ProUITreeTreecelldeleteActionSet()** to set the action function to be called when the DELETE key has been pressed in a cell of the attribute window of the tree.

Use the function **ProUITreeMoveActionSet()** to set the action function to be called when the sash of the tree between the tree hierarchy and the attribute window has been moved.

Master Table of Resource File Attributes

Resource File Attribute	Permitted Values	Description
.ActivateOnReturn	Of Boolean type TRUE (default)	Flag indicating whether a RETURN key press should generate a UI_ACTIVATE_ACTION callback or whether it should cause the default button in the Dialog to be pressed.
.ArcDirection	Enumerated type <i>ProUIArcDirection</i>	The direction to use when drawing arcs in a drawing area: PROUIARCDIR_CLOCKWISE draws in a clockwise direction between the two line segments in order to form the arc. PROUIARCDIR_COUNTERCLOCKWISE Draws in a counterclockwise direction between the two line segments in order to form the arc.
.ArcFillMode	Enumerated type <i>ProUIArcFillMode</i>	The type of fill to generate for an arc drawn in a drawing area: PROUIARCFILL_PIEFills the arc bounded by two line segments joining the endpoints of the arc and its center point. PROUIARCFILL_CHORDFills the arc bounded by a single line segment that joins the two endpoints of the arc.
.Alignment	Enumerated type <i>ProUIAlignment</i> .Default is "left"	The alignment of the text:
.AttachBottom	Of Boolean type TRUE (default), FALSE	The four Attach attributes specify to which sides of its grid location a component is attached. If it is attached to none the component will float inside the area available without changing size when the dialog is resized. If it is attached to the left only, for example, it will stay at the left of the area available. If it is attached both left and right, it will stretch horizontally to fit the space available. If it is attached both top and bottom it will stretch vertically.
.AttachTop		
.AttachRight		
.AttachLeft		
.Autohighlight	Of Boolean type FALSE (default)	Flag indicating whether to highlight the entire row when a selection is made in a cell in the row.
.BackgroundColor	Integer type based on <i>ProUIColorType</i> or a user defined color type.	The background color of the component as a user specified color or a window color of <i>ProUIColorType</i> .

Resource File Attribute	Permitted Values	Description
.BgColor	Integer type based on <i>ProUIColorType</i> or a user defined color type.	The background drawing color of the component as a user specified color or a window color of <i>ProUIColorType</i> . This cannot be set in a resource file.
.BottomOffset	Of Integer type Default Value is -1	The four Offset attributes describe the minimum gap, in pixels, between the component and the edge of the area available to it. The maximum value of an offset is 20. The default, -1, is equivalent to no offset. It is usual in Pro/ENGINEER to put an offset of 4 around each component, so the minimum distance between components is 8 pixels. If you have a vertical column of components such as input panels in a single layout, you can reduce the overall vertical gap between them to 4 pixels.
.TopOffset		
.RightOffset		
.LeftOffset		
.Bitmap	Of String type	The icon of the component. This may be a PNG, PCX, GIF, BMP, ICO or CUR file. The file must be visible to Pro/ENGINEER by residing in Pro/ENGINEER's text directory or in the Pro/TOOLKIT application's text directory.
.ButtonStyle	Enumerated type <i>ProUIButtonStyle</i>	Following are the available types: 1. PROUIBUTTONSTYLE_CHECK—The horizontal and vertical margins are equal, no shadow border is drawn and by default, the button accepts keyboard input. 2. PROUIBUTTONSTYLE_TOGGLE—The horizontal margin is twice as wide as the vertical margin, the shadow border is always drawn and by default, the button accepts keyboard input. 3. PROUIBUTTONSTYLE_TOOL—The horizontal and vertical margins are equal, the shadow border is always drawn and by default, the button does not accept keyboard input. 4. PROUIBUTTONSTYLE_FLAT—The horizontal and vertical margins are equal, the shadow border is only drawn when the pointer is moved over the component and by default the button does not accept keyboard input. 5. PROUIBUTTONSTYLE_LINK—The button works like a hyperlink.

Resource File Attribute	Permitted Values	Description
.CascadeDirection	Enumerated type <i>ProUICascadeDirection</i>	The direction in which the MenuPane should cascade when the CascadeButton is activated. Following are the possible values: PROUICASCADEDIR_TOP_LEFT—Up and to the left PROUICASCADEDIR_TOP_RIGHT—Up and to the right PROUICASCADEDIR_BOTTOM_LEFT—Down and to the left PROUICASCADEDIR_BOTTOM_RIGHT—Down and to the right
.ChildNames	Of String array type	Names of the children of the component. Read Only. This cannot be specified in a resource file.
.ClassName	Of String type	Name of the class of the component. It is Read-only type. These are defined in the ProUI.h file. This cannot be specified in a resource file.
.ClipChildren	Of Boolean type TRUE (default), FALSE	Flag indicating whether drawing operations within a DrawingArea are clipped so that they do not overlap any children contained within the DrawingArea.
.Columns	Of Integer type Default Value is 16	The width of the InputPanel, in character widths. Column widths are determined by the widest possible character in the font of the component (typically 'W'). Thus, a component with a width of 16 will likely hold words with more than 16 characters.
.ColumnLabel	Of WideString type	The tab-separated labels to be displayed as the column headers of a List.
.ColumnLabels	Of WideString type	The labels of the columns of a table.
.ColumnNames	Of String Array type	The names of the columns of a table.
.ColumnResizings	Of Integer Array type Default Value is 0	The resizing factors of the columns of a Table.
.ColumnSelectionPolicy	Enumerated type <i>ProUISelectionpolicy</i>	The type of selection allowed for the columns of a table: PROUISELPOLICY_SINGLE—0 or 1 items can be selected PROUISELPOLICY_BROWSE—exactly one item can be selected. PROUISELPOLICY_MULTIPLE—more than one item can be selected PROUISELPOLICY_EXTENDED—more than one item can be selected, 'Shift' plus select can select a range of items If 0, no selection is permitted.

Resource File Attribute	Permitted Values	Description
.ColumnWidths	Of Integer Array type Default Value is the value of <i>DefaultColumnWidth</i>	The widths of the columns of a Table, in character widths. You can refer to .Columns
.Decorated	Boolean, default is true	Flag indicating whether the component has a decorated shadow border.
.DefaultButton	Of String type	The name of the component of the dialog which is to be treated as the default button of the dialog. The default button is automatically activated when the user hits RETURN or presses the middle mouse button within the Dialog.
.DefaultColumnWidth	Of Integer type Default Value is 8	The default column width if no widths are specified or if any of the ColumnWidths members are less than or equal to 0.
.Denominator	Of Integer type Default Value is 1	The denominator value of the fractional contents of the InputPanel.
.DialogStyle	Enumerated type <i>ProUIDialogStyle</i>	The following are the dialog style types supported by Pro/TOOLKIT: 1. Modal type with respect to its parent Dialog = PROUIDIALOGSTYLE_PARENT_MODAL; This should be the value for all dialogs except those involving progress bars. 2. A transient dialog type is used only to display some state information, usually referring to the progress of some kind of lengthy operation = PROUIDIALOGSTYLE_WORKING; This should be the value for dialogs with progress bars. Other dialog style values are not currently supported for Pro/TOOLKIT dialogs.
.Digits	Of Integer type Default Value is -1	The number of digits to be displayed if the contents are being treated as a number. A value of 0 indicates that this attribute should be ignored when formatting the value.
.Double	Of Float type Default Value is 0	The double value of the contents of the component.

Resource File Attribute	Permitted Values	Description
.DoubleFormat	Of String type Default Value is "%. *lf"	The format of the contents of the component if they are being treated as a double. The value of this attribute is a C formatting string, which can handle the precision and value of the contents of the InputPanel. Note: the formatting is not applied to values entered into the component until the component's Double value is accessed programmatically.
.DoubleIncrement	Of Float type Default Value is 1	Slow increment to be used when Spinbox value is a double.
.DrawingHeight	Of Integer type Default Value is 100	Height in pixels of the drawing area.
.DrawingMode	Enumerated type <i>ProUIDrawingMode</i>	The drawing mode of the DrawingArea. Following are the possible values: PROUIDRWMODE_COPY—Draw using the foreground drawing color. PROUIDRWMODE_NOT—Draw, inverting the existing pixels of the component. PROUIDRWMODE_AND—Draw using a combination of the foreground drawing color AND the existing pixels. PROUIDRWMODE_OR—Draw using a combination of the foreground drawing color OR the existing pixels. PROUIDRWMODE_XOR—Draw using a combination of the foreground drawing color XOR the existing pixels.
.DrawingWidth	Of Integer type Default Value is 100	Width in pixels of the drawing area.
.Editable	Of Boolean type TRUE (default), FALSE	Flag indicating whether the text contents of the component may be modified by the user.
.FastDoubleIncrement	Of Float type Default Value is 10	The fast increment to be used when the SpinBox value is a double
.FastIncrement	Of Integer type Default Value is 10	The fast increment to be used when the SpinBox value is an integer.

Resource File Attribute	Permitted Values	Description
.FillMode	Enumerated type <i>ProUIFillMode</i>	The current drawing fill mode of the DrawingArea. Following are the possible values: PROUIFILLMODE_SOLID—Fill using the foreground drawing color. PROUIFILLMODE_LIGHT_STIPPLE— Fill using a 75% stipple of the foreground and background drawing colors. PROUIFILLMODE_MEDIUM_STIPPLE— Fill using a 50% stipple of the foreground and background drawing colors. PROUIFILLMODE_HEAVY_STIPPLE— Fill using a 25% stipple of the foreground and background drawing colors.
.Focus	Of String type	The name of the component on which the cursor is positioned when the dialog is activated. If the Dialog is already active then this specifies the name of the component within the Dialog on which on which the focus is set and the cursor is positioned.
.FontClass	Enumerated type <i>ProUIFontClass</i>	The base font class to be used to draw text in the component.
.FontSize	Of Float type Default Value is 0	Point size of the font used to draw text in the component.
.FontStyle	Enumerated type <i>ProUIFontStyle</i>	A bitwise OR of the styles of the font used to draw text in the component. Note: All styles are not supported for a given font class and platform.
.ForegroundColor	Integer type based on <i>ProUIColorType</i> or a user defined color type.	The foreground color of the component.
.FgColor	Integer type based on <i>ProUIColorType</i> or a user defined color type.	The foreground drawing color of the component as a user specified color or a window color of <i>ProUIColorType</i> .
.Height	Of Integer type Default Value is -1	Height in pixels of the component.
.HelpText	Of WideString type	The popup help-text to be displayed whilst the pointer is over the component.
.HorzAtPoint	Of Integer type Default Value is 0	The horizontal location from the existing dialog to which a new dialog is being positioned horizontally. The possible alignment types are left/ center/right.
.HorzDialog	Of String type	The name of the dialog to which the existing dialog is relatively being positioned horizontally.
.HorzMethod	Enumerated type <i>ProUIHorzPosition</i>	Horizontal positioning method of the dialog.

Resource File Attribute	Permitted Values	Description
.HorzPoint	Enumerated type ProUIHorzPosition.	The horizontal location on the dialog which is used for positioning.
.HorzPosition	Of Integer type Default Value is -1	Desired horizontal position of the dialog. This is the absolute position.
.HorzPosOffset	Of Integer type Default Value is -1	The desired horizontal offset between the dialog to which it is being relatively positioned and the existing dialog.
.HorzSize	Of Integer type Default Value is -1	The desired width of the dialog.
.Images	Of String array typeDefault Value is NULL	The names of the images, which will be drawn in a DrawingArea. Each image may be a PNG, PCX, GIF, BMP, ICO or CUR file.
.Increment	Of Integer type Default Value is 1	The slow increment to be used when the SpinBox value is an integer.
.Integer	Of Integer typeDefault Value is 0	The integer value of the contents of the component.
.InputType	Enumerated type <i>ProUIInputtype</i>	Datatype of the contents of the input panel. Following are the possible types: PROUIINPUTTYPE_STRING PROUIINPUTTYPE_WSTRING PROUIINPUTTYPE_INTEGER PROUIINPUTTYPE_DOUBLE PROUIINPUTTYPE_FRACTION PROUIINPUTTYPE_RELATION
.ItemHelpText	Of WideString Array type	The popup help-text to be displayed for the items of the component whilst the pointer is over the component. The array size should match the number of items in the component.
.ItemImage	Of String Array type	The images of the items of the component. Each image may be a PNG, PCX, GIF, BMP, ICO or CUR file. The array size should match the number of items in the component.
.Label	Of WideString type	The text shown on the component. (For dialogs: this is the dialog title).
.Labels	Of WideString type	The user visible text for each of the items of the component.
.LabelAlignment	Of Integer type Default Value is 2	The justification of the label of the Tree if the Tree has an attribute window and can be as follows: PROUIALIGNMENT_LEFT PROUIALIGNMENT_RIGHT PROUIALIGNMENT_CENTER

Resource File Attribute	Permitted Values	Description
.Lastentereditem	String	The name of the item which was beneath the pointer the last time the pointer was over the component.
.Length	Of Integer type Default Value is 8	The length of the Slider, in character widths.
.ListState	Of Integer Array type	The state of each of the items of the List.
.LineStyle	Enumerated type <i>ProUILineStyle</i>	The current line drawing style of the DrawingArea. Following are the available types: PROUILINESTYLE_SOLID—Draw solid lines in the foreground drawing color. PROUILINESTYLE_DOTTED—Draw dotted lines in the foreground and background drawing colors. PROUILINESTYLE_DASHED—Draw dashed lines in the foreground and background drawing colors.
.ListType	Enumerated type <i>ProUIListtype</i>	The possible list types are as follows: PROUILISTTYPE_STANDARDNo column headers or check marks. PROUILISTTYPE_TABULATEDDisplay column headers, but do not display check marks. PROUILISTTYPE_CHECKDisplay check marks, but do not display column headers. The value of the check mark for each item is accessed by ProUIListStateGet() and ProUIListStateSet()
.LockedColumns	Of Integer type Default Value is 0	The number of locked columns of the table.
.LockedRows	Of Integer type Default Value is 0	The number of locked rows of the Table.
.Mapped	Of Boolean type FALSE (default)	For a dialog component this flag indicates whether the dialog is visible on the screen or not. For all other components this flag indicates whether the component occupies any space when it is invisible.
.MaxDouble	Of Float type 8.507e+37	The maximum double value of the contents of the component.
.MaxInteger	Of Integer type 2147483647	The maximum integer value of the contents of the component.
.MaxLen	Of Integer type Default Value is 32	The maximum length of the text contents of the component.
.MinColumns	Of Integer type Default Value is 4	The minimum width of the component, in character widths.

Resource File Attribute	Permitted Values	Description
.MinDouble	Of Float type -8.507e+37	The minimum double value of the contents of the component.
.MinInteger	Of Integer type -2147483647	The minimum integer value of the contents of the component.
.MinRows	Of Integer type Default Value is 1	The minimum number of visible rows of the component.
.Names	Of String Array type	The names of the items of the component.
.Numerator	Of Integer type Default Value is 0	The numerator value of the fractional contents of the InputPanel.
.Ordinal	Of Integer type Default Value is 0	The ordinal value of the fractional contents of the InputPanel.
.Orientation	Enumerated type <i>ProUIOrientation</i>	The orientation of the component. It is of the following types: PROUI_HORIZONTAL PROUI_VERTICAL
.ParentName	Of String type	The name of the parent component of the component.
.Password	Of Boolean type FALSE (default)	Flag indicating whether the component is used for password entry.
.PolygonFillMode	Enumerated type <i>ProUIPolygonFillMode</i>	The fill mode to be used when drawing Polygons in the DrawingArea.
.PopupMenu	Of String type	Allows you to designate the popup menu for a component This is the name of a MenuPane component to use as a popup menu for the given component.
.PopupWhenInsen	Of Boolean type	Flag indicating whether the component should display its popup menu when it is insensitive (TRUE) or whether no popup menu should be displayed at such times (FALSE).
.ProgressStyle	Enumerated type <i>ProUIProgressstyle</i>	The display style of the ProgressBar. It is of the following types: PROUIPROGRESS_NOTEXT PROUIPROGRESS_VALUE PROUIPROGRESS_PERCENT PROUIPROGRESS_INTERVALS
.RememberPosition	Of Boolean type TRUE (default), FALSE	Controls whether Pro/ENGINEER should store the location of the dialog when it is destroyed, and apply the position to the dialog again if it is shown again.
.RememberSize	Of Boolean type TRUE (default), FALSE	Controls whether Pro/ENGINEER should store the size the dialog when it is destroyed, and apply the size to the dialog again if it is shown.

Resource File Attribute	Permitted Values	Description
.Resizable	Of Boolean type TRUE (default), FALSE	Identifies whether a component is allowed to resize based on changing of content. If the content is larger than the component, set this to TRUE to resize to contain the new content. Set it to FALSE to truncate or shorten the content.
.ResizableCols	Of Boolean type TRUE (default), FALSE	Flag indicating whether the Tree columns may be resized using the column headers as draggable sashes.
.Rows	Of Integer type Default Value is 8	The number of rows of the component.
.RowLabels	Of WideString Array type	The labels of the rows of the component. If labels are specified then the table will display row headers, otherwise no row headers are displayed.
.RowNames	Of String Array type	The names of the rows of the component.
.ScrollBarsWhenNeeded	Of Boolean type	Flag indicating whether scrollbars should only be displayed when they are required (TRUE) or whether they should always be displayed (FALSE).
.SelectableNames	Of String Array type	The names of the items of the component that may be selected. An empty array indicates that every item is currently selectable.
.SelectedNames	Of String Array type	The names of the currently selected items of the component. An empty array indicates that no item of the component is selected. This attribute may not be set in the resource file.
.SelectedColumnNames	Of String Array type	The names of the currently selected columns of the Table. This attribute may not be set in the resource file.
.SelectedRowNames	Of String Array type	The names of the currently selected rows of the Table. This attribute cannot be set through the resource file.
.SelectionPolicy	Enumerated type <i>ProUISelectionPolicy</i>	The type of selection allowed for items in the component: PROUISELPOLICY_SINGLE—0 or 1 items can be selected PROUISELPOLICY_BROWSE—exactly one item can be selected. PROUISELPOLICY_MULTIPLE—more than one item can be selected PROUISELPOLICY_EXTENDED—more than one item can be selected, 'Shift' plus select can select a range of items If 0, no selection is permitted

Resource File Attribute	Permitted Values	Description
.SelectByCell	Of Boolean type	Flag indicating whether the Tree attribute window allows selection by cell (TRUE) or whether all the cells of the selected node of the Tree are selected when a node is selected (FALSE).
.SelectCBRegardless	Of Boolean type	Flag indicating whether the Tree should generate a UI_SELECT_ACTION callback when the currently selected item is re-selected (TRUE) or whether such a selection should be ignored as the state has not changed.
.Sensitive	Of Boolean type	Flag indicating whether the component is disabled (FALSE) or sensitive to user input (TRUE).
.Set	Of Integer type Default Value is 0	The state of the component.
.ShowGrid	Of Boolean type	Flag indicating whether to display the grid lines of the Table.
.StartLocation	Enumerated type <i>ProUIStartLocation</i>	Following are the possible user defined start locations and anchor positions for a UI component: PROUIDEFAULT_LOCATION—default location PROUITOP_LEFT—top left-hand corner PROUITOP_MIDDLE—top edge PROUITOP_RIGHT—top right-hand corner PROUIMIDDLE_LEFT—left-hand edge PROUIMIDDLE_MIDDLE—middle PROUIMIDDLE_RIGHT—right-hand edge PROUIBOTTOM_LEFT—bottom left-hand corner PROUIBOTTOM_MIDDLE—bottom edge PROUIBOTTOM_RIGHT—bottom right-hand corner
.String	Of String type	The contents of the InputPanel as a string
.TabCharsAllow	Of Boolean type	Flag indicating the behavior of the input panel when the TAB key is pressed. If the value is True, inserts a TAB key press into the text of input panel as a tab character. If the value is False, uses the TAB key press for focus traversal. The input panel ignores the tab key press. This attribute may not be set in the resource file.
.TitleBarImage	Of String type	The icon placed in the title bar of a dialog.

Resource File Attribute	Permitted Values	Description
.Tracking	Of Boolean type	Flag indicating whether the DrawingArea generates a UI_MOVE_ACTION whenever the pointer is moved over the visible region of the component.
.TreeAttributeWindow	Of Boolean type	Flag indicating whether the Tree has an attribute window.
.TreeBoxNodes	Of Boolean type	Flag indicating whether the nodes of the Tree should be displayed with a bounding rectangle.
.TreeCellInput	Of WideString type	The text entered by the user which caused a UI_TREE_CELL_INPUT_ACTION to be generated. This cannot be specified in the resource file.
.TreeCellSelCol	Of String type	The name of the column of the selected cell of the attribute window of the Tree. This cannot be specified in the resource file.
.TreeCellSelNode	Of String type	The name of the node of the selected cell of the attribute window of the Tree. This cannot be specified in the resource file.
.TreeChildOrder	Of Integer type Default Value is 1	The method used to order the children of a parent node of the Tree.
.TreeColumnsJusts	Of Integer Array type	The justifications of the text of the columns of the attribute window of the Tree.
.TreeColumnsNames	Of String Array type	The names of the columns of the attribute window of the Tree.
.TreeColumnsTitles	Of WideStringArray type	The labels of the column headers of the attribute window of the Tree.
.TreeColumnsWidths	Of Integer Array type	The widths of the columns of the attribute window of the Tree.
.TreeColumnOrder	Of String Array type	The display order of the columns of the attribute window of the Tree. An empty array indicates that the columns should be displayed in the order in which they were created.
.TreeCurrent	Of String type	The name of the current node of the Tree. This is the node which is drawn with a dotted focus rectangle if the Tree component has the keyboard input focus. This cannot be specified in the resource file.
.TreeDisplayRoot	Of Boolean type	Flag indicating whether the root node of the Tree should be displayed (TRUE) or whether it should be hidden from view (FALSE).

Resource File Attribute	Permitted Values	Description
.TreeExpColNode	Of String type	The name of the node of the Tree which was expanded or collapsed to generate a UI_EXPAND_ACTION or a UI_COLLAPSE_ACTION, respectively. This cannot be specified in the resource file.
.TreeIndicateChildren	Of Boolean type	Flag indicating whether the nodes of the Tree should be displayed with a "..." suffix to indicate that they have children when the nodes themselves are not expanded.
.TreeInitialSashPercent	Of Integer type Default Value is 50	The initial percentage position of the sash of the Tree between the Tree hierarchy and the attribute window of the Tree.
.TreeKeyboardInput	Of Boolean type	Flag indicating whether the attribute window of the Tree should respond to keyboard input and generate a UI_TREE_CELL_INPUT_ACTION.
.TreeLinkStyle	Of Integer type Default Value is 2	The link-style used to indicate the children of a parent node of the Tree.
.TreeNodeTypeAppends	Of WideStringArray type	The text to be appended to the labels of the nodes of the node types of the Tree.
.TreeNodeTypeCollapsedImages	Of String Array type	The image to be displayed with the collapsed nodes of the node types of the Tree.
.TreeNodeTypeExpandedImages	Of String Array type	The image to be displayed with the expanded nodes of the node types of the Tree.
.TreeNodeTypeHelpTexts	Of WideStringArray type	The help text of the nodes of the node types of the Tree.
.TreeNodeTypeNames	Of String Array type	The names of the node types of the Tree.
.TreeNodeTypePrefixs	Of WideStringArray type	The text to be prepended to the labels of the nodes of the node types of the Tree.
.TreeRedraw	Of Boolean type	Flag indicating whether redraws are allowed in the Tree. This cannot be specified in the resource file.
.TreeRootNode	Of String type	The name of the root node of the Tree. This cannot be specified in the resource file.
.TreeVerticalSBPosition	Of Integer type Default Value is 3.	The position of the vertical scrollbar of the Tree.
.TruncateLabel	Of Boolean type	Flag indicating whether to truncate the labels of newly created cells in the Table to the size of their cell. Modifying this after adding rows and columns will have no effect on any existing cell's appearance.
.UnitsPerRotation	Of Integer type Default Value is 360.	The amount by which the value of the thumbwheel should be incremented for each complete revolution of the wheel.

Resource File Attribute	Permitted Values	Description
.Value	Of WideString type	The text contents of the component.
.VertAtPoint	Enumerated type <i>ProUIVertPosition.</i>	The vertical location from the existing dialog to which a new dialog is being positioned vertically.
.VertDialog	Of String type	The name of the dialog to which the existing dialog is relatively being positioned.
.VertMethod	Enumerated type <i>ProUIPositioningMethod</i>	Vertical positioning method of the dialog.
.VertPoint	Enumerated type <i>ProUIVertPosition.</i>	The vertical location on the dialog, which is used for positioning.
.VertPosOffset	Of Integer type Default Value is -1	The desired vertical offset between the dialog to which it is being relatively positioned and the existing dialog.
.VertSize	Of Integer type Default Value is -1	The desired height of the dialog.
.Visible	Of Boolean type	Flag indicating whether the component is shown (TRUE) or hidden (FALSE).
.VisibleNames	Of String Array type	The names of the visible items of the component. An empty array indicates that every item of the component is visible.
.VisibleRows	Of Integer type Default Value is 4	The number of visible rows of the component.
.WideString	Of WideString type	The contents of the InputPanel as a wide-string.
.Width	Of Integer type Default Value is -1	Width of the component, in pixels.

Using Resource Files

Resource files are text files that describe the overall structure of a dialog box.

Note: For most Pro/TOOLKIT applications, it is not required to use resource files. Instead you can use the functions described in the previous sections to create, lay out, and populate dialog boxes directly. Information in this section is provided for the few applications that still require resource files.

When the Pro/TOOLKIT application wants to show a dialog box to the Pro/ENGINEER user, it simply asks Pro/ENGINEER to load the dialog box from the resource file. The first task for the Pro/TOOLKIT user who wants to display a dialog box is to write the resource file.

The resource file describes:

- Overall attributes of the dialog box.
- A list of components it contains.
- Attributes of the components themselves and the relative positions of the components.
- Rules for how they behave when the user resizes the dialog box.

Many of the dialog box and component attributes can also be read and modified programmatically with Pro/TOOLKIT functions.

The resource files used by Pro/ENGINEER dialog boxes can be found in the following directory:

```
<proe_loadpoint>/text/resource
```

A resource file must be called `dialog_name.res` where `dialog_name` is the name of the dialog box. The name of the resource file should be in the lower case. Resource files used by the Pro/TOOLKIT application must be stored under the `text` directory referred to by the `text_dir` statement in the Pro/TOOLKIT registry file.

Location and Translation of Resource Files

If the application uses language-specific directories, the resource file must be located in the

```
<proe_loadpoint>/text/<language>/resource
```

directory. One resource file must exist for each language supported by the application. If the application supports only one language, then the resource file may be located in the

```
<proe_loadpoint>/text/resource
```

directory.

A Pro/TOOLKIT application can optionally include only one resource file for all supported languages in the

```
<proe_loadpoint>/text/resource
```

directory. This application should also include a text file containing the translated entities and translated text for each additional language, in the format used by the function **ProMessageDisplay()**, in the

```
<proe_loadpoint>/text/<language>/resource
```

directory. The translation text file should include the following items:

- The component and attribute for each translated item in line 1
- The English text in line 2
- The translated entities and translated text in line 3
- Line 4 should be left blank

Function Introduced:

- **ProUITranslationFilesEnable()**

Use the function **ProUITranslationFilesEnable()** to set your Pro/TOOLKIT application to use the single resource file and multiple translation files method of deploying resource files for translated dialog boxes. Call this function from your application's **user_initialize()** function before any call is made to another *ProUI** function.

Syntax of Resource Files

The resource file is composed of nested statements; each statement is enclosed in parentheses, and contains either a keyword or the name of a dialog or component attribute followed by one or more values and/or other statements.

The top-level statement in a resource file for dialog must always be

```
(
Dialog dialog_name      (other statements...  )
)
```

where `dialog_name` is the name of the dialog itself. The dialog name is used to refer to the dialog from the source code of the application. The name of a dialog can be of any length, and contains alphanumeric characters and underscores. Case is ignored.

If the resource file contains only a layout, the top-level statement would be

```
(
Layout layout_name      (other statements...  )
)
```

instead. Collectively, the top statement describes the outermost container in the resource file.

The two statements that follow the name of the container are always Components and Resources.

The Components statement simply lists the types and names of the components that the dialog contains.

Components of type `Layout`, `MenuPane`, `CascadeButton`, `Table`, or `Tree` may have their own container-level statements following the `Dialog` statement. The formats of these statements are exactly the same as the `Dialog` statement.

If one of the components is a tab, the layouts that the tab contains are listed after the tab name in the Components part of the `Dialog` statement. Similarly, if one of the components is a `MenuBar` or a `CascadeButton`, the `MenuPanes` that it contains are listed after the `MenuBar` name in the Components part of the `Dialog` statement.

A `Resources` statement contains one or more attribute assignments. Each attribute assignment statement is of this form

```
(componentname.attributename    value)
```

The `Resource` statement defines the attributes of the dialog and the components, and the `Layout` which defines the relative positions of the components, and the way in which they behave when the dialog is resized. If the component name is missing, the statement (for example, `(.attributename value)`) will be assumed to apply to an attribute of the dialog or layout itself. The attributes and their values are described in detail in the following sections of this chapter.

The last attribute statement in a `Resources` statement is a special one called `.Layout`. The value of a `.Layout` statement is always a single statement of type `Grid`. The `Grid` statement describes a flexible grid of rows and columns in which the components are placed; this defines the neighbor-relations between components, in other words their relative positions. The absolute positions, and the sizes, of the components may change as the grid stretches and shrinks in response to the user resizing the window containing the dialog. Components of type `MenuPane` do not require a `.Layout` statement.

The `Grid` statement contains the following values:

- `Rows` statement—lists the rows in the grid.
- `Cols` statement—lists the columns in the grid.
- `Values` list—specify the contents of each grid location in turn, reading left-to-right, top-to-bottom. Each of the values can be either

- The name of a component
- A Pos statement, to specify the row and column number of the next component in the list, if it is not in the next available location. A Pos statement allows you to skip some locations, leaving them empty.
- Another Grid statement, to show that the location contains several components on their own local grid.

The value of a Rows or Cols statement is a list of integers, one for each row or column. The value of the integer is 0 if the row or column cannot be resized when user resized the dialog. the integer value is 1 if it can be resized. (It is normal to set this to 1, so that all the components in the dialog will stretch when the dialog stretches.)

The size and position of the components within the grid is also partly determined by the values set for the Attach and Offset attributes described in more detail in the section Master Table of Resource File Attributes.

Note: Ensure that the resource file contains a new line after the last parenthesis, or else the function **ProUIDialogCreate()** will return an error.

Example 8: Dialog with All Components

```
(Dialog allcomponents
  (Components
    (PushButton                      PushButton1)
    (PushButton                      PushButton2)
    (Tab                             Tab3
      (Layout11
        (Layout8)
          (Label                      Label16)
        )
      )
  )
  (Resources
    (PushButton1.Label               "OK")
    (PushButton1.TopOffset           4)
    (PushButton1.BottomOffset        4)
    (PushButton1.LeftOffset          4)
    (PushButton1.RightOffset         4)
    (PushButton2.Label               "Cancel")
    (PushButton2.TopOffset           4)
    (PushButton2.BottomOffset        4)
    (PushButton2.LeftOffset          4)
    (PushButton2.RightOffset         4)
    (Tab3.Decorated                  True)
    (Label16.Label                   "Push Buttons")
    (.Label                          "All Components")
  )
)
```

```

        (.Layout
            (Grid (Rows 1 1 1) (Cols 1)
                Tab3
                Label16
                (Grid (Rows 1) (Cols 1 1)
                    PushButton1
                    PushButton2
                )
            )
        )
    )
)
(Layout Layout11
    (Components
        (SubLayout
            Layout1)
    )

    (Resources
        (.Label
            "Tab with two layouts")
        (.Decorated
            True)
        (.Layout
            (Grid (Rows 1) (Cols 1)
                Layout1
            )
        )
    )
)
)
(Layout Layout1
    (Components
        (InputPanel
            InputPanel1)
        (Label
            Label1)
        (CheckBoxButton
            CheckBoxButton1)
        (Label
            Label2)
        (Label
            Label3)
        (Label
            Label4)
        (List
            List1)
        (Label
            Label5)
        (OptionMenu
            OptionMenu1)
        (Label
            Label6)
        (Label
            Label8)
        (RadioGroup
            RadioGroup1)
        (Slider
            Slider1)
        (Label
            Label9)
        (TextArea
            TextArea1)
        (Label
            Label10)
        (CheckBoxButton
            CheckBoxButton3)
        (Label
            Label7)
        (Label
            Label11)
        (ThumbWheel
            ThumbWheel1)
        (ProgressBar
            ProgressBar1)
        (SpinBox
            SpinBox1)
    )
)

```

```

        (Label
        (Separator
        (Label
        (Table
        (Label
    )
    (Resources
        (InputPanel1.InputType
        (Label1.Label
        (Label1.AttachLeft
        (CheckBox1.Label
        (CheckBox1.Set
        (CheckBox1.TopOffset
        (CheckBox1.BottomOffset
        (CheckBox1.LeftOffset
        (CheckBox1.RightOffset
        (Label2.Label
        (Label2.AttachLeft
        (Label2.TopOffset
        (Label2.BottomOffset
        (Label2.LeftOffset
        (Label2.RightOffset
        (Label3.Label
        (Label3.AttachLeft
        (Label4.Label
        (List1.Names
        (List1.Labels
        (Label5.Label
        (Label5.AttachLeft
        (OptionMenu1.Names
        (OptionMenu1.Labels
        (Label6.Label
        (Label6.AttachLeft
        (Label8.Label
        (Label8.AttachLeft
        Label14)
        Separator1)
        Label13)
        Table1)
        Label17)
        2)
        "Input Panel")
        True)
        "Check2")
        True)
        4)
        4)
        4)
        4)
        "Check Buttons")
        True)
        4)
        4)
        4)
        4)
        "Label")
        True)
        "Label Text")
        "n1"
        "n2"
        "n3"
        "n4"
        "n5")
        "Value 1"
        "Value 2"
        "Value 3"
        "Value 4"
        "Value 5")
        "List")
        True)
        "n1"
        "n2"
        "n3"
        "n4"
        "n5")
        "Option 1"
        "Option 2"
        "Option 3"
        "Option 4"
        "Option 5")
        "Option Menu")
        True)
        "Radio Group")
        True)

```

```

(RadioGroup1.Orientation      True)
(RadioGroup1.AttachLeft      False)
(RadioGroup1.AttachRight     False)
(RadioGroup1.AttachTop       False)
(RadioGroup1.AttachBottom    False)
(RadioGroup1.Names           "n1 "
                             "n2 "
                             "n3 ")
(RadioGroup1.Labels          "Rad1 "
                             "Rad2 "
                             "Rad3 ")

(RadioGroup1.Alignment       2)
(Label9.Label                 "Slider")
(Label9.AttachLeft            True)
(Label10.Label                "Text Area")
(Label10.AttachLeft           True)
(CheckButton3.Label           "Check1")
(Label7.Label                 "Thumbwheel")
(Label7.AttachLeft            True)
(Label11.Label                "Progress bar")
(Label11.AttachLeft           True)
(Label14.Label                "Spinbox")
(Label14.AttachLeft           True)
(Label13.Label                "Separator")
(Label13.AttachLeft           True)
(Table1.RowNames              "r1 "
                             "r2 "
                             "r3 ")
(Table1.RowLabels             "1 "
                             "2 "
                             "3 ")
(Table1.ColumnNames           "c1 "
                             "c2 "
                             "c3 ")
(Table1.ColumnLabels          "- A -"
                             "- B -"
                             "- C -")
(Table1.ColumnWidths          4
                             4
                             4)
(Table1.VisibleRows           4)
(Table1.ShowGrid              True)
(Label17.Label                "Table")
(Label17.AttachLeft           True)
(.Label                       "Decorated layout")
(.Decorated                   True)
(.TopOffset                   4)
(.BottomOffset                4)
(.LeftOffset                  4)
(.RightOffset                 4)
(.Layout

```

```

        (Grid (Rows 1 1 1 1 1 1 1 1 1 1 1 1 1) (Cols 1 1)
            Label2
            (Grid (Rows 1 1) (Cols 1)
                CheckButton3
                CheckButton1
            )
            Label11
            InputPanel1
            Label3
            Label4
            Label5
            List1
            Label6
            OptionMenu1
            Label111
            ProgressBar1
            Label8
            RadioGroup1
            Label13
            Separator1
            Label9
            Slider1
            Label14
            SpinBox1
            Label17
            Table1
            Label10
            TextArea1
            Label7
            ThumbWheel1
        )
    )
)

(Layout Layout8
    (Components
        (Label
            Label15)
    )
    (Resources
        (Label15.Bitmap
            "ptc_logo")
        (.Label
            "Second layout")
        (.Layout
            (Grid (Rows 1) (Cols 1)
                Label15
            )
        )
    )
)
)

```

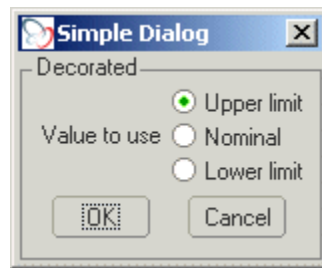

Example 9: Resource File for Dialog with Four Components on 2x2 Grid

This example shows a simple dialog which contains four components on a single 2-by-2 grid.

```
(Dialog Simple
  (Components
    (PushButton OK)
    (PushButton Cancel)
    (Label RadioLabel)
    (RadioGroup Tolgroup)
  )
  (Resources
    (OK.Label "OK")
    (OK.TopOffset 4)
    (OK.BottomOffset 4)
    (OK.LeftOffset 4)
    (OK.RightOffset 4)
    (Cancel.Label "Cancel")
    (Cancel.TopOffset 4)
    (Cancel.BottomOffset 4)
    (Cancel.LeftOffset 4)
    (Cancel.RightOffset 4)
    (RadioLabel.Label "Value to use")
    (RadioLabel.TopOffset 4)
    (RadioLabel.BottomOffset 4)
    (RadioLabel.LeftOffset 4)
    (RadioLabel.RightOffset 4)
    (Tolgroup.Orientation True)
    (Tolgroup.Names "Top"
      "Middle"
      "Bottom")
    (Tolgroup.Labels "Upper limit"
      "Nominal"
      "Lower limit")
    (.Label "Simple Dialog")
    (.Layout
      (Grid (Rows 1 1) (Cols 1 1)
        RadioLabel
        Tolgroup
        OK
        Cancel
      )
    )
  )
)
```

User Interface:
Dialogs

Figure 16-7: : Simple Dialog



The following figure shows a dialog with a subgrid that allows an input panel to take up the width of both OK and Cancel buttons. The dialog itself contains a grid with three rows and one column. The bottom location in this grid contains its own grid with one row and two columns. The following example shows the resource file for this subgrid dialog.

Figure 16-8: Subgrid Dialog



Example 10: Resource File for Subgrid Dialog

This example shows the resource file for the preceding subgrid dialog.

```
(Dialog subgrid
  (Components
    (PushButton OK)
    (PushButton Cancel)
    (InputPanel FeatNamePanel)
    (Label FeatNameLabel)
  )
  (Resources
    (OK.Label "OK")
    (OK.TopOffset 4)
    (OK.BottomOffset 4)
    (OK.LeftOffset 4)
    (OK.RightOffset 4)
    (Cancel.Label "Cancel")
    (Cancel.TopOffset 4)
    (Cancel.BottomOffset 4)
    (Cancel.LeftOffset 4)
```

```

        (Cancel.RightOffset          4)
        (FeatNamePanel.TopOffset    4)
        (FeatNamePanel.BottomOffset 4)
        (FeatNamePanel.LeftOffset   4)
        (FeatNamePanel.RightOffset  4)
        (FeatNameLabel.Label         "Feature name")
        (.Label                      "Subgrid")
        (.Layout
            (Grid (Rows 1 1 1) (Cols 1)
                FeatNameLabel
                FeatNamePanel
                (Grid (Rows 1) (Cols 1 1)
                    OK
                    Cancel
                )
            )
        )
    )
)

```

Example 11: Resource File for Subgrid Dialog with Resize

A better way to lay out the dialog in the previous example would be to place the input panel text to the left of the input panel. If this were done with a single 2-by-2 grid, the input panel label and panel itself would resize to take up the same width as the **OK** and **Cancel** buttons. To avoid this you can put the first two components in a layout of their own, which therefore has its own grid, so the columns can take up their own widths.

This example demonstrates this method.

```

(Dialog subgrid
    (Components
        (PushButton          OK)
        (PushButton          Cancel)
        (InputPanel          FeatNamePanel)
        (Label                FeatNameLabel)
    )
    (Resources
        (OK.Label            "OK")
        (OK.TopOffset        4)
        (OK.BottomOffset     4)
        (OK.LeftOffset       4)
        (OK.RightOffset      4)
        (Cancel.Label        "Cancel")
        (Cancel.TopOffset    4)
        (Cancel.BottomOffset 4)
        (Cancel.LeftOffset   4)
        (Cancel.RightOffset  4)
    )
)

```


- .BMP
- .ICO
- .CUR

Ensure that you provide the filename including a valid file extension, denoting the type of the image as one of the standard formats listed above.

Example 13: Resource File with Text Question, OK and Cancel Buttons

This example shows the use of a simple dialog, which contains a text question and OK and Cancel buttons. It would be used for any kind of confirmation request.

For a programatic method of creating this dialog box refer to Example 1: Source for Dialog with Text Question, OK and Cancel Buttons.

```
(Dialog confirm
  (Components
    (Label                                Question)
    (PushButton                          OK)
    (PushButton                          Cancel)
  )
  (Resources
    (Question.Label                      "Dummy label")
    (Question.TopOffset                  4)
    (Question.BottomOffset              4)
    (Question.LeftOffset                 4)
    (Question.RightOffset               4)
    (OK.Label                           "OK")
    (OK.TopOffset                       4)
    (OK.BottomOffset                    4)
    (OK.LeftOffset                      4)
    (OK.RightOffset                     4)
    (Cancel.Label                       "Cancel")
    (Cancel.TopOffset                   4)
    (Cancel.BottomOffset                4)
    (Cancel.LeftOffset                  4)
    (Cancel.RightOffset                 4)
    (.Label                             "Confirm")
    (.Layout
      (Grid (Rows 1 1) (Cols 1)
        Question
        (Grid (Rows 1) (Cols 1 1)
          OK                      Cancel
        )
      )
    )
  )
)
```

```
)
)
```

Example 14: UI List Resource File

The example below shows the resource file used to create UI List. For a programatic method of creating the UI List refer to Example 2: To use UI List Functions.

```
(Dialog ug_uilist
  (Components
    (SubLayout ug_uilist_layout1)
    (SubLayout ug_uilist_layout2)
  )

  (Resources
    (.Label "UI List")
    (.Layout
      (Grid (Rows 1 1) (Cols 1)
        ug_uilist_layout1
        ug_uilist_layout2
      )
    )
  )
)

(Layout ug_uilist_layout1
  (Components
    (List ug_uilist_comp)
    (Label ug_uilist_label_1)
    (TextArea ug_uilist_txtarea)
    (Label ug_uilist_lable_2)
  )

  (Resources
    (ug_uilist_label_1.Label "List area")
    (ug_uilist_txtarea.FontStyle 4)
    (ug_uilist_lable_2.Label "Text Area")
    (.Decorated True)
    (.DecoratedStyle 3)
    (.Layout
      (Grid (Rows 1 1 1 1) (Cols 1)
        ug_uilist_label_1
        ug_uilist_comp
        ug_uilist_lable_2
        ug_uilist_txtarea
      )
    )
  )
)
)
```

```
(Layout ug_uilist_layout2
  (Components
    (PushButton ug_uilist_ok)
    (PushButton ug_uilist_cancel)
  )

  (Resources
    (ug_uilist_ok.Label "Ok")
    (ug_uilist_ok.TopOffset 4)
    (ug_uilist_ok.BottomOffset 4)
    (ug_uilist_ok.RightOffset 20)
    (ug_uilist_cancel.Label "Cancel")
    (ug_uilist_cancel.TopOffset 4)
    (ug_uilist_cancel.BottomOffset 4)
    (ug_uilist_cancel.LeftOffset 20)
    (.Layout
      (Grid (Rows 1) (Cols 1 1 1)
        ug_uilist_ok
        (Pos 1 3)
        ug_uilist_cancel
      )
    )
  )
)
```

Example 15: Resource File for Dialog with Menubar

The example below shows the resource file used to create the dialog with MenuBars as described in section Example 9: Resource File for Dialog with Four Components on 2x2 Grid.

```
(Dialog menus
  (Components
    (MenuBar MenuBar1
      MenuPanel1
      MenuPane2)
    (TextArea TextArea1)
  )

  (Resources
    (TextArea1.Rows 10)
    (TextArea1.Columns 25)
    (.Label "Menubar dialog")
    (.Layout
      (Grid (Rows 1) (Cols 1)
        TextArea1
      )
    )
  )
)
```

```

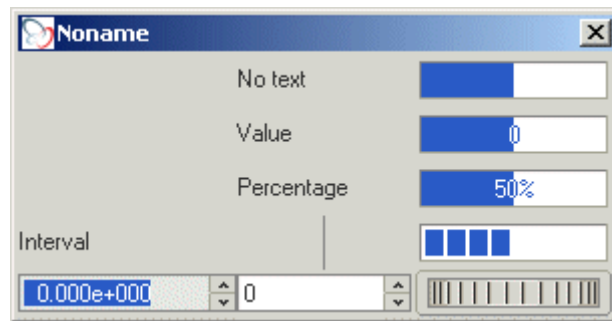
(MenuPane MenuPane1
  (Components
    (PushButton
      (PushButton
        (PushButton1)
        (PushButton2)
      )
    )
  (Resources
    (.Label
      (PushButton1.Label
        (PushButton2.Label
          "MenuPane1"
          "Button1"
          "Button2"
        )
      )
    )
  )

(MenuPane MenuPane2
  (Components
    (PushButton
      (Separator
        (RadioGroup
          (RadioGroup
            (Separator
              (CascadeButton
                (CheckBoxButton
                  (PushButton3)
                  (Separator1)
                  (RadioGroup1)
                  (RadioGroup2)
                  (Separator2)
                  (CascadeButton2)
                  (MenuPane4)
                  (CheckBoxButton1)
                )
              )
            )
          )
        )
      )
    )
  (Resources
    (.Label
      (PushButton3.Label
        (RadioGroup1.Names
          (RadioGroup1.Labels
            (CascadeButton2.Label
              (CheckBoxButton1.Label
                "MenuPane2"
                "Pushbutton"
                "RG1"
                "RG2"
                "Radio group 1"
                "Radio group 2"
                "Cascade button"
                "Checkbutton"
              )
            )
          )
        )
      )
    )
  )

(MenuPane MenuPane4
  (Components
    (PushButton
      (PushButton
        (PushButton5)
        (PushButton6)
      )
    )
  (Resources
    (PushButton5.Label
      (PushButton6.Label
        "Pushbutton"
        "Pushbutton"
      )
    )
  )
)

```


Figure 16-9: Progress Bar



Example 16: Progress Bar Resource File

```
(Dialog progressbars
  (Components
    (ProgressBar                ProgressBar2)
    (ProgressBar                ProgressBar3)
    (ProgressBar                ProgressBar4)
    (ProgressBar                ProgressBar5)
    (Label                     Label12)
    (Label                     Label13)
    (Label                     Label14)
    (Label                     Label15)
    (Separator                 Separator1)
    (SpinBox                   SpinBox1)
    (SpinBox                   SpinBox2)
    (ThumbWheel                ThumbWheel1)
  )

  (Resources
    (ProgressBar2.ProgressStyle 0)
    (ProgressBar2.TopOffset     4)
    (ProgressBar2.BottomOffset  4)
    (ProgressBar2.LeftOffset    4)
    (ProgressBar2.RightOffset   4)
    (ProgressBar3.ProgressStyle 1)
    (ProgressBar3.TopOffset     4)
    (ProgressBar3.BottomOffset  4)
    (ProgressBar3.LeftOffset    4)
    (ProgressBar3.RightOffset   4)
    (ProgressBar4.TopOffset     4)
    (ProgressBar4.BottomOffset  4)
    (ProgressBar4.LeftOffset    4)
    (ProgressBar4.RightOffset   4)
    (ProgressBar5.ProgressStyle 3)
    (ProgressBar5.TopOffset     4)
    (ProgressBar5.BottomOffset  4)
    (ProgressBar5.LeftOffset    4)
  )
)
```

```

(ProgressBar5.RightOffset      4)
(Label2.Label                  "No text")
(Label2.AttachLeft             True)
(Label3.Label                  "Value")
(Label3.AttachLeft             True)
(Label4.Label                  "Percentage")
(Label4.AttachLeft             True)
(Label5.Label                  "Interval")
(Label5.AttachLeft             True)
(Separator1.Orientation       True)
(SpinBox1.Columns              8)
(SpinBox1.Editable             False)
(SpinBox1.IntegerBase         8)
(SpinBox1.Rate                 1)
(SpinBox1.InputType           3)
(SpinBox1.Digits               3)
(SpinBox1.DoubleFormat         "%12.*1e")
(ThumbWheel1.DecoratedStyle    4)
(ThumbWheel1.InfiniteRange     False)
(.Layout
  (Grid (Rows 1 1 1 1 1 1) (Cols 1 1 1)
    (Pos 1 2)
    Label2
    ProgressBar2
    (Pos 2 2)
    Label3
    ProgressBar3
    (Pos 3 2)
    Label4
    ProgressBar4
    (Pos 5 1)
    Label5
    Separator1
    ProgressBar5
    SpinBox1
    SpinBox2
    ThumbWheel1
  )
)
)
)

```

Example 17: Component Visibility Resource File

For a programatic method of creating this dialog box refer to Example 3: Controlling Component Visibility or Sensitivity at Runtime.

```
(Dialog uguivisibility
```

```

        (Components
            (SubLayout
                (PushButton
                    PushbuttonVisibility)
                CloseButton)
        )

        (Resources
            (CloseButton.Label
                "Close")
            (CloseButton.TopOffset
                4)
            (CloseButton.BottomOffset
                4)
            (CloseButton.LeftOffset
                4)
            (CloseButton.RightOffset
                4)
            (.Label
                "Component Visibility")
            (.Layout
                (Grid (Rows 1 1) (Cols 1)
                    PushbuttonVisibility)
            CloseButton
        )
    )

    (Layout
        (Components
            (PushButton
                TargetBtn)
            (CheckButton
                VisibleCheck)
            (CheckButton
                SensitiveCheck)
            (InputPanel
                ButtonLabel)
        )
        (Resources
            (TargetBtn.Label
                "PushButton")
            (TargetBtn.TopOffset
                4)
            (TargetBtn.BottomOffset
                4)
            (TargetBtn.LeftOffset
                4)
            (TargetBtn.RightOffset
                4)
            (TargetBtn.Resizeable
                True)
            (VisibleCheck.Label
                "Visible")
            (VisibleCheck.Set
                True)
            (VisibleCheck.TopOffset
                4)
            (VisibleCheck.BottomOffset
                4)
            (VisibleCheck.LeftOffset
                4)
            (VisibleCheck.RightOffset
                4)
            (SensitiveCheck.Label
                "Sensitive")
            (SensitiveCheck.Set
                True)
            (SensitiveCheck.TopOffset
                4)
            (SensitiveCheck.BottomOffset
                4)
            (SensitiveCheck.LeftOffset
                4)
            (SensitiveCheck.RightOffset
                4)
            (ButtonLabel.Value
                "PushButton")
            (ButtonLabel.TopOffset
                4)
            (ButtonLabel.BottomOffset
                4)
            (ButtonLabel.LeftOffset
                4)
            (ButtonLabel.RightOffset
                4)
        )
    )

```

```

        (.Label                                "PushButtons")
        (.Layout
            (Grid (Rows 1) (Cols 1 1 1 1 1)
                TargetBtn
                VisibleCheck
                SensitiveCheck
                ButtonLabel
            )
        )
    )
)

```

Example 18: Resource File for Dialog with Slider and Linked InputPanel

This example shows the resource file for a dialog with slider and linked input panel. For a programatic method of creating this dialog box refer to Example 4: Source of Dialog with Slider and Linked InputPanel.

```

(Dialog angle
    (Components
        (SubLayout                                Layout1)
        (SubLayout                                Layout2)
    )
    (Resources
        (.Label                                "Angle")
        (.DefaultButton                        "OK")
        (.Layout
            (Grid (Rows 1 1) (Cols 1)
                Layout1                                Layout2
            )
        )
    )
)
(Layout Layout1
    (Components
        (Slider                                Slider)
        (InputPanel                            InputPanel)
        (Label                                Prompt)
    )
    (Resources
        (Slider.MinInteger                    -180)
        (Slider.MaxInteger                    180)
        (Slider.Length                        12)
        (Slider.Tracking                      True)
        (Slider.TopOffset                     4)
        (Slider.BottomOffset                  4)
        (Slider.LeftOffset                    4)
        (Slider.RightOffset                   4)
        (InputPanel.Columns                   4)
    )
)

```

```

        (InputPanel.AttachLeft      False)
        (InputPanel.AttachRight    False)
        (InputPanel.MinInteger     -180)
        (InputPanel.MaxInteger     180)
        (InputPanel.TopOffset      4)
        (InputPanel.BottomOffset   4)
        (InputPanel.LeftOffset     4)
        (InputPanel.RightOffset    4)
        (InputPanel.InputType      2)
        (Prompt.Label              "Dummy text")
        (.Layout
            (Grid (Rows 1 1) (Cols 1)
                Prompt
                (Grid (Rows 1) (Cols 1 1)
                    Slider
                    InputPanel
                )
            )
        )
    )
)
(Layout Layout2
    (Components
        (PushButton      OK)
        (PushButton      Cancel)
    )
    (Resources
        (OK.Label        "OK")
        (OK.TopOffset    4)
        (OK.BottomOffset 4)
        (OK.LeftOffset   4)
        (OK.RightOffset  4)
        (Cancel.Label     "Cancel")
        (Cancel.TopOffset 4)
        (Cancel.BottomOffset 4)
        (Cancel.LeftOffset 4)
        (Cancel.RightOffset 4)
        (.Layout
            (Grid (Rows 1) (Cols 1 1)
                OK      Cancel
            )
        )
    )
)
)

```

Example 19: UG Tables Resource File

The following example shows a resource file containing a table. Components may be assigned to a table through the table layout. Components of the table layout are invisible until assigned or copied in to a table cell.

```

(Dialog uguitableexample
  (Components
    (SubLayout
      (SubLayout
        (OptionsMenu
          ToCopy)
        )
      )
    )
  (Resources
    (ToCopy.Visible
      False)
    (ToCopy.AttachTop
      True)
    (ToCopy.AttachBottom
      True)
    (ToCopy.Names
      "ONE"
      "MANY")
    (ToCopy.Labels
      "Select One"
      "Select Many")
    (.Label
      "UG Tables")
    (.Layout
      (Grid (Rows 1 0 1) (Cols 1)
        TableLayout
        ButtonLayout
        ToCopy
      )
    )
  )
)

(Layout TableLayout
  (Components
    (Table
      LargeTable)
  )
  (Resources
    (LargeTable.Columns
      40)
    (LargeTable.MinRows
      4)
    (LargeTable.TopOffset
      4)
    (LargeTable.BottomOffset
      4)
    (LargeTable.LeftOffset
      4)
    (LargeTable.RightOffset
      4)
    (LargeTable.RowNames
      "A"
      "B"
      "C"
      "D"
      "E")
    (LargeTable.ColumnNames
      "1"
      "2"
      "3"
      "4")
    (LargeTable.RowLabels
      "Alpha"
      "Beta"
      "Gamma"
      "Delta")
  )
)

```

```

        (LargeTable.ColumnLabels
            "Epsilon")
            "One"
            "Two"
            "Three"
            "Four")
        (LargeTable.ShowGrid
            (.AttachLeft
                True)
            (.AttachRight
                True)
            (.AttachTop
                True)
            (.AttachBottom
                True)
            (.Layout
                (Grid (Rows 1) (Cols 1)
                    LargeTable
                )
            )
        )
    )
)

(TableLayout LargeTable
    (Components
        (PushButton
            BaseButton)
        (CheckBox
            BaseCheckBox)
        (InputPanel
            BaseInputPanel)
    )

    (Resources
        (BaseButton.Label
            "Table Button")
        (BaseCheckBox.Label
            "Table Check Button")
        (BaseInputPanel.Value
            "Table Input Panel")
    )
)

(Layout ButtonLayout
    (Components
        (PushButton
            CloseButton)
        (PushButton
            ModifySelectButton)
        (Label
            Label1)
    )

    (Resources
        (CloseButton.Label
            "Close")
        (CloseButton.TopOffset
            4)
        (CloseButton.BottomOffset
            4)
        (CloseButton.LeftOffset
            4)
        (CloseButton.RightOffset
            4)
        (ModifySelectButton.Label
            "Toggle cells")
        (ModifySelectButton.Visible
            False)
        (ModifySelectButton.TopOffset
            4)
        (ModifySelectButton.BottomOffset
            4)
        (ModifySelectButton.LeftOffset
            4)
        (ModifySelectButton.RightOffset
            4)
    )
)

```

```

        (.Layout
            (Grid (Rows 1 1) (Cols 1 1 1)
                (Pos 1 3)
                Label1
                ModifySelectButton
                CloseButton
            )
        )
    )
)

```

Example 20: UG Tables Component Resource File

The following example demonstrates different methods that can be used to assign components into table cells. The example directly assigns a stored component, copies a stored component, copies a component from elsewhere in the dialog, and copies a component from another independent dialog.

The Example 19: UG Tables Resource File is the primary resource file for the following example. The table components dialog is another dialog from which the following resource code will copy a component into the table.

For a programatic method of assigning components into table cells refer to Example 5: To Assign Components into Table Cells.

```

(Dialog uguitablecomponents
    (Components
        (PushButton ExternalButtonToCopy)
    )

    (Resources
        (ExternalButtonToCopy.Label "From other dialog...")
        (.Layout
            (Grid (Rows 1) (Cols 1)
                ExternalButtonToCopy
            )
        )
    )
)

```


17

User Interface: Dashboards

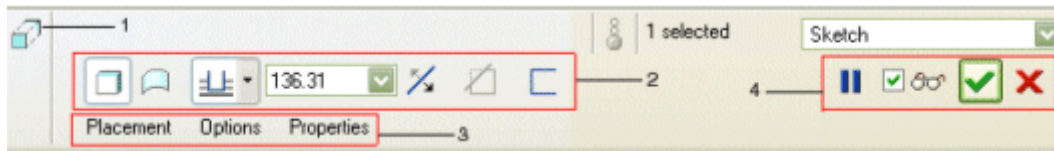
This chapter introduces the basics of dashboards and describes how Pro/TOOLKIT offers the capability to create and control dashboards. Pro/TOOLKIT applications can construct dashboards for any tool where they would typically require a dialog box.

Topic	Page
Introduction to Dashboards	17 - 2
Dashboard	17 - 3
Dashboard Page	17 - 5

Introduction to Dashboards

A dashboard is an embedded "dialog box" at the top of the Pro/ENGINEER graphics window. A dashboard typically appears when you create or modify a feature in Pro/ENGINEER. It offers the necessary controls, inputs, status and guidance for creating or editing of features.

Figure 17-1: Dashboard components



- 1 Tool Icon
- 2 Dialog Bar
- 3 Slide-down Panels
- 4 Standard Buttons

A dashboard consists of the following components:

- A main dialog bar, which show the commonly used commands and entry fields. You perform most of your modeling tasks in the graphics window and the dialog bar. When you activate a tool, the dialog bar displays commonly used options and collectors.
- Standard buttons for controlling the tool.
- Slide-down panels that open to reveal less commonly-used functionality. You can use them to perform advanced modeling actions or retrieve comprehensive feature information.
- A bitmap identifies the tool (typically the same icon used on buttons that invoke the tool).

Pro/ENGINEER uses the dashboard to create features that involve extensive interaction with user interface components and geometry manipulation. You can use dashboards in Pro/TOOLKIT applications:

- Where a dialog box is too large in size or is intrusive onto the graphics window. Dashboards enable you to make a smooth-flow tool.

- To present a streamlined "simple-user" activity with more complicated actions available to "expert users".
- Where Pro/TOOLKIT dashboards are not only limited to feature creation activities and solid model modes.

Dashboard

An opaque pointer *ProUIDashboard* is a handle to the overall dashboard tool after it has been shown.

Showing a Dashboard

Functions Introduced:

- **ProUIDashboardShow()**
- **ProUIDashboardshowoptionsAlloc()**
- **ProUIDashboardshowoptionsNotificationSet()**
- **ProUIDashboardshowoptionsFree()**
- **ProUIDashboardDestroy()**

Use the function **ProUIDashboardShow()** to push a new dashboard User Interface into the dashboard stack mechanism. The dashboard will be shown in the message area of Pro/ENGINEER. This function creates an event loop, and thus does not exit until the dashboard is being dismissed.

Use the function **ProUIDashboardshowoptionsAlloc()** to allocate a handle containing data used to build a dashboard. The input arguments for this function are:

- *main_page*—Specifies the main page for the dashboard.
- *slideup_pages*—Specifies a *ProArray* of handles representing the slide-down pages, if needed.
- *appdata*—Specifies the application data to be stored with the dashboard.

Use the function **ProUIDashboardshowoptionsNotificationSet()** to assign a callback function to be called for the indicated event occurrence in the dashboard. The input arguments for this function are:

- *options*—Specifies a handle to data used to build a dashboard.
- *notification*—Specifies the notification function to be called for the given event.

- *appdata*—Specifies the application data to be passed to the callback function when it is invoked.

You can register event notifications for the following events on the dashboard:

- **PRO_UI_DASHBOARD_CREATE**—when the dashboard is first initialized (before the pages are initialized).
- **PRO_UI_DASHBOARD_SHOW**—when the dashboard is shown or resumed.
- **PRO_UI_DASHBOARD_HIDE**—when the dashboard is paused and replaced by another tool.
- **PRO_UI_DASHBOARD_DISMISS**—upon the dashboard “close” event.
- **PRO_UI_DASHBOARD_DESTROY**—when the dashboard is finally destroyed.

You can use the function **ProUIDashboardshowoptionsFree()** to free a handle containing the data used to build a dashboard.

Use the function **ProUIDashboardDestroy()** to pop the dashboard from the dashboard stack mechanism. The dashboard User Interface will be destroyed.

Accessing a Dashboard

Functions Introduced:

- **ProUIDashboardUserdataGet()**
- **ProUIDashboardStdlayoutGet()**
- **ProUIDashboardBitmapSet()**

Use the functions **ProUIDashboardUserdataGet()** to get the application data you attached to the dashboard items upon registration. This can be used to store the current state of the tool, and thus to control the visibility of components.

Pro/TOOLKIT does not currently provide access to create the feature dashboard’s standard buttons. However, you can use **ProUIDashboardStdlayoutGet()** to get the layout name where you can create and place the buttons in the standard button area. Typically this consists of (at least) **OK** and **Cancel** buttons.

Use the function **ProUIDashboardBitmapSet()** to assign the icon for the dashboard, which appears to the left of the slide-down panel buttons.

Dashboard Page

Each section of content in a dashboard is called a dashboard page. The opaque handle *ProUIDashboardPage* represents an individual page, that is, either the dialog bar, or a single slide-down panel.

Dashboard Page Options

Functions Introduced:

- **ProUIDashboardpageoptionsAlloc()**
- **ProUIDashboardpageoptionsNotificationSet()**
- **ProUIDashboardpageoptionsFree()**

Use the function **ProUIDashboardpageoptionsAlloc()** to allocate a handle representing a single page (or layout) that will be shown in a dashboard. The input arguments for this function are:

- *page_name*—Specifies the page name (must be unique).
- *resource_name*—Specifies the name of the resource file to use (whose top component must be a layout, not a dialog). If NULL, an empty default layout is used.
- *application_data*—Specifies the application data stored for the page.

Use the function

ProUIDashboardpageoptionsNotificationSet() to assign a function to be called upon a certain event occurring in the dashboard. The input arguments for this function are:

- *options*—Specifies a handle representing a dashboard page.
- *notification*—Specifies the function to be called upon the designated event occurrence.
- *appdata*—Specifies the application data passed to the callback function when it is invoked.

You can register event notifications for the following events on the dashboard page:

- **PRO_UI_DASHBOARD_PAGE_CREATE**—when the page is first created.
- **PRO_UI_DASHBOARD_PAGE_SHOW**—when the page is shown (slide-down panels only).
- **PRO_UI_DASHBOARD_PAGE_HIDE**—when the page is hidden (slide-down panels only).

- **PRO_UI_DASHBOARD_PAGE_DESTROY**—when the page is destroyed.

You can use the function **ProUIDashboardpageoptionsFree()** to free a handle representing a single page (or layout) that will be shown in a dashboard.

Accessing a Dashboard Page

Functions Introduced:

- **ProUIDashboardPageGet()**
- **ProUIDashboardpageTitleSet()**
- **ProUIDashboardpageForegroundColorSet()**
- **ProUIDashboardpageNameGet()**
- **ProUIDashboardpageDashboardGet()**
- **ProUIDashboardpageUserdataGet()**
- **ProUIDashboardpageClose()**

Use the function **ProUIDashboardPageGet()** to obtain the handle to a given page from the dashboard. The input arguments for this function are:

- *dashboard*—Specifies the dashboard handle.
- *name*—Specifies the page name. Pass NULL to get the handle to the main page.

Use the function **ProUIDashboardpageTitleSet()** to assign the title string for the dashboard page. This will be shown as the button name for the slide-down panel. This should typically be called from the CREATE notification of the dashboard page.

Use the function **ProUIDashboardpageForegroundColorSet()** to set the text color for the button that invokes a slide-down panel. This technique is used in several Pro/ENGINEER tools to notify the user that they must enter one of the panels to complete the tool.

Use the function **ProUIDashboardpageNameGet()** to obtain the name of the page.

Use the function **ProUIDashboardpageDashboardGet()** to obtain the dashboard that owns this page.

Use the function **ProUIDashboardpageUserdataGet()** to obtain the application stored with this dashboard page on registration.

Use the function **ProUIDashboardpageClose()** to close the dashboard slide-down page.

Accessing Components in the Dashboard Pages

Components added to the dashboard are actually "owned" by the Pro/ENGINEER dialog window. Pro/ENGINEER automatically modifies the names of components loaded from resource files to ensure that no name collisions occur when the components are added. The functions in this section allow you to locate the names of components that you need to access.

Functions Introduced:

- **ProUIDashboardpageDevicenameGet()**
- **ProUIDashboardpageComponentnameGet()**

Use the function **ProUIDashboardpageDevicenameGet()** to obtain the device name owning the dashboard page. This name should be used in other ProUI functions to access the components stored in the dashboard page.

Use the function **ProUIDashboardpageComponentnameGet()** to obtain the real component name in the dashboard page, if the page contents were loaded from a layout. This name should be used in ProUI functions for accessing a component in the dashboard.

User Interface: Basic Graphics

This chapter describes all the functions provided by Pro/TOOLKIT that create and manipulate graphics and object displays.

Pro/TOOLKIT refers to windows using integer identifiers. The base window (the big graphics window created automatically when you enter Pro/ENGINEER) is window 0, and the text message window at the bottom is window 1.

In many of the functions in this section, you can use the identifier “-1” to refer to the current window (the one current to the Pro/ENGINEER user).

Topic	Page
Manipulating Windows	18 - 2
Solid Orientation	18 - 6
Graphics Colors and Line Styles	18 - 10
Displaying Graphics	18 - 15
Displaying Text	18 - 17
Display Lists	18 - 19
Getting Mouse Input	18 - 20
Cosmetic Properties	18 - 23

Manipulating Windows

This section describes how to manipulate windows using Pro/TOOLKIT. It is divided into the following subsections:

- Resizing Windows
- Manipulating the Embedded Browser in Windows
- Repainting Windows
- Controlling Which Window is Current
- Creating and Removing Windows
- Retrieving the Owner of a Window
- Visiting Windows
- Activating Windows

Resizing Windows

Functions Introduced:

- **ProWindowSizeGet()**
- **ProGraphicWindowSizeGet()**

The function **ProWindowSizeGet()** returns the size of the Pro/ENGINEER window including the User Interface border.

The function **ProGraphicWindowSizeGet()** returns the size of the Pro/ENGINEER graphics window without the border. If the window occupies the whole screen, the window size is returned as 1. If the screen is 1024 pixels wide and the window is 512 pixels, the width will be returned as 0.5.

Manipulating the Embedded Browser in Windows

Functions Introduced:

- **ProWindowBrowserSizeGet()**
- **ProWindowBrowserSizeSet()**
- **ProWindowURLShow()**
- **ProWindowURLGet()**

The functions **ProWindowBrowserSizeGet()** and **ProWindowBrowserSizeSet()** enable you to find and change size of the embedded browser in the Pro/ENGINEER window. These functions refer to the browser size in terms of a percentage of the graphics window (0.0 to 100.0).

The functions **ProWindowURLGet()** and **ProWindowURLShow()** enable you to find and change the URL displayed in the embedded browser in the Pro/ENGINEER window.

Repainting Windows

Functions Introduced:

- **ProWindowClear()**
- **ProWindowRepaint()**
- **ProWindowRefresh()**
- **ProWindowRefit()**
- **ProTreetoolRefresh()**

The function **ProWindowClear()** temporarily removes all graphics from the specified window. If you give the function a window identifier of -1, it clears the current window.

This function is *not* equivalent to the Pro/ENGINEER option to quit the window. It does not break the connection between the current solid and the window. That is the purpose of the function **ProWindowDelete()**, described later in this chapter.

The **ProWindowRepaint()** function is equivalent to the Pro/ENGINEER command **View, Repaint**, and removes highlights. The function accepts -1 as the identifier, which indicates the current view.

The function **ProWindowRefresh()** is designed primarily for the purposes of animation. It updates the display very efficiently, but does not remove highlights. The function accepts -1 as the window identifier, which indicates the current window.

The function **ProWindowRefit()** performs exactly the same action as the Pro/ENGINEER command **View Refit**. (It does not accept -1 as the current window. Use **ProWindowCurrentGet()** if you need the id of the current window.)

The function **ProTreetoolRefresh()** refreshes the display of the model tree for the specified model. This function is useful when you are modifying the model in some way, such as when you are creating patterns or features.

Controlling Which Window is Current

Functions Introduced:

- **ProWindowCurrentGet()**
- **ProWindowCurrentSet()**

The functions **ProWindowCurrentGet()** and **ProWindowCurrentSet()** enable you to find out and change the current window. The window is current only for the purposes of the other Pro/TOOLKIT commands that affect windows.

The function **ProWindowCurrentSet()** is *not* equivalent to the Pro/ENGINEER command to change the window and has no effect on the object returned by **ProMdlCurrentGet()**.

Creating and Removing Windows

Functions Introduced:

- **ProObjectwindowCreate()**
- **ProWindowDelete()**
- **ProWindowCurrentClose()**
- **ProAccessorywindowWithTreeCreate()**

The function **ProObjectwindowCreate()** opens a new window containing a specified solid. The solid must already be in memory. If a window is already open on that solid, the function returns the identifier of that window. If the Main Window is empty, the function uses it instead of creating a new one. The section Graphics Colors and Line Styles shows how to use **ProObjectwindowCreate()**.

The function **ProWindowDelete()** is the equivalent of the Pro/ENGINEER command to quit the window. If the window is not the Main Window, it is also deleted from the screen.

To close the current window, use the function **ProWindowCurrentClose()**. This function duplicates the behavior of the **Close** option on the Window pull-down menu from the Pro/ENGINEER menu bar. If the current window is the original window created when Pro/ENGINEER was started, the function clears the window; otherwise, the function removes the window from the screen.

Note: Any work done since the last save will be lost.

In Pro/ENGINEER, when the main window is active, you can open an accessory window for operations such as edit an inserted component or a feature, preview an object, select a reference, and some other operations. The model tree associated with this Pro/ENGINEER object is also displayed in the accessory window. For more information about the accessory window, refer the Pro/ENGINEER help.

In Pro/TOOLKIT, the function **ProAccessorywindowWithTreeCreate()** opens an accessory window containing the specified object. If a window is already open with the specified object, the function returns the identifier of that window. If an empty window exists, the function uses that window to open the object. The input argument *tree_flag* controls the display of the model tree in the accessory window. If this flag is set to PRO_B_TRUE the model tree is displayed. The function **ProAccessorywindowWithTreeCreate()** supersedes the function **ProAccessorywindowCreate()**.

Retrieving the Owner of a Window

Function Introduced:

- **ProWindowMdlGet()**

The function **ProWindowMdlGet()** retrieves the Pro/ENGINEER model that owns the specified window. This function gives you details about the window needed to perform necessary actions on it.

Note: If no model is associated with the specified window, Pro/TOOLKIT returns NULL as a model pointer and PRO_TK_NO_ERROR as a return value.

Visiting Windows

Function Introduced:

- **ProWindowsVisit()**

The function **ProWindowsVisit()** enables you to visit all the Pro/ENGINEER windows. For a detailed explanation of visiting functions, see the section Visit Functions in the Fundamentals chapter.

Activating Windows

Function Introduced:

- **ProWindowActivate()**

The function **ProWindowActivate()** activates the specified window and sets it as the current window. It does not take effect until control returns to Pro/ENGINEER.

This functionality is equivalent to changing the active window by selecting and activating a window using the Window pull-down menu from the Pro/ENGINEER menu bar.

Note: This function works in asynchronous mode only.

Solid Orientation

Functions Introduced:

- **ProWindowCurrentMatrixGet()**
- **ProViewMatrixGet()**
- **ProViewMatrixSet()**
- **ProViewReset()**
- **ProViewRotate()**
- **ProWindowPanZoomMatrixSet()**

Each graphics window in solid (Part or Assembly) mode has two transformation matrices associated with it—the view matrix and the window matrix. The view matrix describes the transformation between solid coordinates and screen coordinates. Therefore, the view matrix describes the orientation of the solid in the window.

The window matrix is the transformation between screen coordinates and window coordinates. The window matrix describes the pan and zoom factors. The screen coordinate at which a particular point on a solid is displayed is not affected by pans and zooms—this affects window coordinates only.

The view matrix is important because the mouse input functions and some of the graphics drawing functions use screen coordinates, while all solid geometry uses solid coordinates. The view matrix enables you to transform between the two systems. The function **ProWindowCurrentMatrixGet()** provides the window matrix for the current window.

The function **ProWindowPanZoomMatrixSet()** enables you to set the pan and zoom matrix (window matrix) for the current window.

Getting and Setting the View Matrix

The function **ProViewMatrixGet()** provides the view matrix for a specified window. Set the view argument to NULL for the current view.

The function **ProViewMatrixSet()** enables you to set the view matrix (if normalized), and therefore the orientation of the solid in the view.

Note: Function **ProViewMatrixSet()** does not cause the view to be repainted.

A 4x4 transformation matrix describes a shift and a scaling, as well as a reorientation. You set the view matrix to define a new orientation. Pro/ENGINEER applies its own shift and scaling to the view matrix you provide to ensure that the solid fits properly into the view. This implies the following:

- The matrix output by **ProViewMatrixGet()** is not the same as the one you previously input to the function **ProViewMatrixSet()**, although its orientation is the same.
- Each row of the matrix you provide to **ProViewMatrixSet()** must have a length of 1.0, and the bottom row must be 0, 0, 0, 1.
- The matrix you provide to **ProViewMatrixSet()** must be normalized—it cannot include scaling or shift. Example 1: Saving Three Views shows how to normalize a matrix.

Storing Named Views

Functions Introduced:

- **ProViewStore()**
- **ProViewRetrieve()**
- **ProViewNamesGet()**

The **ProViewStore()** and **ProViewRetrieve()** functions enable you to save and use a named view of the solid. They are equivalent to the Pro/ENGINEER commands **View**, **Saved Views**, **Save**, and **View**, **Saved Views**, then selecting the view you want from the list of view names.

The function **ProViewNamesGet()** retrieves the names of the views in the specified solid.

Example 1: Saving Three Views

This example shows how to set the view matrix and store a named view.

```
/*=====*\
FUNCTION: UserMatrixNormalize
PURPOSE:  Normalize a 4x4 transformation matrix.
\*=====*/
void UserMatrixNormalize(
    ProMatrix  m
)
{
    int  row, col;
    double scale;

/*-----*\
    Get the scaling factor.
\*-----*/
    scale = sqrt (m[0][0] * m[0][0] +
                  m[0][1]*m[0][1] +
                  m[0][2]*m[0][2]);

/*-----*\
    Remove the scaling.
\*-----*/
    for (row = 0; row < 3; row++)
        for (col = 0; col < 3; col++)
            m[row][col] /= scale;

/*-----*\
    Remove the shift.
\*-----*/
    m[3][0] = 0.0;
    m[3][1] = 0.0;
    m[3][2] = 0.0;
}
/*=====*\
FUNCTION: UserViews
PURPOSE:  Stores three orthogonal views called FRONT, SIDE,
          and TOP.
\*=====*/
int UserViews()
{
    ProMatrix  vmatrix, matrix;
    ProName    name;
    ProMode    mode;

    int  i, j;
    ProError  err;

/*-----*\
    This test should only be run in Part or Assembly mode.
\*-----*/
```



```

    err = ProModeCurrentGet (&mode);
    if (mode != PRO_PART && mode != PRO_ASSEMBLY)
        return (0);
/*-----*\
    Remember the current view matrix, and normalize it.
\*-----*/
    err = ProViewMatrixGet (NULL, NULL, vmatrix);
    UserMatrixNormalize (vmatrix);
/*-----*\
    Store the current view (so you can recover it after picking
    the following).
\*-----*/
    ProStringToWstring (name, "ORIGINAL");
    err = ProViewStore (NULL, name, NULL);
/*-----*\
    Store the FRONT view (x = x, y = y - unitary view matrix).
\*-----*/
    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
            matrix[i][j] = 0;
    matrix[3][3] = 1;
    matrix[0][0] = matrix[1][1] = matrix[2][2] = 1;

    err = ProViewMatrixSet (NULL, NULL, matrix);
    ProStringToWstring (name, "FRONT");
    err = ProViewStore (NULL, name, NULL);
/*-----*\
    Store the SIDE view (x = y, y = z).
\*-----*/
    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
            matrix[i][j] = 0;
    matrix[3][3] = 1;
    matrix[0][2] = matrix[1][0] = matrix[2][1] = 1;

    err = ProViewMatrixSet (NULL, NULL, matrix);
    ProStringToWstring (name, "SIDE");
    err = ProViewStore (NULL, name, NULL);
/*-----*\
    Store the TOP view (x = z, y = x).
\*-----*/
    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
            matrix[i][j] = 0;
    matrix[3][3] = 1;
    matrix[0][1] = matrix[1][2] = matrix[2][0] = 1;

    err = ProViewMatrixSet (NULL, NULL, matrix);
    ProStringToWstring (name, "TOP");
    err = ProViewStore (NULL, name, NULL);
/*-----*\

```

```

    Reset the view matrix.
/*-----*/
    ProViewMatrixSet (NULL, NULL, vmatrix);
    return (0);
}

```

Graphics Colors and Line Styles

Pro/ENGINEER uses several predefined colors in its color map. The colors are represented by the values of **ProColorType()**, defined in the file ProToolkit.h. The names of the types generally indicate what they are used for in Pro/ENGINEER, although many colormap entries are used for several different purposes. These also correspond to the system colors presented to the user through the user interface.

Note: PTC reserves the right to change both the definitions of the predefined colormap and also of the assignment of entities to members of the color map as required by improvements to the user interface. PTC recommends not relying on the predefined RGB color for displaying of Pro/TOOLKIT entities or graphics, and also recommends against relying on the relationship between certain colormap entries and types of entities. The following section describe how to construct your application so that it does not rely on potentially variant properties in Pro/ENGINEER.

Setting Colors to Desired Values

Functions Introduced:

- **ProTextColorModify()**
- **ProGraphicsColorModify()**

The functions **ProTextColorModify()** and **ProGraphicsColorModify()** enable you to select a different color to be used for either of the following:

- Graphics text—User custom text drawn by the ProGraphicsTextDisplay. Graphics text is by default displayed using PRO_COLOR_LETTER.
- Graphics—User custom graphics drawn by ProGraphics function, which is by default displayed using PRO_COLOR_DRAWING.

Both functions only affect the color used for new graphics, you draw using Pro/TOOLKIT, not the colors used for items Pro/ENGINEER draws.

Both functions take a ProColor structure as input. This structure allows you to specify color by one of the following three methods:

- **DEFAULT**—use the default Pro/ENGINEER color entry for new graphics or text.
- **TYPE**—use a predefined ProColor type color.
- **RGB** - use a custom RGB value. This method should be used for any graphics which should not be allowed to change color (for example; if an application wants a yellow line on the screen that should always be yellow and not depend on the chosen color scheme).

Both functions output the value of the previous setting. It is good practice to return the color to its previous value after having finished drawing an object.

Setting Colors to Match Existing Entities

Functions Introduced:

- **ProColorByTypeGet()**

The functions **ProGraphicsColorModify()** and **ProColorTextModify()** allow you to draw graphics that will change color based on changes to the Pro/ENGINEER colormap. This allows you to draw entities in similar colors to related entities created by Pro/ENGINEER. However, if the associations between objects and colormap entries should change in a new release of Pro/ENGINEER, the association between the application entities and the Pro/ENGINEER entities would be lost. The function **ProColorByTypeGet()** returns the standard colormap entry corresponding to a particular entity in Pro/ENGINEER. This allows applications to draw graphics that will always match the color of a particular Pro/ENGINEER entity.

Example 2: Setting the Graphics Color to a Specific RGB Value

```
ProError UserDrawYellow(char* dummy, int dummy_2)
{
    ProColor yellow;
    ProColor old_g_color, old_t_color;
    ProError status;
    ProFileName msgfil;
```

```

ProStringToWstring(msgfil, "msg Uggraph.txt");

yellow.method = PRO_COLOR_METHOD_RGB;
yellow.value.map.red = 1.0;
yellow.value.map.green = 1.0;
yellow.value.map.blue = 0.0;

status = ProGraphicsColorModify (&yellow, &old_g_color);
ERROR_CHECK ("UserDrawYellow", "ProGraphicsColorModify", status);

status = ProTextColorModify (&yellow, &old_t_color);
ERROR_CHECK ("UserDrawYellow", "ProTextColorModify", status);

/* In a typical implementation, the application would draw graphics here,
   and then reset the color back to the previous color.

   For this example: users should choose one of the other pt_userguide
   menu buttons to observe the results of this change.*/

ProMessageDisplay (msgfil, "USER Color set to yellow");
return status;
}

```

Example 3: Setting The Graphics Color to Follow the Color of Pro/ENGINEER Entity

```

ProError UserDrawWithDatumAxisColor(char* dummy, int dummy_2)
{
    ProColor dtm_color;
    ProColor old_g_color, old_t_color;
    ProError status;
    ProFileName msgfil;
    ProStringToWstring(msgfil, "msg Uggraph.txt");

    dtm_color.method = PRO_COLOR_METHOD_TYPE;

    status = ProColorByTypeGet (PRO_DISP_DTM_AXIS, &dtm_color.value.type);
    ERROR_CHECK ("UserDrawWithDatumAxisColor", "ProColorByTypeGet", status);

    status = ProGraphicsColorModify (&dtm_color, &old_g_color);
    ERROR_CHECK ("UserDrawWithDatumAxisColor",
        "ProGraphicsColorModify", status);
    status = ProTextColorModify (&dtm_color, &old_t_color);
    ERROR_CHECK ("UserDrawWithDatumAxisColor", "ProTextColorModify",
        status);

    /* In a typical implementation, the application would draw graphics here,
       and then reset the color back to the previous color.

```

```

    For this example: users should choose one of the other pt_userguide
    menu buttons to observe the results of this change. */

    ProMessageDisplay (msgfil, "USER Color set to datum axis");
    return status;
}

```

Modifying the Pro/ENGINEER Color Map

Functions Introduced:

- **ProColormapGet()**
- **ProColormapSet()**

These functions enable you to find out and alter the color settings for Pro/ENGINEER. Each color is defined in terms of the red, green, and blue values. The RGB values should be expressed in a range from 0.0 to 1.0. Note that some colors related to selection and highlighting are fixed and may not be modified.

Pro/ENGINEER uses these colors for everything it displays.

Changes to the color map are preserved for the rest of the Pro/ENGINEER session. If you want to have permanent changes, call **ProColormapSet()** in **user_initialize()**.

Note: Changing the Pro/ENGINEER color map can have unintended effects if the user has chosen an alternate color scheme. It may cause certain entries to blend into the background or to be confused with other types of entries.

Example 4: Modifying the Color of the HALF_TONE Display

This example shows how to modify the default half-tone color (gray) to a brighter color, using the function **ProColormapSet()**. Pro/ENGINEER uses half-tone color to display hidden lines.

```

UserAdjustHalftone (dummy, lighter)
    char    *dummy;
    int     lighter;
{
    ProColormap colors;
    ProError  err;

    err = ProColormapGet (PRO_COLOR_HALF_TONE, &colors);
    if (lighter)
    {
        colors.red    += 0.1;
        colors.green += 0.1;
    }
}

```

```

        colors.blue += 0.1;
    }
    else
    {
        colors.red -= 0.1;
        colors.green -= 0.1;
        colors.blue -= 0.1;
    }

    err = ProColormapSet (PRO_COLOR_HALF_TONE, &colors);
    err = ProWindowRefresh (-1);
}

```

Pro/ENGINEER Color Schemes

Functions Introduced:

- **ProColormapAlternateschemeGet()**
- **ProColormapAlternateschemeSet()**

The functions **ProColormapAlternateschemeGet()** and **ProColormapAlternateschemeSet()** enable you to change the color scheme of Pro/ENGINEER to a predefined color scheme by turning on and off the blended background (light to dark grey) for alternate schemes. The possible alternate color schemes are as follows:

- **PRO_COLORMAP_ALT_BLACK_ON_WHITE**—Displays black entities shown on a white background.
- **PRO_COLORMAP_ALT_WHITE_ON_BLACK**—Displays white entities shown on a black background.
- **PRO_COLORMAP_ALT_WHITE_ON_GREEN**—Displays white entities shown on a dark green background.
- **PRO_COLORMAP_OPTIONAL1**—Represents the color scheme with a dark background.
- **PRO_COLORMAP_OPTIONAL2**—Represents the color scheme with a medium background.
- **PRO_COLORMAP_CLASSIC_WF**—Resets the color scheme to the light to dark grey background (default upto Pro/ENGINEER Wildfire 4.0).
- **PRO_COLORMAP_ALT_DEFAULT**—Resets the color scheme to the default color scheme of light to dark blue gradient background (from Pro/ENGINEER Wildfire 5.0 onwards).

Setting Line Styles for Pro/TOOLKIT Graphics

Functions Introduced:

- **ProLinestyleSet()**
- **ProLinestyleDataGet()**
- **ProGraphicsModeSet()**

The function **ProLineStyleSet()** enables you to set the style of graphics you draw. The function **ProLinestyleDataGet()** queries the definition of the line style.

The possible values for the line style are as follows:

- **PRO_LINestyle_SOLID**—Solid line
- **PRO_LINestyle_DOT**—Dotted line
- **PRO_LINestyle_CENTERLINE**—Alternating long and short dashes
- **PRO_LINestyle_PHANTOM**—Alternating long dashes and two dots

Displaying Graphics

Functions Introduced:

- **ProGraphicsPenPosition()**
- **ProGraphicsLineDraw()**
- **ProGraphicsPolylineDraw()**
- **ProGraphicsMultiPolylinesDraw()**
- **ProGraphicsArcDraw()**
- **ProGraphicsCircleDraw()**
- **ProGraphicsPolygonDraw()**

All the functions in this section draw graphics in the current window (the Pro/ENGINEER current window, unless redefined by a call to **ProWindowCurrentSet()**), and use the color and line style set by calls to **ProGraphicsColorSet()** and **ProLineStyleSet()**. The functions draw the graphics in the Pro/ENGINEER graphics color. The default graphics color is white.

By default, the graphics elements are not stored in the Pro/ENGINEER display list, so they do not get redrawn by Pro/ENGINEER when the user selects **View**, **Repaint** or **View, Orientation**. However, if you store graphics elements in either 2-D or 3-D display lists, Pro/ENGINEER redraws them. See the section Display Lists for more information.

The functions **ProGraphicsPenPosition()** and **ProGraphicsLineDraw()** draw three-dimensional polylines in solid mode, and take solid coordinates.

The function **ProGraphicsPenPosition()** sets the point at which you want to start drawing the line. The function **ProGraphicsLineDraw()** draws a line to the given point from the position given in the last call to either of the two functions. You call **ProGraphicsPenPosition()** for the start of the polyline, and **ProGraphicsLineDraw()** for each vertex.

If you use these functions in Drawing mode they work correctly, but use screen coordinates instead of solid coordinates.

The **ProGraphicsPolylineDraw()** and **ProGraphicsMultiPolylinesDraw()** functions also draw polylines, but you need to have the whole polyline defined in a local array before you call either function. If you are drawing many lines, use **ProGraphicsMultipolylinesDraw()** to minimize the number of function calls.

The function **ProGraphicsArcDraw()** draws an arc, in screen coordinates.

The function **ProGraphicsCircleDraw()** uses solid coordinates for the center of the circle and the radius value, but draws the circle parallel to the plane of the window. You can position the circle at a chosen solid vertex, for example, and the circle will always be clearly visible as a circle, regardless of the current solid orientation.

Example 5: Displaying Lines and Circles

In this example, the user selects a series of points on a part surface. Pro/TOOLKIT draws a circle at each point and draws lines between the points.

```
UserDraw()  
{  
    ProFileName      msgfil;  
    ProColortype      old_color, dummy_color;  
    ProMouseButton    button;  
    ProPoint3d         point;  
    ProError           err;
```



```

    int                first = 1;

    ProStringToWstring (msgfil, "msg_ug6.txt");
/*-----*\
    Set the Pro/TOOLKIT graphics color to MAGENTA.
/*-----*\
    err = ProGraphicsColorSet (PRO_COLOR_ERROR, &old_color);
/*-----*\
    Tell the user to select the points.
/*-----*\
    ProMessageDisplay (msgfil, "USER Select points");
    while (ProMousePickGet (NULL, &button, point) == PRO_TK_NO_ERROR
           && button == PRO_LEFT_BUTTON)
    {
/*-----*\
        Draw a small circle at the point.
/*-----*\
        ProGraphicsCircleDraw (point, 0.1);
/*-----*\
        For the first point, move there. At subsequent points,
        draw a line from the previous point to the current one.
/*-----*\
        if (first)
        {
            ProGraphicsPenPosition (point);
            first = 0;
        }
        else
            ProGraphicsLineDraw (point);
    }
/*-----*\
    Set the color back.
/*-----*\
    err = ProGraphicsColorSet (old_color, &dummy_color);
}

```

Displaying Text

Function Introduced:

- **ProGraphicsTextDisplay()**

The function **ProGraphicsTextDisplay()** places text, specified as a wide string, at a position specified in screen coordinates. Therefore, if you want to add text to a particular position on the solid, you must transform the solid coordinates into screen coordinates by using the view matrix.

Like the graphics polylines, arcs, and so on (added by the functions described in the section Displaying Graphics), the text items drawn by **ProGraphicsTextDisplay()** are not known to Pro/ENGINEER, and therefore are not redrawn when you select **View, Repaint**. Use the notify or display list functions to tell Pro/ENGINEER about the items. See the section Display Lists for more information on the display list functions. To add permanent text to a drawing (for example, a drawing note), see the Drawings.

Controlling Text Attributes

Functions Introduced:

- **ProTextAttributesCurrentGet()**
- **ProTextFontIdCurrentSet()**
- **ProTextHeightCurrentSet()**
- **ProTextRotationAngleCurrentSet()**
- **ProTextSlantAngleCurrentSet()**
- **ProTextWidthFactorCurrentSet()**

These functions control the attributes of text added by calls to **ProGraphicsTextDisplay()**. You can get and set the following information:

- The font identifier
- The text height, in screen coordinates
- The ratio of the width of each character (including the gap) as a proportion of the height
- The angle of rotation of the whole text, in counterclockwise degrees
- The angle of slant of the text, in clockwise degrees

Controlling Text Fonts

Functions Introduced:

- **ProTextFontDefaultIdGet()**
- **ProTextFontNameGet()**
- **ProTextFontNameTold()**

The function **ProTextFontDefaultIdGet()** returns the identifier of the default Pro/ENGINEER text font.

The text fonts are identified in Pro/ENGINEER by names, and in Pro/TOOLKIT by integer identifiers. To move between the two types of font identifiers, use the functions

ProTextFontNameGet() and **ProTextFontNameToId()**.

Because the internal font identifiers could change between Pro/ENGINEER sessions, it is important to call

ProTextFontNameToId() each time you want to modify the font in Pro/TOOLKIT.

Display Lists

Functions Introduced:

- **ProDisplist2dCreate()**
- **ProDisplist2dDisplay()**
- **ProDisplist2dDelete()**
- **ProDisplist3dCreate()**
- **ProDisplist3dDisplay()**
- **ProDisplist3dDelete()**

To generate the display of a solid in a window, Pro/ENGINEER maintains two display lists. A display list contains a set of vectors used to represent the shape of the solid in the view.

The 3-D display list contains a set of three-dimensional vectors that represent an approximation to the geometry of the edges of the solid. It gets rebuilt every time the solid is regenerated.

The 2-D display list contains the two-dimensional projections of the 3-D display list onto the current window. It is rebuilt from the 3-D display list when the orientation of the solid changes.

The functions in this section enable you to add your own vectors to the display lists, so your own graphics will be redisplayed automatically by Pro/ENGINEER, until the display lists are rebuilt.

For example, if you just use the functions described in the section Displaying Graphics, the items you draw remain on the screen only until the window is repainted. Furthermore, the objects are not plotted with the rest of the contents of the window because Pro/ENGINEER does not know about them.

If, however, you add the same graphics items to the 2-D display list, they will survive each repaint (when zooming and panning, for example) and will be included in plots created by Pro/ENGINEER.

If you add graphics to the 3-D display list, you get the further benefit that the graphics survive a change to the orientation of the solid and are displayed, even when you spin the solid dynamically.

To add graphics to a display list, you must write a function that displays the necessary vectors *in three dimensions*, using the graphics display functions in the usual way. For the 2-D display list, you call **ProDisplist2dCreate()** to tell Pro/ENGINEER to use your function to create the display list vectors, then call **ProDisplist2dDisplay()** to ask it to display the new graphics those vectors represent.

Note: If you save the display information, you can reuse it in any session. The application should delete the display list data when it is no longer needed.

Using 3-D display lists is exactly analogous to using 2-D display lists.

Note that the function **ProWindowRefresh()** does not cause either of the display lists to be regenerated, but simply repaints the window using the 2-D display list.

The function **ProSolidDisplay()** regenerates both display lists, and therefore not only recenters the solid in the view and removes any highlights, but also removes any graphics you added using the 2-D display list functions.

Getting Mouse Input

Functions Introduced:

- **ProMousePickGet()**
- **ProMouseTrack()**
- **ProMouseBoxInput()**

The functions **ProMousePickGet()** and **ProMouseTrack()** are used to read the position of the mouse, in screen coordinates. Each function outputs the position and an enumerated type description of which mouse buttons were pressed when the mouse was at that position. The values are defined in *ProGraphic.h*, and are as follows:

- **PRO_LEFT_BUTTON**

- PRO_MIDDLE_BUTTON
- PRO_RIGHT_BUTTON

The function **ProMousePickGet()** reports the mouse position only when you press a button. It has an input argument that is a description of the mouse buttons you want to wait for (you must have at least one). For example, you can set the *expected_button* argument to:

```
PRO_LEFT_BUTTON | PRO_RIGHT_BUTTON
```

In this example, the function does not return until you press either the left or right button. You could also specify the value PRO_ANY_BUTTON.

The function **ProMouseTrack()** returns whenever the mouse is moved, regardless of whether a button is pressed. Therefore, the function can return the value PRO_NO_BUTTON. Its input argument enables you to control whether the reported positions are snapped to grid.

Example 6: Drawing a Rubber-Band Line

This example shows how to use the complement drawing mode. The example follows the mouse position dynamically.

```
/*=====*\
FUNCTION:  UserGetLine
PURPOSE :  When the user inputs a point, p2, the function draws
           a rubber-band line from p1 to p2.
\*=====*/
UserGetLine (ProPoint3d p1, ProPoint3d p2)
{
    ProMouseButton  button;
    ProPoint3d      op2;
    ProDrawMode     old_graphics_mode;
    ProError        err;

/*-----*\
    Set the drawing mode to COMPLEMENT.
\*-----*/
    err = ProGraphicsModeSet (PRO_DRAW_COMPLEMENT_MODE,
                             &old_graphics_mode);

/*-----*\
    Start a line from the input position.
\*-----*/
    ProGraphicsPenPosition (p1);

/*-----*\
    Get the first sample mouse position, and draw a line to it.
\*-----*/
```

```

    err = ProMouseTrack (0, &button, p2);
    ProGraphicsLineDraw (p2);
/*-----*\
    Remember the position.
/*-----*/
    op2[0] = p2[0];
    op2[1] = p2[1];
    op2[2] = p2[2];

    while (1)
    {
/*-----*\
        If the user hit a button...
/*-----*/
        if (button)
            break;
/*-----*\
        Sample the mouse.
/*-----*/
        err = ProMouseTrack (0, &button, p2);
/*-----*\
        Remove the previous rubber-band line. (Just redraw the
        line to the previous position.)
/*-----*/
        ProGraphicsPenPosition (p1);
        ProGraphicsLineDraw (op2);
        op2[0] = p2[0];
        op2[1] = p2[1];
        op2[2] = p2[2];
/*-----*\
        Draw a line to the new position.
/*-----*/
        ProGraphicsPenPosition (p1);
        ProGraphicsLineDraw (p2);
    }
/*-----*\
    Remove the rubber-band line.
/*-----*/
    ProGraphicsLineDraw (p1);
/*-----*\
    Set the drawing mode back to SET.
/*-----*/
    ProGraphicsModeSet (PRO_DRAW_SET_MODE, &old_graphics_mode);

    return (button);
}

```

Cosmetic Properties

You can enhance your model using Pro/TOOLKIT functions that change the surface properties, or set different light sources. The following sections describe these functions in detail.

Surface Properties

Functions Introduced:

- **proddb_get_surface_props()**
- **ProTexturePathGet()**
- **proddb_set_surface_props()**
- **proddb_unset_surface_props()**
- **proddb_surface_tessellation()**
- **ProPartTessellate()**
- **ProPartTessellationFree()**

The surface properties functions enable you to set and retrieve the surface properties for the specified element using the *Pro_surf_props* data structure.

The data structure is defined as follows:

```
typedef struct pro_surf_props
{
    double          ambient;
    double          diffuse;
    double          highlite;
    double          shininess;
    int             threshold;
    double          transparency;
    double          color_rgb[3];
    double          highlight_color[3];
    double          reflection;
    double          decal_intensity;      /* future use only */
    char            *decal;
    char            *texture_map;
    char            *bump_map;
    double          bump_height;          /* future use only */
    Projection_type projection;
    double          local_csys[4][3];
    int             copies;               /* future use only */
    float           horizontal_offset;
    float           vertical_offset;
    float           rotate;
}
```

```

float          horizontal_scale;
float          vertical_scale;
} Pro_surf_props;

```

The *Pro_surf_props* data structure contains the following fields:

- *ambient*—Specifies the indirect, scattered light the model receives from its surroundings. The valid range is 0.0 to 1.0.
- *diffuse*—Specifies the reflected light that comes from directional, point, or spot lights. The valid range is 0.0 to 1.0.
- *highlite*—Specifies the intensity of the light reflected from a highlighted surface area. The valid range is 0.0 to 1.0.
- *shininess*—Specifies the properties of a highlighted surface area. A plastic model would have a lower shininess value, while a metallic model would have a higher value. The valid range is 0.0. to 1.0.
- *threshold*—Enables you to selectively remove surfaces from view based on this value. Any surface that has a threshold higher than the *threshold* field will not be displayed. The valid range is 0 to 100.
- *transparency*—Specifies the transparency value, which is between 0 (completely opaque) and 1.0 (completely transparent).
- *color_rgb[3]*—Specifies the color, in terms of red, green, and blue. The valid range is 0.0. to 1.0.
- *highlight_color*—Specifies the highlight color, in terms of red, green, and blue. The valid range is 0.0 to 1.0.
- *reflection*—Specifies how reflective the surface is. The valid range is 0 (dull) to 100 (shiny).
- *decal*—Specifies the full path to the texture map with the alpha channel (transparency). Otherwise, use NULL.
- *texture_map*—Specifies the full path to the texture map.
- *bump_map*—Specifies the full path to the bump map. A bump map enables you to create bumps on the surface of the texture map.
- *projection*—Specifies the projection type—planar, spherical, or cylindrical.
- *local_csys*—Specifies the direction (for planar projection), or the whole coordinate system (which defines the center for the other projection types).

- *horizontal_offset* and *vertical_offset*—Specifies the percentage of horizontal and vertical shift of the texture map on the surface.
- *rotate*—Specifies the angle to rotate the texture map on the surface.
- *horizontal_scale* and *vertical_scale*—Specifies the horizontal and vertical scaling of the texture map.

In cosmetic shade mode, Pro/ENGINEER tessellates each surface by breaking it into a set of connected triangular planes. The Pro/DEVELOP-style function **prodb_surface_tessellation()** invokes this algorithm on a single specified surface and provides the coordinates of the triangle corners and the normal at each vertex. This function tessellates a single surface only.

Note: The function **ProSurfaceTessellationGet()** supersedes the function **prodb_surface_tessellation()**. Refer to chapter “Core: 3D Geometry” for more information.

The function **prodb_get_surface_props()** returns only the file name and extension for the options *decal*, *bump_map*, and *texture_map*. To retrieve the full path to any of these files use the function **ProTexturePathGet()**.

Refer to the Tessellation section of the Core: 3D Geometry chapter for functions **ProPartTessellate()** and **ProPartTessellationFree()**.

Setting Light Sources

Functions Introduced:

- **pro_get_light_sources()**
- **pro_set_light_sources()**

These functions retrieve and set the information associated with the specified window, respectively, using the *Pro_light* data structure. The data structure is defined as follows:

```
typedef struct pro_light
{
    wchar_t          name[PRODEV_NAME_SIZE];
    Pro_light_type    type;
    int               status;
    double            rgb[3];
}
```

```

double          position[3];
double          direction[3];
double          spread_angle;
int             cast_shadows;
} Pro_light;

```

The *Pro_light* structure contains the following fields:

- *name*—Specifies the name of the light source.
- *type*—Specifies the light type—ambient, direction, point, or spot.
- *status*—Specifies whether the light source is active or inactive.
- *rgb*—Specifies the red, green, and blue values, regardless of the light type.
- *position*—Specifies the position of the light, for point and spot lights only.
- *direction*—Specifies the direction of the light, for direction and spot lights only.
- *spread_angle*—Specifies the angle the light is spread, for spot lights only.
- *cast_shadows*—Specifies whether the light casts shadows. This applies to Pro/PHOTORENDER only. for more information on Pro/PHOTORENDER.

19

User Interface: Selection

This chapter contains functions that enable you to select objects in Pro/ENGINEER from within the Graphics Window or the Model Tree using the mouse or the keyboard.

Topic	Page
The Selection Object	19 - 2
Interactive Selection	19 - 5
Highlighting	19 - 5
Selection Buffer	19 - 10

The Selection Object

Like *ProModelitem*, the object *ProSelection* identifies a model item in the Pro/ENGINEER database. *ProSelection*, however, contains more information than *ProModelitem*, and is therefore sometimes used instead of *ProModelitem* in situations where the extra information is needed. The most important use of *ProSelection* is as the output of the function for interactive selection, **ProSelect()** (thus the name *ProSelection*).

ProSelection is declared as an opaque pointer, and is, strictly speaking, a WHandle because, although the model item is a reference to a Pro/ENGINEER database item, the other information is not.

Functions Introduced:

- **ProSelectionAlloc()**
- **ProSelectionSet()**
- **ProSelectionCopy()**
- **ProSelectionFree()**
- **ProSelectionAsmcompopathGet()**
- **ProSelectionModelitemGet()**
- **ProSelectionUvParamGet()**
- **ProSelectionViewGet()**
- **ProSelectionPoint3dGet()**
- **ProSelectionDepthGet()**
- **ProSelectionVerify()**
- **ProSelectionWindowIdGet()**
- **ProSelectionUvParamSet()**
- **ProSelectionViewSet()**
- **ProSelectionPoint3dSet()**
- **ProSelectionDrawingGet()**
- **ProSelectionDwgtblcellGet()**

Unpacking a ProSelection Object

For each item of information that *ProSelection* can contain, there is a Pro/TOOLKIT function that extracts that information. The following table lists these items.

Pro/TOOLKIT Function	Pro/TOOLKIT Object	Meaning
ProSelectionAsmcomppathGet()	<i>ProAsmcomppath</i>	Assembly component path
ProSelectionModelitemGet()	<i>ProModelitem</i>	Model item
ProSelectionPoint3dGet()	<i>ProPoint3d</i>	3-D point on the model item
ProSelectionUvParamSet()	<i>ProUvParam</i>	<i>u</i> and <i>v</i> , or <i>t</i> , of that point
ProSelectionDepthGet()	double	Selection depth
ProSelectionViewGet()	<i>ProView</i>	Drawing view in which the selection was made
ProSelectionWindowIdGet()	int	Window where a selection is done
ProSelectionDrawingGet()	<i>ProDrawing</i>	Drawing in which the selection was made.
ProSelectionDwgtblcellGet()	several integers	Table segment, row, and column of a selected drawing table cell

The assembly component path is the path down from the root assembly to the model that owns the database item being referenced. It is represented by the object *ProAsmcomppath* and is described fully in the Assembly: Basic Assembly Access section.

The model item describes the database item in the context of its owning model, but does not refer to any parent assembly.

The 3-D point is the location, in solid coordinates, of a selected point on the model item, if it is a geometry object. The solid coordinates are those of the solid directly owning the model item.

If the model item is a surface, *ProUvParam* contains the *u* and *v* values that correspond to the 3-D selection point described above. If the item is an edge or curve, *ProUvParam* contains the *t* value.

The selection depth is the distance between the selected point and the point from which the selection search started. This is important only when you are using **ProSolidRayIntersectionCompute()**, described in the section Ray Tracing.

The view is used to distinguish different views of a solid in a drawing.

Building a ProSelection Object

Some Pro/TOOLKIT function require a ProSelection object as an input. In many cases the assembly path—*ProAsmcomppath*—and the modelitem will be all that is needed, so **ProSelectionAlloc()** or **ProSelectionSet()** can be used. In other cases, for example when a ProSelection needs to identify a specific drawing view, or a specific location on a geometry item, you may also need to call functions **ProSelectionViewSet()**, **ProSelectionUvParamSet()**, and **ProSelectionPoint3dSet()**.

ProSelection Function Examples

Examples of Pro/TOOLKIT functions that use *ProSelection* are as follows:

- **ProSelect()** uses *ProSelection* as its output to describe everything about the selected item.
- **ProGeomitemDistanceEval()** uses *ProSelection* as its input, instead of *ProGeomitem*, so it can measure the distance between model items in different subassemblies.
- **ProSelectionHighlight()** and **ProSelectionUnhighlight()** use *ProSelection* as inputs to distinguish different instances of the same model item in different subassemblies, and also different drawing views of the same model.
- **ProFeatureCreate()** usually uses ProSelection objects to identify the geometry items the feature needs to reference.
- **ProDrawingDimCreate()** uses ProSelection objects to identify the entities the dimension will attach to and the drawing view in which the dimension is to be displayed.

In a case such as **ProGeomitemDistanceEval()**, which uses *ProSelection* as an input, you might need to build a *ProSelection* object out of its component data items. The function **ProSelectionAlloc()** allocates a new *ProSelection* structure, and sets the *ProAsmcomppath* and *ProModelitem* data in it. The function **ProSelectionSet()** sets that information in a *ProSelection* object that already exists. The function **ProSelectionVerify()** checks to make sure the contents of a *ProSelection* are consistent.

The function **ProSelectionCopy()** copies one *ProSelection* object to another. **ProSelectionFree()** frees the memory of a *ProSelection* created by **ProSelectionAlloc()** or as output by a Pro/TOOLKIT function.

Interactive Selection

Function Introduced:

- **ProSelect()**

ProSelect() is the Pro/TOOLKIT function that forces the user to make an interactive graphics selection in Pro/ENGINEER. Using this function the user can specify filters which control the items that can be selected.

Typically, the user has control over the filter options available from the **filter** menu located in the status bar at the bottom of the Pro/ENGINEER graphics window. A call to **ProSelect()** sets application desired filters for the next expected selection.

The user interface shown when prompting for an interactive selection will be the selection dialog, with **OK** or **Cancel** buttons. Depending on the selection types permitted the user will be able to select items via:

- The graphics window
- The model tree
- The search tool

None of these selections or possible selections will provide guidance to the user about what to select and why. Therefore, it is important to provide detailed instructions to the user through the message window or dialog box that the application expects the user to make a selection. Use the function **ProMessageDisplay()** to explain to the user to the type or purpose of the selection you want them to make.

Note: When using this function in a UI command, make sure that the command priority for the UI command is appropriate for using **ProSelect()**. Improper priority settings can cause unpredictable results. See also Normal priority actions in the User Interface: Menus, Commands, and Popupmenus chapter.

The synopsis of **ProSelect()** is as follows:

```
ProError ProSelect (

    char            option[],          /* (In)  The selection filter. */
    int             max_count,         /* (In)  The maximum number of
                                         selections allowed. */
    ProSelection     *p_in_sel,        /* (In)  An array of pointers to
                                         selection structures used
                                         to initialize the array
                                         of selections. This can
                                         be NULL. */
    ProSelFunctions  *sel_func,        /* (In)  A pointer to a structure
                                         of filter functions. This
                                         can be NULL. */
    ProSelectionEnv  sel_env,          /* (In)  Use NULL. */
    ProSelAppAction  appl_act_data,    /* (In)  Use NULL. */
    ProSelection     **p_sel_array,    /* (Out) A pointer to an array of
                                         pointers to selected items.
                                         This argument points to static
                                         memory allocated by the
                                         function. It is reallocated
                                         on subsequent calls to this
                                         function.*/
    int             *p_n_sels          /* (Out) The actual number of
                                         selections made. The
                                         function allocates the
                                         memory for this function
                                         and reuses it on
                                         subsequent calls.*/
)
```

The first input argument to **ProSelect()**, *option*, is the set of item types that can be selected. This is in the form of a C string which contains the names of the types separated by commas (but no spaces). The following table lists the item types that can be selected by **ProSelect()**, the name of the type that must appear in the *option* argument, and the value of *ProType* contained in the *ProModelitem* for the selected item.

Pro/ENGINEER Database Item	ProSelect() Option	ProType
Geometry Items		
Datum point	point	PRO_POINT
Datum axis	axis	PRO_AXIS
Datum plane	datum	PRO_SURFACE
Coordinate system datum	csys	PRO_CSYS
Coordinate System Axis	csys_axis	PRO_CSYS_AXIS_X PRO_CSYS_AXIS_Y PRO_CSYS_AXIS_Z
Edge (solid or datum surface)	edge	PRO_EDGE
Vertex	edge_end	PRO_EDGE_START, or PRO_EDGE_END
Datum curve	curve	PRO_CURVE
Datum curve end	curve_end	PRO_CRV_START, or PRO_CRV_END
Composite Curve	comp_crv	PRO_CURVE
Edge (solid only)	sldedge	PRO_EDGE
Edge (datum surface only)	qltedge	PRO_EDGE
Surface (solid or quilt)	surface	PRO_SURFACE
Surface (solid)	sldface	PRO_SURFACE
Surface (datum surface)	qltface	PRO_SURFACE
Quilt	dtmqlt	PRO_QUILT
Annotations		
Dimension	dimension	PRO_DIMENSION
Reference dimension	ref_dim	PRO_REF_DIMENSION
Geometric tolerance	gtol	PRO_GTOL
3D symbol	symbol_3d	PRO_SYMBOL_INSTANCE
Note	any_note	PRO_NOTE

Pro/ENGINEER Database Item	ProSelect() Option	ProType
3D Note	note_3d	PRO_NOTE
3D surface finish	surffin_3d	PRO_SURF_FIN
Annotation element	annot_elem	PRO_ANNOTATION_ELEM
Drawing Items		
Drawing view	dwg_view	PRO_VIEW
Drawing table	dwg_table	PRO_DRAW_TABLE
Draft entity	draft_ent	PRO_DRAFT_ENTITY
Detail symbol	dtl_symbol	PRO_SYMBOL_INSTANCE
Detail note	any_note	PRO_NOTE
Table cell	table_cell	PRO_DRAW_TABLE
Solids and Features		
Feature	feature	PRO_FEATURE
Part	part	PRO_PART
Component Feature	membfeat	PRO_FEATURE
Assembly component model	component	PRO_PART, PRO_ASSEMBLY
Part or subassembly	prt_or_asm	PRO_PART, or PRO_ASSEMBLY
Miscellaneous Items		
Mechanica Items	Refer to the table in the section Selection of Mechanica Items	
External object	ext_obj	PRO_EXTOBJ
Diagram fixed connector, fixed component, or parametric connector	dgm_obj	PRO_DIAGRAM_OBJECT
Diagram wire (not a cable)	dgm_non_cable_wire	PRO_DIAGRAM_OBJECT

The second argument specifies the maximum number of items the user can select. If there is no maximum, set this argument to -1.

The third argument to **ProSelect()** is an expandable array of *ProSelection* structures (created using **ProArrayAlloc()** and **ProSelectionAlloc()**) used to initialize the selection list. For more information refer to section Expandable Arrays in the chapter Fundamentals. This is used in situations like **Feature, Define** in Pro/ENGINEER where the user has the option of removing a default selection for a feature reference.

The fourth argument is an optional structure that specifies three, user-defined filter functions. These enable you to filter the items that are selectable in a customized way. For example, you could arrange that only straight edges are selectable by writing a filter that would check the type of the edge, and return an appropriate status. This function would then be called within **ProSelect()** to prevent the user from selecting a curved edge.

The fifth and sixth arguments are not used in this release and should be set to NULL.

The final two arguments are the outputs, that is, an expandable array of *ProSelection* structures, and the number of items in the array. The previous section explains how to analyze the contents of a *ProSelection* object.

Note: The array of *ProSelections* is allocated and reused on subsequent calls to **ProSelect()**.

Therefore, you must not free the output array. Also, if you wish to preserve any of the selections made, you should copy that selection using **ProSelectionCopy()**.

Highlighting

Functions Introduced:

- **ProSelectionHighlight()**
- **ProSelectionDisplay()**
- **ProSelectionUnhighlight()**

The function **ProSelectionHighlight()** highlights an item specified by a *ProSelection* object in a color chosen from the enumerated type *ProColortype*. This highlight is the same as the one used by Pro/ENGINEER (and **ProSelect()**) when selecting an item—it just repaints the wire-frame display in the new color. The highlight is removed if you use the View Repaint command or **ProWindowRepaint()**; it is not removed if you use **ProWindowRefresh()**.

The function **ProSelectionUnhighlight()** removes the highlight.

ProSelectionHighlight() will not change the highlight color of an item already highlighted. If you need to do this, call **ProSelectionUnhighlight()** on the first item.

The function **ProSelectionDisplay()** does the same highlight as **ProSelectionHighlight()** but uses the standard highlight color used by Pro/ENGINEER.

Note: For performance reasons, calls to **ProSelectionHighlight()** are cached and executed after a short delay or after a window repaint.

Selection Buffer

Introduction to Selection Buffers

Selection is the process of choosing items on which you want to perform an operation. In Pro/ENGINEER, before a feature tool is invoked, the user can select items to be used in a given tool's collectors. Collectors are like storage bins of the references of selected items. The location where preselected items are stored is called the selection buffer.

Depending on the situation, different selection buffers may be active at any one time. In Part and Assembly mode, Pro/ENGINEER offers the default selection buffer, the Edit selection buffer, and other more specialized buffers. Other Pro/ENGINEER modes offer different selection buffers.

In the default Part and Assembly buffer there are two levels at which selection is done:

- First Level Selection

Provides access to higher-level objects such as features or components. You can make a second level selection only after you select the higher-level object.

- Second Level Selection

Provides access to geometric objects such as edges and faces.

Note: First-level and second-level objects are usually incompatible in the selection buffer.

Pro/TOOLKIT allows access to the contents of the currently active selection buffer. The available functions allow your application to:

- Get the contents of the active selection buffer.
- Remove the contents of the active selection buffer.
- Add to the contents of the active selection buffer.

Reading the Contents of the Selection Buffer

Functions Introduced:

- **ProSelbufferSelectionsGet()**
- **ProSelectionCollectionGet()**

Use the function **ProSelbufferSelectionsGet()** to access the contents of the current selection buffer. The function returns independent copies of the selections in the selection buffer (if the buffer is cleared, this array is still valid).

If there is no active selection buffer this function returns an error.

Use the standard Pro/TOOLKIT functions to parse the contents of the *ProSelection* array.

However, the selection buffer stores chain and surface collections using a special mechanism. The function

ProSelectionCollectionGet() can be used to extract the *ProCollection* object from a *ProSelection*.

Note: As per the manner of storage of the collection in the selection buffer, once the collection has been cleared from the buffer the *ProSelection* referring to the collection is no longer valid. Therefore it is recommended to extract the *ProCollection* object from the *ProSelection* before there is a possibility that it may be cleared from the selection buffer.

Removing the Items from the Selection Buffer

Functions Introduced:

- **ProSelbufferClear()**
- **ProSelbufferSelectionRemove()**

Use the function **ProSelbufferClear()** to clear the currently active selection buffer of all contents. After the buffer is cleared, all contents are lost.

Use the function **ProSelbufferSelectionRemove()** to remove a specific selection from the selection buffer. The input argument is the index of the item (the index where the item was found in the call to **ProSelbufferSelectionsGet()**).

Note: Because of the specialized nature of the Edit buffer in Pro/ENGINEER, modification of the contents of the Edit buffer is not supported.

Adding Items to the Selection Buffer

Functions Introduced:

- **ProSelbufferSelectionAdd()**
- **ProSelbufferCollectionAdd()**

Use the function **ProSelbufferSelectionAdd()** to add an item to the active selection buffer.

Note: The selected item must refer to an item that is in the current model such as its owner, component path or drawing view.

This function may fail due to any of the following reasons:

- There is no current selection buffer active.
- The selection does not refer to the current model.
- The item is not currently displayed and so cannot be added to the buffer.
- The selection cannot be added to the buffer in combination with one or more objects that are already in the buffer. For example: geometry and features cannot be selected in the default buffer at the same time.

Use the function **ProSelbufferCollectionAdd()** to add a chain or surface collection to the active selection buffer. It has similar restrictions as **ProSelbufferSelectionAdd()**.

Note: because of the specialized nature of the Edit buffer in Pro/ENGINEER, modification of the contents of the Edit buffer is not supported.

Refer to the sample code 'Example 3: Assigning Pro/ENGINEER command to popup menus'. This example demonstrates the use of the selection buffer to determine the context for a popup menu button.

User Interface: Curve and Surface Collection

This chapter describes the Pro/TOOLKIT functions to access the details of curve and surface collections for query and modification. Curve and surface collections are required inputs to a variety of Pro/ENGINEER tools such as Round, Chamfer, and Draft.

Topic	Page
Introduction to Curve and Surface Collection	20 - 2
Interactive Collection	20 - 3
Accessing Collection object from Selection Buffer	20 - 6
Adding a Collection Object to the Selection Buffer	20 - 7
Programmatic Access to Collections	20 - 7
Access of Collection Object from Feature Element Trees	20 - 15
Programmatic Access to Legacy Collections	20 - 16

Introduction to Curve and Surface Collection

A curve collection or chain is a group of separate edges or curves that are related, for example, by a common vertex, or tangency. Once selected, these separate entities are identified as a chain so they can be modified as one unit.

The different chain types are as follows:

- One-by-one—a chain of edges, curves or composite curves, each adjacent pair of which has a coincident endpoint. Some applications may place other conditions on the resulting chain.
- Tangent—a chain defined by the selected item and the extent to which adjacent entities are tangent.
- Curve—an entire composite curve or some portion of it that is defined by two component curves of the curve.
- Boundary—an entire loop of one-sided edges that bound a quilt or some portion thereof defined by two edges of the boundary loop.
- Surf Chain—an entire loop of edges that bound a face (solid or surface) or some portion of it that is defined by two edges of the loop.
- Intent Chain—an intent chain entity, usually created as the result of two intersecting features.
- From/To—a chain that begins at a start-point, follows an edge line, and ends at the end-point.

Surface sets are one or more sets of surfaces either for use within a tool, or before entering a tool.

The definition of a surface set may not be independent in all respects from that of any other. In other words, the ability to construct some types of surface sets may depend on the presence of or on the content of others. On this account, we have different surface sets as follows:

- One-by-One Surface Set—Represents a single or a set of single selected surfaces, which belong to solid or surface geometry.
- Intent Surface Set—Represents a single or set of intent surfaces, which are used for the construction of the geometry of features. This instruction facilitates the reuse of the feature construction surface geometry as "intent" reference. This is also known as "logical object surface set".

- All Solid or Quilt Surface Set—Represents all the solid or quilt surfaces in the model.
- Loop Surface Set—Represents all the surfaces in the loop in relation with the selected surface and the edge. This is also known as "neighboring surface set".
- Seed and Boundary Surface Set—Represents all the surfaces between the seed surface and the boundary surface, excluding the boundary surface itself.

Surface set collection can be identified as a gathering a parametric set of surfaces in the context of a tool that specifically requests surface sets and is nearly identical to selection of a surface set.

Chain collection can be identified as a gathering a chain in the context of a tool that specifically requests chain objects and is nearly identical to chain selection.

Collection is related to Selection as follows:

Selection is the default method of interaction with Pro/ENGINEER system. Selection is performed without the context of any tool. In other words, the system does not know what to do with selected items until the user tells the system what to do. Collection is essentially Selection within the context of a tool. Items are gathered for a specific use or purpose as defined by the tool, which forms the Collection. It is possible to convert the collections into the sets of selections using the collection APIs.

The ProCollection object

A *ProCollection* object is an opaque pointer to a structure in which a surface or curve collection is stored. It represents a chain or surface set and extracts the details from an appropriate structure from Pro/ENGINEER.

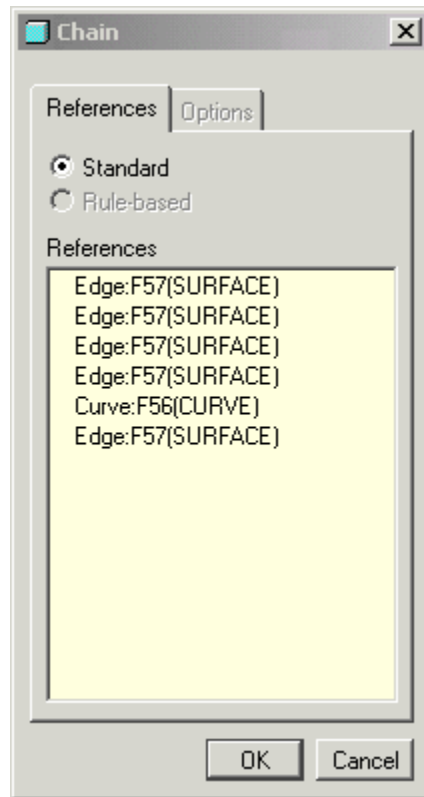
Interactive Collection

Functions Introduced:

- **ProCurvesCollect()**
- **ProSurfacesCollect()**

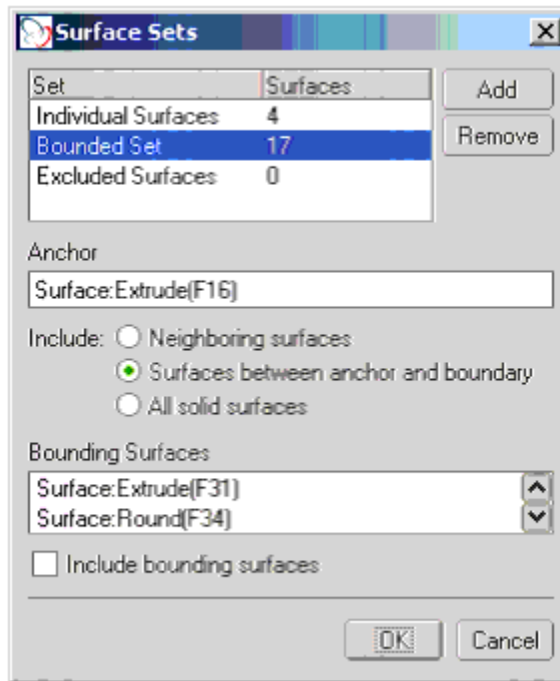
Use the function **ProCurvesCollect()** to interactively create a collection of curves by invoking a chain collection user interface.

Figure 20-1: Chain collection User Interface



Use the function **ProSurfacesCollect()** to interactively create a collection of surfaces by invoking a surface sets dialog.

Figure 20-2: Surface Sets dialog box



For the functions **ProCurvesCollect()** and **ProSurfacesCollect()** the input arguments are as follows:

- *types*—Specifies an array defining the permitted instruction types.
- *n_types*—Specifies the size of the types array.
- *filter_func*—Specifies the filter function. The filter function is called before each selection is accepted. If your application wishes to reject a certain selection, it can return an error status from this filter. You may pass NULL to skip the filter.
- *app_data*—Specifies the application data that will be passed to *filter_func*. You may pass NULL when this is not required.

- *collection*—Specifies the collection object where the results will be stored. This should be preallocated using **ProCollectionAlloc()** (for **ProCurvesCollect()**) and **ProSrfaceCollectionAlloc()** (for **ProSurfacesCollect()**) respectively. The collection properties and instructions will be stored in this handle after the call to the functions **ProCurvesCollect()** and **ProSurfacesCollect()**. Use the functions for programmatic access to curve and surface collections and to extract the required information.

The output arguments are as follows:

- *sel_list*—Specifies a pointer to an array of *ProSelection* objects describing the current resulting curves and edges or surfaces resulting from the collection. You may pass NULL if you are not currently interested in the results of the collection.
- *n_sel*—Specifies the number of entries in *sel_list*.

Accessing Collection object from Selection Buffer

Function Introduced:

- **ProSelectionCollectionGet()**

The selection buffer stores chain and surface collections using a special mechanism. The function **ProSelectionCollectionGet()** can be used to extract the *ProCollection* object from a *ProSelection*. The function **ProSelectionCollectionGet()** may fail due to the following conditions:

- **PRO_TK_INVALID_TYPE**—The selection object does not contain a collection.
- **PRO_TK_INVALID_PTR**—The selection object contains a chain, but this chain reference is no longer valid. Chain references contained in *ProSelections* are only valid for as long as the chain is selected in the selection buffer.

Note: It is recommended to extract the *ProCollection* object from the *ProSelection* before it can be cleared from the selection buffer.

Adding a Collection Object to the Selection Buffer

Function Introduced:

- **ProSelbufferCollectionAdd()**

Use the function **ProSelbufferCollectionAdd()** to add a chain or surface collection to the active selection buffer. The function **ProSelbufferCollectionAdd()** may fail due to the following conditions:

- **PRO_TK_INVALID_ITEM** - The collection does not correctly reference the current model.
- **PRO_TK_NOT_DISPLAYED** - The collection contains one or more items that are not currently displayed (for example, due to inactive geometry) and so cannot be added to the buffer.
- **PRO_TK_BAD_CONTEXT** - The selection cannot be added to the buffer in combination with one or more objects that are already in the buffer (for example geometry and features selected at the same time).

For further information on related functions, refer to the section Selection Buffer of chapter User Interface: Selection.

User Interface: Curve and Surface Collection

Programmatic Access to Collections

A *ProCollection* object may be returned from some indeterminate contexts or functions, however it can only be one of the following types:

- **PRO_COLLECTION_LEGACY**—legacy curve collection type (generated by **ProCurvesCollect()** or a superseded function like **ProCollectionAlloc()**.
- **PRO_COLLECTION_SRFCOLL**—surface collection type.
- **PRO_COLLECTION_CRVCOLL**—curve collection type.

Function Introduced:

- **ProCollectionTypeGet()**

Use the function **ProCollectionTypeGet()** to get the *ProCollectionType*.

Contents of Curve Collection

Functions Introduced:

- **ProCrvcollectionInstructionsGet()**
- **ProCrvcollinstrTypeGet()**
- **ProCrvcollinstrAttributelsSet()**
- **ProCrvcollinstrReferencesGet()**
- **ProCrvcollinstrValueGet()**
- **ProCrvcollinstrFree()**
- **ProCrvcollinstrArrayFree()**

Use the function **ProCrvcollectionInstructionsGet()** to get the instructions from the curve collection.

Use the function **ProCrvcollinstrTypeGet()** to get the curve collection instruction type.

Curve collection instructions can be of the following types:

- **PRO_CURVCOLL_ADD_ONE_INSTR**—for creating "One by One" chain.
- **PRO_CURVCOLL_TAN_INSTR**—for creating "Tangent" chain.
- **PRO_CURVCOLL_CURVE_INSTR**—for creating "Curve" chain.
- **PRO_CURVCOLL_SURF_INSTR**—for creating "Surface Loop" chain.
- **PRO_CURVCOLL_BNDRY_INSTR**—for creating "Boundary Loop" chain.
- **PRO_CURVCOLL_LOG_OBJ_INSTR**—for creating "Logical Object" chain.
- **PRO_CURVCOLL_PART_INSTR**—for creating chain on all possible references, or to choose from convex or concave only.
- **PRO_CURVCOLL_FEATURE_INSTR**—for creating chain from feature curves.
- **PRO_CURVCOLL_FROM_TO_INSTR**—for creating "From-To" chain.
- **PRO_CURVCOLL_EXCLUDE_ONE_INSTR**—for excluding the entity from the chain.
- **PRO_CURVCOLL_TRIM_INSTR**—to trim chain.

- PRO_CURV_COLL_EXTEND_INSTR—to extend chain.
- PRO_CURV_COLL_START_PNT_INSTR—to set the chain start point.
- PRO_CURV_COLL_ADD_TANGENT_INSTR—to add all edges tangent to the ends of the chain.
- PRO_CURV_COLL_ADD_POINT_INSTR—to add selected point or points to the collection.

Use the function **ProCrvcollinstrAttributeIsSet()** to check whether a special attribute is set for the curve collection instruction.

The curve collection instruction attributes can be of the following types:

- PRO_CURV_COLL_NO_ATTR—applicable when there are no attributes present.
- PRO_CURV_COLL_ALL—applicable for all edges.
- PRO_CURV_COLL_CONVEX—applicable for convex edges only.
- PRO_CURV_COLL_CONCAVE—applicable for concave edges only.

Use the function **ProCrvcollinstrReferencesGet()** to get the references contained in a curve collection instruction.

Use the function **ProCrvcollinstrValueGet()** to get the value of a curve collection instruction. This is valid for instructions of PRO_CURV_COLL_TRIM_INSTR and PRO_CURV_COLL_EXTEND_INSTR type.

Use the function **ProCrvcollinstrFree()** to release a curve collection instruction.

Use the function **ProCrvcollinstrArrayFree()** to release the *ProArray* of curve collection instructions.

Creation and Modification of Curve Collections

Functions Introduced:

- **ProCrvcollectionAlloc()**
- **ProCrvcollectionCopy()**
- **ProCrvcollectionInstructionAdd()**
- **ProCrvcollectionInstructionRemove()**
- **ProCrvcollinstrAlloc()**
- **ProCrvcollinstrAttributeSet()**
- **ProCrvcollinstrAttributeUnset()**
- **ProCrvcollinstrReferenceAdd()**
- **ProCrvcollinstrReferenceRemove()**
- **ProCrvcollinstrValueSet()**
- **ProCrvcollectionRegenerate()**

Use the function **ProCrvcollectionAlloc()** to allocate a curve collection.

Use the function **ProCrvcollectionCopy()** to copy a curve collection which is a newly allocated collection object internally and can be freed from memory using the function **ProCollectionFree()**.

Note: **ProCrvcollectionCopy()** function should be used to convert the collection object returned by **ProCurvesCollect()** to a **PRO_COLLECTION_CRV_COLL** type of collection so that it can be used by the curve collection access functions.

Use the function **ProCrvcollectionInstructionAdd()** to add an instruction to a curve collection.

Use the function **ProCrvcollectionInstructionRemove()** to remove an instruction from a curve collection.

Use the function **ProCrvcollinstrAlloc()** to allocate a curve collection instruction.

Use the function **CrvcollinstrAttributeSet()** to add an attribute to the curve collection instruction.

Use the function **ProCrvcollinstrAttributeUnset()** to remove an attribute of a curve collection instruction.

Use the function **ProCrvcollinstrReferenceAdd()** to add a reference to curve collection instruction references.

Use the function **ProCrvcollinstrReferenceRemove()** to remove a reference to curve collection instruction.

Use the function **ProCrvcollinstrValueSet()** to set value of the curve collection instruction.

Use the function **ProCrvcollectionRegenerate()** to generate an array of objects based on the rules and information in the collection.

Contents of Surface Collection

Functions Introduced:

- **ProSrfcollectionInstructionsGet()**
- **ProSrfcollinstrTypeGet()**
- **ProSrfcollinstrIncludeGet()**
- **ProSrfcollinstrReferencesGet()**
- **ProSrfcollinstrArrayFree()**
- **ProSrfcollinstrFree()**
- **ProSrfcollrefTypeGet()**
- **ProSrfcollrefItemGet()**
- **ProSrfcollrefArrayFree()**
- **ProSrfcollrefFree()**

Use the function **ProSrfcollectionInstructionsGet()** to get an array of instructions assigned to the surface collection.

Use the function **ProSrfcollinstrTypeGet()** to get the type of surface collection instruction.

Surface collection instructions can be of the following types:

- **PRO_SURFCOLL_SINGLE_SURF**—Instruction specifying a set of single surfaces.
- **PRO_SURFCOLL_SEED_N_BND**—Instruction specifying a combination of Seed and Boundary type of surfaces.
- **PRO_SURFCOLL_QUILT_SRFS**—Instruction specifying quilt type of surfaces.
- **PRO_SURFCOLL_ALL_SOLID_SRFS**—Instruction specifying all solid surfaces in the model.

- PRO_SURFCOLL_NEIGHBOR—Instruction specifying neighbor type of surfaces (boundary loop).
- PRO_SURFCOLL_LOGOBJ_SRFS—Instruction specifying intent surfaces. Intent surfaces are also known as "logical objects".

The following flags are used as an input to *ProSurfacesCollect* and drive its behavior of the interactive collection.

- PRO_SURFCOLL_DISALLOW_QLT—Do not allow selections from quilts.
- PRO_SURFCOLL_DISALLOW_SLD—Do not allow selections from solid geometry.
- PRO_SURFCOLL_DONT_MIX—Allow selections from only solid or only quilt but no mixing.
- PRO_SURFCOLL_SAME_SRF_LST—Allow selections from same solid or same quilt.
- PRO_SURFCOLL_USE_BACKUP—Prompts Pro/ENGINEER to regenerate using backups.
- PRO_SURFCOLL_DONT_BACKUP—Do not back up copy of references.
- PRO_SURFCOLL_DISALLOW_LOBJ—Do not allow selections from intent surfaces or logical objects.
- PRO_SURFCOLL_ALLOW_DTM_PLN—Allows datum plane selection. It is not supported in Pro/ENGINEER Wildfire 2.0.

Use the function **ProSrffcollinstrIncludeGet()** to check whether the include flag of the surface collection instruction is set. If the include flag is PRO_B_TRUE, the surfaces generated by this instruction add surfaces to the overall set. If the include flag is PRO_B_FALSE, the surfaces generated by this instruction are removed from in the overall set.

Use the function **ProSrffcollinstrReferencesGet()** to get the references contained in a surface collection instruction.

Use the function **ProSrffcollinstrArrayFree()** to free a *ProArray* of surface collection reference handles.

Use the function **ProSrffcollinstrFree()** to release the surface collection instructions.

Use the function **ProSrffcollrefTypeGet()** to get the type of reference contained in the surface collection reference.

Surface collection references can be of the following types:

- **PRO_SURFCOLL_REF_SINGLE**—Specifying the collection reference belonging to the "single surface set" type of instruction. This type of reference can belong to single surface type of instruction.
- **PRO_SURFCOLL_REF_SEED**—Specifying the collection reference to be the seed surface. This type of reference can belong to seed and boundary type of instruction.
- **PRO_SURFCOLL_REF_BND**—Specifying the collection reference to be a boundary surface. This type of reference can belong to seed and boundary type of instruction.

A single seed and boundary type of instruction will have at least one of each seed and boundary type of reference.

- **PRO_SURFCOLL_REF_NEIGHBOR**—Specifying the collection reference to be of neighbor type. This type of reference belongs neighbor type of instruction.
- **PRO_SURFCOLL_REF_NEIGHBOR_EDGE**—Specifying the collection reference of neighbor edge type. This type of reference belongs to neighbor type of instruction.

A neighbor type of instruction will have one neighbor and one neighbor edge type of reference.

- **PRO_SURFCOLL_REF_GENERIC**—Specifying the collection reference to be of generic type. This type of reference can belong to intent surfaces , quilt and all-solid type of instructions.

Use the function **ProSrffcollrefArrayFree()** to free a ProArray of surface collection reference handles.

Use the function **ProSrffcollrefFree()** to free a surface collection reference handle.

Use the function **ProSrffcollrefItemGet()** to get the geometry item contained in the surface collection reference.

Creation and Modification of Surface Collections

Functions Introduced:

- **ProSrffcollectionAlloc()**
- **ProSrffcollectionCopy()**
- **ProSrffcollectionInstructionAdd()**
- **ProSrffcollectionInstructionRemove()**
- **ProSrffcollinstrAlloc()**
- **ProSrffcollinstrIncludeSet()**
- **ProSrffcollinstrReferenceAdd()**
- **ProSrffcollinstrReferenceRemove()**
- **ProSrffcollrefAlloc()**
- **ProSrffcollectionRegenerate()**

Use the function **ProSrffcollectionAlloc()** to allocate a surface collection.

Use the function **ProSrffcollectionCopy()** to copy a surface collection into a newly allocated *ProCollection* handle.

Use the function **ProSrffcollectionInstructionAdd()** to add an instruction to surface collection.

Use the function **ProSrffcollectionInstructionRemove()** to remove the instructions from surface collections.

The function **ProSrffcollectionInstructionRemove()** may fail due to invalid index specifications for the instruction or if the collection contains instruction for curves.

Use the function **ProSrffcollinstrAlloc()** to allocate a surface collection instruction.

Use the function **ProSrffcollinstrIncludeSet()** to set the include flag for a surface collection instruction.

Use the function **ProSrffcollinstrReferenceAdd()** to add a reference to surface collection instruction reference.

Use the function **ProSrffcollinstrReferenceRemove()** to remove a reference to surface collection instruction.

Use the function **ProSrffcollectionRegenerate()** to generate an array of objects based on the rules and information in the collection.

Access of Collection Object from Feature Element Trees

Functions Introduced:

- **ProElementCollectionGet()**
- **ProElementCollectionSet()**
- **ProElementCollectionProcess()**

Use the function **ProElementCollectionGet()** to extract a collection object from an element of a feature element tree of the following types:

- PRO_E_STD_CRV_COLLECTION_APPL
- PRO_E_STD_SURF_COLLECTION_APPL

Use the function **ProElementCollectionSet()** to assign a collection object into the appropriate elements of the above mentioned types.

Note: Point collections (PRO_E_POINT_COLLECTION_APPL) are accessed from the element as normal *ProReference* object—by using **ProElementReferenceGet()** and **ProElementReferenceSet()** APIs.

Use the function **ProElementCollectionProcess()** to generate a list of geometric items in the form of array of *ProReference* object constituting the collection. It returns a *ProArray* of the selected curves, edges, or surfaces that exist in the collection owned by the element. This function differs from **ProCrvcollectionRegenerate()** and **ProSrfcollectionRegenerate()** because those functions generate a list of collected entities based on standard rules. The function **ProElementCollectionProcess()**, if applied to a collection element from a given feature, will use the feature's rules to process the collection and return the exact geometric entities used by the feature.

Note: If this element is extracted from an existing feature tree using **ProFeatureElementtreeExtract()**, the returned *reference_array* will be on the basis of the feature rules. On the other hand, if this element is newly created and not yet assigned to a feature, then the returned *reference_array* will be as per the default rules.

Programmatic Access to Legacy Collections

Functions Introduced:

- **ProCollectionAlloc()**
- **ProCollectionrefAlloc()**
- **ProCollectionrefSelectionSet()**
- **ProCollectionrefTypeSet()**
- **ProCollectioninstrAlloc()**
- **ProCollectioninstrTypeSet()**
- **ProCollectioninstrRefAdd()**
- **ProCollectionInstrAdd()**
- **ProCollectionInstrVisit()**
- **ProCollectioninstrTypeGet()**
- **ProCollectioninstrRefVisit()**
- **ProCollectionrefSelectionGet()**
- **ProCollectionrefTypeGet()**
- **ProCollectionFree()**
- **ProCollectioninstrFree()**
- **ProCollectionrefFree()**
- **ProCollectionInstrRemove()**
- **ProCollectioninstrRefRemove()**
- **ProCrvcollectionInstrRegen()**

The functions listed in this section have been superseded. PTC recommends using the *ProCrvcollection** and *ProSrfcollection** functions instead of these; documentation for these functions is provided for maintenance of existing applications.

Example 1: Interactive Curve Collection using Pro/TOOLKIT

This example demonstrates the interactive curves collection and computes the total curve length - consisting of addition of lengths of individual curves in resulting selection set. The collection filter function is set to allow only edge type of selections.

```
/*----- Pro/Toolkit Includes -----*/
#include <ProToolkit.h>
#include <ProCollect.h>
#include <ProParameter.h>
#include <ProParamval.h>
#include <ProGeomitem.h>
/*----- Application Includes-----*/
#include <TestError.h>
#include <ProCrvcollection.h>
/*----- Function Prototypes-----*/
ProError UserCurveLength();
ProError UserCrvcollectionFilter (ProSelection  cur_sel,
ProAppData  app_data);
/*----- Static variables -----*/
static int crv_entry_count;
/*=====*\
Function: UserCurveLength()
Purpose: Compute Area of the Surface Collection
\*=====*/
ProError UserCurveLength()
{
    int sel_count;
    ProError status;
    ProCharName name;
    ProModelitem model_item;
    ProFileName msgfile;
    ProName w_name;
    ProCollection collection;
    ProSelection * regen_sels;
    int n_regen_sels;
    double total_length = 0;
    ProSelection * coll_sels;
    int coll_sels_size;
    ProCollectioninstrType coll_flags[6] = {
        PRO_CURVCOLL_ONE_BY_ONE,
        PRO_CURVCOLL_TAN_CHAIN,
        PRO_CURVCOLL_CURVE_CHAIN,
        PRO_CURVCOLL_BNDRY_CHAIN,
        PRO_CURVCOLL_SURF_CHAIN,
        PRO_CURVCOLL_LOG_EDGE };
    crv_entry_count++;
/*-----*\
Prompt user for Curve Collection
\*-----*/
```

User Interface: Curve and
Surface Collection

```

    ProStringToWstring(msgfile,"msg_uggeom.txt");
    status = ProMessageDisplay(msgfile,"USER Collect Curves to find total
length:");
    ERROR_CHECK("UserCurveLength", "ProMessageDisplay",status);
    status = ProCollectionAlloc ( &collection );
    ERROR_CHECK( "UserCurveLength", "ProCollectionAlloc", status );
    /*-----*\
Interactive Curve Collection
\*-----*/
    status = ProCurvesCollect ( coll_flags,
                                6,
                                (ProCrvcollFilter)UserCrvcollectionFilter,
                                (ProAppData ) NULL,
                                &collection,
                                &coll_sels,
                                &coll_sels_size );
    ERROR_CHECK( "UserCurveLength", "ProCurvesCollect", status );
    /*-----*\
Regenerating the returned collection to get the list of constituent curves
\*-----*/
    status = ProCrvcollectionRegenerate ( collection, &regen_sels,
&n_regen_sels );
    ERROR_CHECK( "UserCurveLength", "ProCrvcollectionRegenerate", status );
    if ( (status != PRO_TK_NO_ERROR) || (n_regen_sels <= 0) )
        return PRO_TK_GENERAL_ERROR;
    /*-----*\
        Evaluating the overall length from edge collection
    \*-----*/
    for (sel_count = 0; sel_count < n_regen_sels; sel_count++ )
    {
        ProEdge edge;
        double length;
        status = ProSelectionModelitemGet (regen_sels[sel_count],
&model_item );
        ERROR_CHECK( "UserCurveLength", "ProSelectionModelitemGet",status);
        status = ProGeomitemToEdge ( &model_item, &edge );
        ERROR_CHECK( "UserCurveLength", "ProGeomitemToEdge", status );
    /*-----*\
        Evaluating the length of individual curve
    \*-----*/
        status = ProEdgeLengthEval ( edge, &length );
        ERROR_CHECK( "UserCurveLength", "ProSurfaceAreaEval", status );
        total_length = total_length + length;
    }
    status = ProMessageDisplay(msgfile,"USER Total Curve Length is %0f",
&total_length);
    ERROR_CHECK( "UserCurveLength", "ProMessageDisplay", status );
    /*-----*\
        Freeing Memory
    \*-----*/
    status = ProCollectionFree ( &collection );

```



```

ERROR_CHECK( "UserCurveLength", "ProCollectionFree", status );
status = ProSelectionarrayFree ( regen_sels );
ERROR_CHECK( "UserCurveLength", "ProSelectionarrayFree", status );
status = ProSelectionarrayFree ( coll_sels );
ERROR_CHECK( "UserCurveLength", "ProSelectionarrayFree", status );
ProMessageClear();
return(status);
}
/*=====*\
Function: UserCrvcollectionFilter()
Purpose: Filter function for curve collection.
        Filters only PRO_EDGE type of selections
\*=====*/
ProError UserCrvcollectionFilter (ProSelection  cur_sel,
ProAppData  app_data)
{
    ProError status;
    ProModelitem model_item;
    status = ProSelectionModelitemGet (cur_sel, &model_item);
    ERROR_CHECK( "UserCrvcollectionFilter", "ProSelectionModelitemGet",
status );

    if ( model_item.type == PRO_EDGE )
        return PRO_TK_NO_ERROR;
    else
        return PRO_TK_GENERAL_ERROR;
}

```

Example 2: Interactive Surface Collection using Pro/TOOLKIT

This example demonstrates the interactive surface collection and computes the total area - consisting of addition of areas of individual surfaces in resulting selection set. It also creates parameter of double type on the individual surfaces and assigns it with the value of the individual surface area.

```

/*----- Pro/Toolkit Includes -----*/
#include <ProToolkit.h>
#include <ProCollect.h>
#include <ProParameter.h>
#include <ProParamval.h>
#include <ProSrfcollection.h>
/*----- Application Includes -----*/
#include <TestError.h>
/*----- Function Prototypes -----*/
ProError UserSurfArea();
/*----- Static variables -----*/
static int srf_entry_count;

/*=====*\
Function: UserSurfArea()

```

```

Purpose: Compute Area of the Surface Collection
\*=====*/
ProError UserSurfArea()
{
    int sel_count;
    ProError status;
    ProCharName name;
    ProModelitem model_item;
    ProFileName msgfile;
    ProName w_name;
    ProCollection collection;
    ProSelection * regen_sels;
    int n_regen_sels;
    double total_area = 0;
    ProParameter param;
    ProParamvalue param_value;
    ProName param_name;
    char param_name_s[PRO_NAME_SIZE];
    ProSelection * coll_sels;
    int coll_sels_size;
    ProCollectioninstrType coll_filters[] = { PRO_SURFCOLL_SINGLE_SURF };
    srf_entry_count++;
/*-----*\
Prompt user for Surface Collection
\*-----*/
    ProStringToWstring(msgfile, "msg_uggeom.txt");
    status = ProMessageDisplay(msgfile, "USER Collect Surfaces to find total
                                   area:");
    ERROR_CHECK("UserSurfArea", "ProMessageDisplay", status);
    status = ProSrfcollectionAlloc ( &collection );
    ERROR_CHECK( "UserSurfArea", "ProSrfcollectionAlloc", status );
/*-----*\
Regenerating the returned collection to get the list of
constituent surfaces
\*-----*/
    status = ProSrfcollectionRegenerate ( collection, &regen_sels,
                                         &n_regen_sels );
    if ( ( status != PRO_TK_NO_ERROR ) || ( n_regen_sels <= 0 ) )
        return PRO_TK_GENERAL_ERROR;
/*-----*\
Evaluating the overall area from surface collection & assigning
parameters to the individual surfaces
\*-----*/
    for ( sel_count = 0; sel_count < n_regen_sels; sel_count++ )
    {
        ProSurface surface;
        double area;
        status = ProSelectionModelitemGet ( regen_sels[sel_count],
                                           &model_item );
        ERROR_CHECK( "UserSurfArea", "ProSelectionModelitemGet", status );
        status = ProGeomitemToSurface ( &model_item, &surface );
    }
}

```

```

        ERROR_CHECK( "UserSurfArea", "ProGeomitemToSurface", status );

/*-----*\
Evaluating the area of individual surface
\*-----*/
    status = ProSurfaceAreaEval ( surface, &area );
    ERROR_CHECK( "UserSurfArea", "ProSurfaceAreaEval", status );
    total_area = total_area + area;

/*-----*\
Initializing Parameter value
\*-----*/
    param_value.type = PRO_PARAM_DOUBLE;
    param_value.value.d_val = area;
    ProTKSprintf ( param_name_s, "%s_%d", "srf_dbl",
                  srf_entry_count+sel_count );
    ProStringToWstring ( param_name, param_name_s );
    status = ProModelitemNameSet ( &model_item, param_name );
    ERROR_CHECK( "UserSurfArea", "ProModelitemNameSet", status );

/*-----*\
Creating Parameter for surface
\*-----*/
    status = ProParameterCreate ( &model_item, param_name, &param_value,
                                &param );
    ERROR_CHECK( "UserSurfArea", "ProParameterCreate", status );
}
status = ProMessageDisplay(msgfile,"USER Total Surface Area is %0f",
                          &total_area);
ERROR_CHECK( "UserSurfArea", "ProMessageDisplay", status );

/*-----*\
Freeing Memory
\*-----*/
status = ProCollectionFree ( &collection );
ERROR_CHECK( "UserSurfArea", "ProCollectionFree", status );

status = ProSelectionarrayFree ( regen_sels );
ERROR_CHECK( "UserSurfArea", "ProSelectionarrayFree", status );

status = ProSelectionarrayFree ( coll_sels );
ERROR_CHECK( "UserSurfArea", "ProSelectionarrayFree", status );

ProMessageClear();
return(status);
}

```


21

User Interface: Animation

This chapter describes the Pro/TOOLKIT functions that enable you to create animation frames and movies.

Topic	Page
Introduction	21 - 2
Animation Objects	21 - 3
Animation Frames	21 - 3
Playing Animations	21 - 4

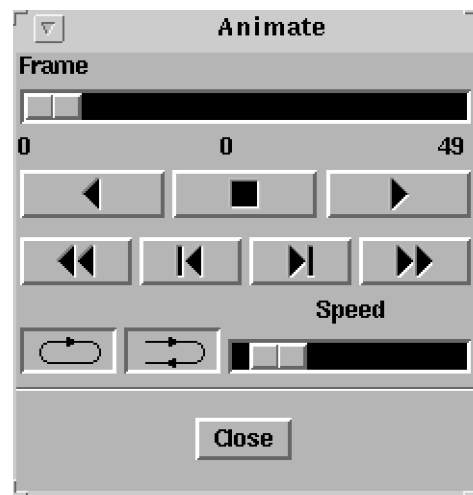
Introduction

Pro/TOOLKIT provides the functions that enable you to animate parts and assemblies. Pro/TOOLKIT handles lower-level considerations, such as hardware-dependent graphics and the user interface.

Two animation techniques are available:

- Batch animation—You create an animation movie (*ProAnimMovie* object) and users control the movie using an interface similar to a VCR. Users can perform such operations as “Play”, “Fast-Forward”, “Rewind”, and so on. The following figure shows the animation dialog box.

Figure 21-1: Animation Dialog Box



- Frame-by-frame (single) animation—You create a single animation (*ProSingleAnim* object) and code the control loop into your application. The batch animation interface does not appear. To replay the movie, you must reexecute the control loop of your application.

For both batch and single animation, you must build the animation from two important elements. These elements are:

- Animation object (*ProAnimObj* object)—Contains the object to be animated and its transformation, with respect to its immediate parent. In constructing the animation, you can show motion by creating a series of *ProAnimObj* objects, each with a different transformation.

- Animation frame (*ProAnimFrame* object)—Captures the image of the animation objects at one instance of the animation. Your final animation is a series of animation frames shown in succession.

Animation Objects

Functions Introduced:

- **ProAnimobjectCreate()**
- **ProAnimobjectDelete()**

An animation object can be a part or an assembly. To create an animation object, call the function **ProAnimobjectCreate()** and pass as input the component path of the object to be animated. You must also supply the location of the animation object with respect to its immediate parent—*not* with respect to the top-level assembly. This distinction is important when the depth of the assembly hierarchy is greater than 1.

To delete an animation object, use the function **ProAnimobjectDelete()**.

User Interface:
Animation

Animation Frames

Functions Introduced:

- **ProAnimframeCreate()**
- **ProAnimframeObjAdd()**
- **ProAnimframeObjRemove()**
- **ProAnimframeDelete()**
- **ProAnimframeObjVisit()**

After you have created animation objects, you must create an animation frame in which to store the objects. To create an animation frame, call the function **ProAnimframeCreate()** and supply a frame view matrix. The frame view matrix is a transformation from the top model coordinate system that allows you to alter the view of the top-level model in your animation. This functionality could be used, for example, to change the view of an assembly while the assembly components (animation objects) move as specified in the call to **ProAnimobjectCreate()**.

Note: The frame view matrix is specified as a transformation from the root assembly.

Creating an animation frame does not cause the animation objects to be contained in the frame. To add animation objects to a frame, you must call the function **ProAnimframeObjAdd()**.

To remove an object from a frame, call the function **ProAnimframeObjRemove()**. To delete a frame, call **ProAnimframeDelete()**.

The function **ProAnimframeObjVisit()** enables you to visit each animation object in an animation frame. The input arguments of the function specify the action and filtration functions, which are of type *ProAnimObjAct*.

Playing Animations

This section describes how to use your animation frames to construct and play the animation. As previously mentioned, there are two types of animation—single and batch.

Single Animation

Functions Introduced:

- **ProSingleAnimationInit()**
- **ProSingleAnimationPlay()**
- **ProSingleAnimationClear()**

If you want to use single animation, your Pro/TOOLKIT application must include a control loop that displays one animation frame after another. Before executing your control loop, initialize the single animation by calling **ProSingleAnimationInit()**. Within the loop, display each frame in turn using the function **ProSingleAnimationPlay()**.

Note: Single animation does not involve the batch animation user interface. The control over a single animation is contained entirely within your application.

The function **ProSingleAnimationClear()** clears the specified single animation.

Batch Animation

Batch animation implements the user interface shown in the Introduction section on page 21 - 2. The interface enables users to control the playing of your animation movie.

Animation Movies

Functions Introduced:

- **ProAnimmovieCreate()**
- **ProAnimmovieFrameAdd()**
- **ProAnimmovieFrameRemove()**
- **ProAnimmovieDelete()**
- **ProAnimmovieFrameVisit()**

The function **ProAnimmovieCreate()** creates an animation movie. At its creation, a movie does not contain any frames. To add frames to the animation movie, call the function **ProAnimmovieFrameAdd()**.

To remove a frame from an animation movie, call the function **ProAnimmovieFrameRemove()**. Note that this action does not cause the frame to be deleted; use **ProAnimmovieDelete()** to release the memory of the animation frame.

The function **ProAnimmovieFrameVisit()** enables you to visit each of the frames in an animation movie. The input arguments to the function specify the action and filtration functions, which are of type *ProAnimFrameAct*.

Playing a Batch Animation

Function Introduced:

- **ProBatchAnimationStart()**
- **ProAnimmovieSpinflagSet()**

Batch animation manages the display of animation frames inside Pro/ENGINEER. When you call **ProBatchAnimationStart()**, the system displays the VCR-like user interface. This interface enables users to control the speed and direction of the animation.

The function **ProBatchAnimationStart()** requires as input the animation movie to be started (animated). In addition, you can supply a callback function to be invoked before each animation frame is displayed. The callback function is of type *ProBatchAnimAct*.

Use the function **ProAnimmovieSpinflagSet()** to set the *ProBoolean* flag that allows spin in a batch animation process. If the animation includes view modifications, this flag should be set to false, otherwise it can be true.

Example 1: Creating a Batch Animation

The following example code shows how to animate an assembly component. The selected component rotates about the x-axis.

```

/*-----*\
Pro/TOOLKIT includes
\*-----*/
#include "ProToolkit.h"
#include "ProObjects.h"
#include "ProSelection.h"
#include "ProAsmcomppath.h"
#include "ProAnimate.h"
#include <ProMessage.h>
#include <ProUtil.h>
#include <ProModelitem.h>
/*-----*\
Application includes
\*-----*/
#include "TestError.h"
/*-----*\
C system includes
\*-----*/
#include "math.h"
/*-----*\
Application data
\*-----*/
#define MAX_NUM_FRAMES 20
#define PI 3.1451
/*=====*\
FUNCTION:  UserAsmcompAnimate()
PURPOSE:  Animates an assembly component
\*=====*/
int UserAsmcompAnimate()
{
    ProMdl      model;          /* current model */
    ProModelitem anim_mditem;   /* modelitem for the animation
                                component */
    ProMdl      anim_model;     /* model for the animation
                                component */

```

```

ProSelection  *p_sel_comp;    /* ProSelection for the animation
                               component */
ProAnimObj    anim_obj;      /* animation object for the animation
                               component */
ProAsmcomppath comp_path;    /* component path for the animation
                               component */
int           n_sel;          /* number of selections */
ProMatrix     pos_mat;        /* transformation matrix of the
                               assembly component */
double        angle;          /* rotation angle */
int           num_frames;     /* number of frames in the
                               animation */
int           range[2];       /* allowable range for input */
ProAnimFrame  frame;          /* animation frame */
ProAnimMovie  anim_movie;     /* movie */
int           i;              /* counter */
ProName       msg_file;       /* message file */
int           err;            /* return status */
ProMatrix     frame_view = {
                               {1.0, 0.0, 0.0, 0.0},
                               {0.0, 1.0, 0.0, 0.0},
                               {0.0, 0.0, 1.0, 0.0},
                               {0.0, 0.0, 0.0, 1.0}
}; /* view matrix for the frame */

void ProUtilRotX (double, ProMatrix); /* rotation function */

ProStringToWstring (msg_file, "msg_ugfund.txt");
/*-----*\
Get the current model.
\*-----*/
err = ProMdlCurrentGet (&model);
if (err != PRO_TK_NO_ERROR)
{
    ProMessageDisplay (msg_file, "USER %0s",
        "Error getting current model.");
    return (err);
}

/*-----*\
Get the component to be animated (only one is allowed).
\*-----*/
ProMessageDisplay (msg_file, "USER %0s",
    "Select a component to be animated.");
err = ProSelect ("prt_or_asm", 1, NULL, NULL, NULL, NULL,
    &p_sel_comp, &n_sel);
if (err != PRO_TK_NO_ERROR)
{
    ProMessageDisplay (msg_file, "USER %0s",
        "Error or abort during selection.");
    return (err);
}

```

```

    }
/*-----*\
    Get the transformation matrix of the selection.
/*-----*\
    err = ProSelectionAsmcomppathGet (p_sel_comp[0], &comp_path);
    err = ProAsmcomppathTrfGet (&comp_path, PRO_B_TRUE, pos_mat);
/*-----*\
    Get the ProModelitem and ProMdl for the component.
/*-----*\
    err = ProSelectionModelitemGet (p_sel_comp[0], &anim_mdllitem);
    err = ProModelitemMdlGet (&anim_mdllitem, &anim_model);
/*-----*\
    Create the animation.
/*-----*\
    err = ProAnimmovieCreate (anim_model, &anim_movie);

    angle = 0.0;
    ProMessageDisplay (msg_file, "USER %0s",
        "Enter the number of frames: ");
    range[0] = 2;
    range[1] = MAX_NUM_FRAMES;

    ProMessageIntegerRead (range, &num_frames);

    for (i = 0; i < num_frames; i++)
    {
        angle += 360.0/ (double) num_frames;
        ProUtilRotX (angle, pos_mat);

        err = ProAnimobjectCreate (p_sel_comp[0], pos_mat, &anim_obj);
        err = ProAnimframeCreate (frame_view, &frame);
        err = ProAnimframeObjAdd (frame, anim_obj);
        err = ProAnimmovieFrameAdd (anim_movie, frame);
    }

    err = ProBatchAnimationStart (anim_movie, NULL, NULL);

    return (PRO_TK_NO_ERROR);
}
/*-----*\
FUNCTION:  ProUtilRotX()
PURPOSE:   Add X rotation values to the transformation matrix.
RETURNS:   None
NOTE:      Only the rotation components of the matrix are
           modified.
/*-----*\
void ProUtilRotX (
    double    angle,    /* (In)  The rotation angle */
    ProMatrix mx)        /* (Out) The transformation matrix */
{
    mx[0][0] = 1.0;

```

```
mx[0][1] = mx[0][2] = mx[1][0] = mx[2][0] = 0.0;  
mx[1][1] = mx[2][2] = cos (angle * PI / 180.0);  
mx[1][2] = sin (angle * PI / 180.0);  
mx[2][1] = - mx[1][2];  
}
```

User Interface:
Animation

Annotations: Annotation Features and Annotations

This chapter describes how to access annotation features for special customizations. This chapter provides specific functions for creation, access, and modification of annotation features and elements.

Topic	Page
Overview of Annotation Features	22 - 3
Creating Annotation Features	22 - 4
Redefining Annotation Features	22 - 4
Visiting Annotation Features	22 - 5
Visiting Annotation Features	22 - 5
Visiting Annotation Elements	22 - 5
Accessing Annotation Elements	22 - 6
Modification of Annotation Elements	22 - 9
Automatic Propagation of Annotation Elements	22 - 12
Access to Annotations	22 - 14
Annotation Orientation	22 - 17
Annotation Associativity	22 - 19
Interactive Selection	22 - 21
Display Modes	22 - 21
Designating Dimensions and Symbols	22 - 21

Dimensions	22 - 22
Notes	22 - 50
Geometric Tolerances	22 - 56
Accessing Set Datum Tags	22 - 56
Accessing Set Datums for Datum Axes or Planes	22 - 58
Surface Finish Annotations	22 - 59
Symbol Annotations	22 - 61

Overview of Annotation Features

An annotation feature is a new feature available in Pro/ENGINEER Wildfire 2.0. It is composed of one or more "annotation elements". Each annotation element is composed of references, parameters and annotations (notes, symbols, geometric tolerances, surface finishes, reference dimensions, driven dimensions, and manufacturing template annotations). The annotation feature allows annotation information to have the same benefits as geometry in Pro/ENGINEER models, that is, parameters can be assigned to these annotation elements, and missing geometric references can cause features to fail in some situations.

The feature type PRO_FEAT_ANNOTATION represents an annotation feature. Functions referring to annotation features use the structure ProAnnotationfeat, which is identical to ProFeature.

Functions referring to annotation elements use the structure ProAnnotationElem which is identical to the structure ProModelitem and is defined as

```
typedef struct pro_model_item
{
    ProType type;

    int id;

    ProMdl owner;
}ProAnnotationElem
```

Like other ProModelitem derivatives, each annotation element has a unique id assigned to it in the model.

Annotation elements may belong to annotation features, or may also be found in data-sharing features (features like Copy Geometry, Publish Geometry, Merge, Cutout, and Shrinkwrap features).

Pro/TOOLKIT does not expose the feature element tree for annotation features because some elements in the tree are used for non-standard purposes. Instead, Pro/TOOLKIT provides specific functions for creating, redefining, and reading the properties of annotation features and annotation elements.

Creating Annotation Features

Functions Introduced:

- **ProAnnotationfeatCreate()**

The function **ProAnnotationfeatCreate()** creates a new annotation feature in the model. Specify the following as the input parameters for this function:

- *owner*—Specify the solid model on which the feature will be created. Specify the component path if the feature is created in an assembly context.
- *use_ui*—Specifies a boolean flag that determines how the annotation features will be created. It can have the following values:
 - FALSE—Indicates that the feature will be a new empty annotation feature with one general annotation element in it. Modify the new annotation element and add others using subsequent steps.
 - TRUE—Indicates that Pro/ENGINEER will invoke the annotation feature creation user interface.

Redefining Annotation Features

Redefining an annotation feature involves creation of new annotation elements, deletion of elements that are not required and modification of the feature properties.

Note:

- The functions in this section are shortcuts to redefining the feature containing the annotation elements. Because of this, Pro/ENGINEER must regenerate the model after making the required changes to the annotation element. The functions include a flag to optionally allow the **Fix Model** User Interface to appear upon a regeneration failure.
- The functions below do not support the modification of Datum Target Annotation Features (DTAFs).

Functions Introduced:

- **ProAnnotationfeatElementAdd()**
- **ProAnnotationfeatElementArrayAdd()**
- **ProAnnotationfeatElementDelete()**

The function **ProAnnotationfeatElementAdd()** adds a new non-graphical annotation element to the feature.

The function **ProAnnotationfeatElementArrayAdd()** adds a series of new annotation elements to the feature. Each element may be created as non-graphical or may be assigned a pre-existing annotation.

The function **ProAnnotationfeatElementDelete()** deletes an annotation element from the feature. The function deletes the annotation element, its annotations, parameters, references, and application data from the feature.

Visiting Annotation Features

Functions Introduced:

- **ProSolidFeatVisit()**
- **ProModelitemNameGet()**

Use the function **ProSolidFeatVisit()** to visit the annotation features in the specified model.

The function **ProModelitemNameGet()** returns the name of the annotation feature.

Visiting Annotation Elements

The functions described in this section enable you to visit all the annotation elements in a solid model.

Functions Introduced:

- **ProFeatureAnnotationelemsVisit()**
- **ProSolidAnnotationelemsVisit()**

The function **ProFeatureAnnotationelemsVisit()** visits the annotation elements in the specified feature.

The function **ProSolidAnnotationelemsVisit()** visits the annotation elements in a solid model.

The filter function **ProAnnotationelemFilterAction()** and the visit function **ProAnnotationelemVisitAction()** are specified as input arguments for the above functions.

The function **ProAnnotationelemFilterAction()** is a generic function for filtering an annotation element. It returns the filter status of the specified annotation element. The filter status is passed as the input argument to the function **ProAnnotationelemVisitAction()** which is a generic function for visiting annotation elements.

Accessing Annotation Elements

The following functions provide access to the properties of an annotation element.

Functions Introduced:

- **ProAnnotationelemAnnotationGet()**
- **ProAnnotationelemCopyGet()**
- **ProAnnotationelemFeatureGet()**
- **ProAnnotationelemIsDependent()**
- **ProAnnotationelemIsIncomplete()**
- **ProAnnotationelemReferencesCollect()**
- **ProAnnotationelemQuiltreferenceSurfacesCollect()**
- **ProAnnotationelemTypeGet()**
- **ProAnnotationelemReferenceDescriptionGet()**
- **ProAnnotationelemReferencesIsStrong()**
- **ProAnnotationelemReferenceAutopropagateGet()**
- **ProAnnotationelemHasMissingrefs()**

The function **ProAnnotationelemAnnotationGet()** returns the annotation contained in an annotation element.

The function **ProAnnotationelemCopyGet()** returns the copy flag of the annotation elements. This property is not supported for elements in data sharing features.

The function **ProAnnotationelemFeatureGet()** returns the feature that owns the annotation element.

The function **ProAnnotationelemIsDependent()** returns the value of the dependency flag for the annotation element. This property is supported only for the elements in data sharing features.

The function **ProAnnotationelemIsIncomplete()** returns a true value if the annotation element has missing strong references.

The function **ProAnnotationelemReferencesCollect()** returns an array of references contained in the specified annotation element. The input arguments for this function are:

- *element*—Specifies the annotation element.
- *ref_type*—Specifies the type of references and can have one of the following values:
 - PRO_ANNOTATION_REF_ALL—All references
 - PRO_ANNOTATION_REF_WEAK—Weak references
 - PRO_ANNOTATION_REF_STRONG—Strong references
- *source*—Specifies the source of the references and can have one of the following values:
 - PRO_ANNOTATION_REF_ALL—From both user and annotation.
 - PRO_ANNOTATION_REF_FROM_ANNOTATION—From the annotation (or custom data) only.
 - PRE_ANNOTATION_REF_FROM_USER—From the user only.

Annotation elements have special default behavior for propagation of datum points to features in other models. The flag that controls this behavior can automatically propagate datum points or any other *applicable items* to data sharing features after the user has selected all other strong references of the annotation elements.

Here, applicable items are items that are designated to auto-propagate, using a checkbox in the references collector, for specific annotation elements.

The function

ProAnnotationelemQuiltreferenceSurfacesCollect() returns the surfaces which make up a quilt surface collection reference for the annotation element.

Note: All the surfaces made inactive by features occurring after the annotation element in the model regeneration are also returned.

The function

ProAnnotationelemReferenceAutopropagateGet() gets the autopropagate flag of the specified annotation element reference.

The function **ProAnnotationelemTypeGet()** returns the type of the annotation contained in the annotation element. It can have one of the following values:

- **PRO_ANNOT_TYPE_NONE**—Specifies a non-graphical annotation.
- **PRO_ANNOT_TYPE_NOTE**—Specifies a note. Refer to the section Notes for details.
- **PRO_ANNOT_TYPE_GTOL**—Specifies a geometric tolerance. Refer to the chapter Annotations: Geometric Tolerances for details.
- **PRO_ANNOT_TYPE_SRFFIN**—Specifies a surface finish. Refer to the section Surface Finish Annotations for details.
- **PRO_ANNOT_TYPE_SYMBOL**—Specifies a symbol. Refer to the section Symbol Annotations for details.
- **PRO_ANNOT_TYPE_DIM**—Specifies a driven dimension. Refer to the section Accessing Reference and Driven Dimensions for details.
- **PRO_ANNOT_TYPE_REF_DIM**—Specifies a reference dimension. Refer to the section Accessing Reference and Driven Dimensions for details.
- **PRO_ANNOT_TYPE_CUSTOM**—Specifies a manufacturing template annotation.
- **PRO_ANNOT_TYPE_SET_DATUM_TAG**—Specifies a set datum tag. Refer to the section Accessing Set Datum Tags for details.
- **PRO_ANNOT_TYPE_DRIVINGDIM**—Specifies a driving dimension annotation element. Refer to the section Driving Dimension Annotation Elements for details.

The function **ProAnnotationelemReferenceDescriptionGet()** gets the description property for a given annotation element reference.

Note: The description string is same as that of the tooltip text for the reference name in the Annotation Feature UI.

The function **ProAnnotationelemReferenceIsStrong()** identifies if a reference is weak or strong in a given annotation element.

The function **ProAnnotationelemHasMissingrefs()** enables you to identify if an annotation element has missing references. The input parameters of this function allow you to investigate specific types and sources of references, or check all references simultaneously.

Modification of Annotation Elements

The functions described in this section allow you to modify the properties of an annotation element.

Note: The functions in this section are shortcuts to redefining the feature containing the annotation elements. Because of this, Pro/ENGINEER must regenerate the model after making the indicated changes to the annotation element. The functions include a flag to optionally allow the **Fix Model** User Interface to appear upon a regeneration failure.

Functions Introduced:

- **ProAnnotationelemAnnotationSet()**
- **ProAnnotationelemCopySet()**
- **ProAnnotationelemDependencySet()**
- **ProAnnotationelemReferenceAdd()**
- **ProAnnotationelemReferenceRemove()**
- **ProAnnotationelemReferenceStrengthen()**
- **ProAnnotationelemReferenceWeaken()**
- **ProAnnotationelemReferenceDescriptionSet()**
- **ProAnnotationelemReferenceAutopropagateSet()**
- **ProAnnotationelemReferencesSet()**
- **ProAnnotationelemArrayReferencesSet()**
- **ProAnnotationreferencesetAlloc()**
- **ProAnnotationreferencesetReferenceAdd()**
- **ProAnnotationreferencesetFree()**
- **ProAnnotationreferencesetProarrayFree()**

The function **ProAnnotationelemAnnotationSet()** allows you to modify the annotation contained in an annotation element. Specify the value for the input argument *annotation* as NULL to modify the annotation element to be a non-graphical annotation.

Note: The above function does not support Datum Target Annotation Elements (DTAEs).

If you modify the annotation contained in the annotation element, the original annotation is automatically removed from the element and is owned by the model.

The function **ProAnnotationelemCopySet()** provides write access to the copy flag of the annotation element. This property is not supported for annotations in data sharing features.

The function **ProAnnotationelemDependencySet()** sets the value for the dependency flag. This property is supported only for annotations in data sharing features.

The function **ProAnnotationelemReferenceAdd()** adds a strong user-defined reference to the annotation element.

The function **ProAnnotationelemReferenceRemove()** removes the user defined reference from the annotation element.

The function **ProAnnotationelemReferenceStrengthen()** converts a weak reference to a strong reference.

The function **ProAnnotationelemReferenceWeaken()** converts a strong reference to a weak reference.

The function **ProAnnotationelemReferenceDescriptionSet()** sets the description property for a given annotation element reference.

The function

ProAnnotationelemReferenceAutopropagateSet() sets the autopropagate flag of the specified annotation element reference.

The function **ProAnnotationelemReferencesSet()** replaces all the user-defined references in the annotation element with a *ProArray* of references specified as the input argument.

The function **ProAnnotationelemArrayReferencesSet()** replaces all the user-defined references for a *ProArray* of annotation elements with the *ProArray* of reference sets specified as the input argument.

Note: All the annotation elements must belong to the same feature. The number of reference sets should match the number of annotation elements to be modified.

The function **ProAnnotationreferencesetAlloc()** allocates a set of user-defined references to be assigned to an annotation element.

The function **ProAnnotationreferencesetReferenceAdd()** adds a new reference to an existing set of user-defined annotation references.

The function **ProAnnotationreferencesetFree()** releases the set of user-defined references to be assigned to an annotation element.

The function **ProAnnotationreferencesetProarrayFree()** releases an array of reference sets to be assigned to an annotation element.

Parameters Assigned to Annotation Elements

The functions described in this section allow you to access parameters assigned to the annotation element. Specify the annotation element as the parameter owner for these functions.

Functions Introduced:

- **ProParameterIsEnumerated()**
- **ProParameterRangeGet()**

The function **ProParameterIsEnumerated()** identifies an enumerated parameter and returns the available values assigned to it. The output of the function is `PRO_B_TRUE` if the parameter is enumerated and `PRO_B_FALSE` if it is not. The output argument also presents `ProArray` of values that can be assigned to the enumerated parameter.

The function **ProParameterRangeGet()** identifies whether a parameter is restricted to a range of values. It optionally provides the boundary conditions for the range.

For more information on functions that allow you to view, create, delete, and modify parameters, refer to *Core: Parameters*.

Automatic Propagation of Annotation Elements

Automatic local propagation of annotation elements can be done based on some specified conditions, using a Pro/TOOLKIT application. A Pro/TOOLKIT application can register the following notification event types (`ProNotifyType`):

- `PRO_FEATURE_CREATE_POST`
- `PRO_FEATURE_REDEFINE_POST`

When an appropriate event occurs during a Pro/ENGINEER session, the associated notification function can invoke a local propagation.

Functions Introduced:

- **`ProAnnotationelemAutopropagate()`**

The function **`ProAnnotationelemAutopropagate()`** causes the local and automatic propagation of annotation elements to the currently selected feature within the same model, after a supported feature has either been created or modified. The propagation behavior is dependant on the standard Pro/ENGINEER algorithm and on the current contents of the selection buffer.

Following are the list of supported features:

- Draft
 - Surface
 - Solid
- Offset
 - Surface
 - With Draft
 - Expand
- Mirror Surface
- Copy Surface
- Move Surface

The Pro/TOOLKIT application can be written so as to specify the condition for the automatic propagation, based on created feature-type, subtype or any other required properties.

The function propagates based on the current contents of the selection buffer.

For notification of type PRO_FEATURE_REDEFINE_POST, Pro/ENGINEER does not automatically populate the selection buffer with the feature that was redefined. The Pro/TOOLKIT application would be then required to populate the selection buffer with the feature using the appropriate functions. Refer to the section Selection Buffer in the chapter User Interface: Selection for more information on selection buffer.

Note: The function **ProAnnotationelemAutopropagate()** works regardless of the current value for the configuration option, `auto_propagate_ae = YES/NO`. PTC advises that the Pro/TOOLKIT application respect the current value of this configuration option; otherwise, duplicate versions of the propagated annotations can result.

Example 1: Local Autopropagation of Annotation Elements

This example demonstrates how to enable automatic, same model propagation of Annotation Elements to supported features after feature creation.

```
#include <ProToolkit.h>
#include <ProFeature.h>
#include <ProTMC_LevelA3X.h>
#define NO_AUTOPROP_STRING "NO_AUTOPROP"

/*=====*\
FUNCTION :   PTAFExAEAAutopropagate()
PURPOSE  :   Locally autopropagate after feature creation, unless the
              feature name contains the string "NO_AUTOPROP"
\*=====*/

ProError PTAFExAEAAutopropagate (ProFeature* feature)
{
    ProFeattype f_type;
    ProName f_name;
    ProCharName c_name;
    ProError status;
    char* result;

    /*-----*\
Check if the feature type created supports local propagation.
\*-----*/
    status = ProFeatureTypeGet (feature, &f_type);
    PT_TEST_LOG_SUCC ("ProFeatureTypeGet()");
```

```

if (f_type == PRO_FEAT_DRAFT ||
    f_type == PRO_FEAT_SRF_MDL ||
    f_type == PRO_FEAT_OFFSET ||
    f_type == PRO_FEAT_DATUM_SURF)
{
    status = ProModelitemNameGet (feature, f_name);
    PT_TEST_LOG_SUCC ("ProModelitemNameGet()");

/*-----*\
If the feature has a name, check if it contains the "no autoprop"
keystring.
\*-----*/
    if (status == PRO_TK_NO_ERROR)
    {
        ProWstringToString (c_name, f_name);

        result = strstr (c_name, NO_AUTOPROP_STRING);

        if (result != NULL)
            status = PRO_TK_NO_CHANGE;
    }
    else
        status = PRO_TK_NO_ERROR;

/*-----*\
If the feature permits it, autopropagate any annotation features onto
this feature. The feature is already selected in the selection buffer
during the POST create notification action.
\*-----*/
    if (status == PRO_TK_NO_ERROR)
    {
        status = ProAnnotationelemAutopropagate ();
        PT_TEST_LOG_SUCC ("ProAnnotationelemAutopropagate()");
    }
}

return (PRO_TK_NO_ERROR);
}

```

Access to Annotations

The structure for the annotations is similar to the structure ProModelitem and is defined as

```

typedef struct pro_model_item
{
    ProType type;

```

```

    int id;

    ProMdl owner;
}ProAnnotation

```

The value of *type* for different annotations is as follows:

- PRO_NOTE—Specifies a note. Functions specific to notes use the object type *ProNote*.
- PRO_SYMBOL_INSTANCE—Specifies a symbol instance. Functions specific to symbols use the object *ProDtlsyminst*.
- PRO_GTOL—Specifies a geometric tolerance. Functions specific to Gtols use the object *ProGtol*.
- PRO_SURF_FIN—Specifies a surface finish. Functions specific to surface finish use the object *ProSurfFinish*.
- PRO_REF_DIMENSION—Specifies a reference dimension.
- PRO_DIMENSION—Specifies a driving or driven dimension. Reference, driven and driving dimension functions may use the object type *ProDimension*.
- PRO_SET_DATUM_TAG—Specifies a set datum tag annotation. Functions specific to set datum tag use the object type *ProSetdatumtag*.
- PRO_CUSTOM_ANNOTATION—Specifies a custom annotation type. Currently, used only for manufacturing template annotations.

Functions Introduced:

- **ProAnnotationElementGet()**
- **ProAnnotationShow()**
- **ProAnnotationErase()**
- **ProAnnotationIsShown()**
- **ProAnnotationIsInactive()**
- **ProAnnotationDisplay()**
- **ProAnnotationUndisplay()**
- **ProAnnotationUpdate()**
- **ProFeatureParamsDisplay()**

The function **ProAnnotationElementGet()** returns the annotation element that contains the annotation.

The function **ProAnnotationShow()** specifies that the annotation be shown. The annotation will be visible until it is explicitly erased from the model database. This function is also capable of showing an annotation in a drawing view similar to the Pro/ENGINEER command “Show and Erase”. This function supersedes the functions **ProDimensionShow()** and **ProNoteDisplay()**.

The function **ProAnnotationErase()** erases the annotation from the model database. The annotation will become visible again only if it is explicitly set to be shown using **ProAnnotationShow()**. This function supersedes the functions **ProDimensionErase()** and **ProNoteUndisplay()**.

The function **ProAnnotationIsShown()** returns the display status of the annotation. It specifies whether the annotation is shown in a particular model or drawing.

The function **ProAnnotationIsInactive()** indicates whether the annotation can be shown or not. An annotation becomes inactive if all the weak references it points to have been lost or consumed.

The functions **ProAnnotatonDisplay()** and **ProAnnotationUndisplay()** should be used together. To edit a shown annotation, it must be first undisplayed using the function **ProAnnotationUndisplay()** followed by the editing function calls, and finally must be redisplayed using the function **ProAnnotationDisplay()**, so that the updated annotation is correctly visible to the user.

The function **ProAnnotationUpdate()** updates the display of an annotation, but does not actually display it. If the annotation is not currently displayed (because it is hidden by layer status or inactive geometry), the text extracted from the annotation with the mode `PRODISPMODE_NUMERIC` may include callout symbols, instead of the text shown to the user. **ProAnnotationUpdate()** informs Pro/ENGINEER to update the contents of the annotation in order to cross-reference these callout values. This function supports 3D model notes of the type *ProNote* and detail notes of the type *ProDtlnote*.

When you want to force the display of dimensions or parameters, geometric tolerances (gtols), and so on on a single feature, use the function **ProFeatureParamsDisplay()**.

Note:

- The function **ProAnnotationDisplay()** supersedes the functions **ProDimensionShow()** and **ProGtolShow()**.

- The function **ProAnnotationUndisplay()** supersedes the functions **ProDimensionErase()** and **ProGtolErase()**.
- The function **ProDimensionDisplayUpdate()** is superseded by a combination of **ProAnnotationDisplay()** and **ProAnnotationUndisplay()**.

Annotation Orientation

An Annotation Orientation refers to the annotation plane or the parallel plane in which the annotation lies, the viewing direction, and the text rotation. You can define the annotation orientation using a datum plane or flat surface, a named view, or as flat to screen. If the orientation is defined by a datum plane, you can set its reference to “frozen” or driven; where frozen indicates that the reference to the datum plane or named view has been removed.

Functions Introduced:

- **ProAnnotationplaneCreate()**
- **ProAnnotationplaneFromViewCreate()**
- **ProAnnotationplaneFlatToScreenCreate()**
- **ProAnnotationplaneTypeGet()**
- **ProAnnotationplaneReferenceGet()**
- **ProAnnotationplanePlaneGet()**
- **ProAnnotationplaneVectorGet()**
- **ProAnnotationplaneFrozenGet()**
- **ProAnnotationplaneFrozenSet()**
- **ProAnnotationplaneForcetoplaneflagGet()**
- **ProAnnotationplaneForcetoplaneflagSet()**
- **ProAnnotationplaneViewnameGet()**
- **ProAnnotationplaneAngleGet()**
- **ProAnnotationRotate()**

The function **ProAnnotationplaneCreate()** creates a new annotation plane from either a datum plane, a flat surface, or an existing annotation that already contains an annotation plane.

The function **ProAnnotationplaneFromViewCreate()** creates a new annotation plane from a saved model view.

The function **ProAnnotationplaneFlatToScreenCreate()** returns the annotation plane item representing a flat-to-screen annotation in the model. This function takes a *ProBoolean* input argument *by_screen_point*, which identifies whether the annotations on this plane are located by screen points, or by model units.

Note: You can only place notes, surface finishes, and symbols as flat to screen. Dimensions, geometric tolerances and set datum tags are not supported as flat-to-screen annotations.

Use the function **ProAnnotationplaneTypeGet()** to obtain the annotation plane type. It can have one of the following values:

- **PRO_ANNOTATIONPLANE_REFERENCE**—The annotation plane is created from a datum plane or a flat surface, and can be frozen or be associative to the reference.
- **PRO_ANNOTATIONPLANE_NAMED_VIEW**—The annotation plane is created from a named view or a view in the drawing.
- **PRO_ANNOTATIONPLANE_FLATTOSCREEN_BY_MODEL_PNT**—The annotation plane is flat-to-screen and annotations are located by model units.
- **PRO_ANNOTATIONPLANE_FLATTOSCREEN_BY_SCREEN_PNT**—The annotation plane is flat-to-screen and annotations are located by screen points.
- **PRO_ANNOTATIONPLANE_FLATTOSCREEN_LEGACY**—The annotation uses a legacy flat-to-screen format (located in model space).

The function **ProAnnotationplaneReferenceGet()** returns the planar surface used as the annotation plane.

The function **ProAnnotationplanePlaneGet()** returns the geometry of the annotation plane in terms of the *ProPlanedata* object containing the origin and orientation of the annotation plane.

The functions **ProAnnotationplaneFrozenGet()** and **ProAnnotationplaneFrozenSet()** determine and assign, respectively, whether the annotation orientation is frozen or driven by reference to the plane geometry. These functions are applicable only for annotation planes governed by references.

The functions **ProAnnotationplaneForcetoplaneflagGet()** and **ProAnnotationplaneForcetoplaneflagSet()** return and assign, respectively, the value of the *ProBoolean* argument *force_to_plane* for an annotation plane. If this argument is set to `PRO_B_TRUE`, then the annotations that reference the annotation plane are placed on that plane. If the annotation orientation is not frozen, that is, driven by the reference plane, and if the reference plane is moved, then the annotations also move along with the plane.

The function **ProAnnotationplaneViewnameGet()** obtains the name of the view that was originally used to determine the orientation of the annotation plane.

Note: If the named view orientation has been changed after the annotation plane was created, the orientation of the plane will not match the current orientation of the view.

The function **ProAnnotationplaneVectorGet()** returns the normal vector that determines the viewing direction of the annotation plane.

The function **ProAnnotationplaneAngleGet()** returns the current rotation angle in degrees for a given annotation plane and the text orientation of all annotations on that plane.

The function **ProAnnotationRotate()** rotates a given annotation by the specified angle. This moves the annotation to a new annotation plane with the appropriate rotation assigned. Other annotations on the annotation's current plane are unaffected by this function.

Note: You can only rotate annotations that belong to annotation elements using the above function.

Annotation Associativity

The functions described in this section allow you to ensure associativity between shown annotations in drawings and 3D models. You can independently control the position associativity and attachment associativity of a drawing annotation.

Note:

- Drawing annotations can have only uni-directional associativity, that is, changes to the position and attachment of the annotation in the 3D model are reflected for the annotation in the drawing view, but not vice-versa.

- You cannot modify the position associativity and attachment associativity of a drawing annotation from the 3D model.
- You cannot make free, flat-to-screen annotations in a drawing view associative to the annotations in the 3D model.
- Annotation properties such as text, jogs, breaks, skew, witness line clippings, and flip arrow states are not associative.

Functions Introduced:

- **ProAnnotationIsAssociative()**
- **ProAnnotationPositionUpdate()**
- **ProAnnotationAttachmentUpdate()**

The function **ProAnnotationIsAssociative()** identifies if a given annotation in a drawing view is associative to the annotation in the 3D model. It has the following output arguments:

- *assoc_position*—Specifies if the position of the annotation is associative.
- *assoc_attach*—Specifies the attachment associativity. It takes one of the following values:
 - **PRO_ANNOTATTACH_ASSOCIATIVITY_PARTIAL**—Specifies that the drawing annotation is partially associative.
 - **PRO_ANNOTATTACH_ASSOCIATIVITY_FULL**—Specifies that the drawing annotation is fully associative.
- *future_use*—This argument is for future use.

The function **ProAnnotationPositionUpdate()** updates the position of the drawing annotation, and makes it associative to the position of the annotation in the 3D model.

Note: You can update the associative position only for drawing annotations that have their placement based on model coordinates.

The function **ProAnnotationAttachmentUpdate()** updates the attachment of the drawing annotation, and makes it associative to the attachment of the annotation in the 3D model. Associative attachment of an annotation refers to both – its references and its attachment point on its references.

Note: You can update the associative attachment only for drawing annotations that are attached to the geometry.

Interactive Selection

Annotation features, annotation elements, and annotations can be selected interactively using **ProSelect()** or can be obtained from the selection buffer using the function **ProSelbufferSelectionsGet()**.

For more information on interactive selection refer to chapter User Interface: Selection.

Display Modes

Functions Introduced:

- **ProDisplaymodeGet()**
- **ProDisplaymodeSet()**

These functions specify whether the display of dimensions shows symbols or values, and enables you to switch the mode. This is the equivalent to the Pro/ENGINEER command **Info, Switch Dim.**

Designating Dimensions and Symbols

Function Introduced:

- **ProSymbolDesignate()**
- **ProSymbolUndesignate()**
- **ProSymbolDesignationVerify()**

The function **ProSymbolDesignate()** designates a dimension, dimension tolerance, geometric tolerance or surface finish symbol to the specified model.

The function **ProSymbolUndesignate()** undesignates the dimension, dimension tolerance, geometric tolerance or surface finish symbol from the specified model.

The function **ProSymbolDesignationVerify()** determines if a dimension, dimension tolerance, geometric tolerance or surface finish symbol has been designated to a model.

Dimensions

The ProDimension Object

The ProDimension object handle is a DHandle that is equivalent to ProModelitem. The owner field can be a solid or a drawing, depending upon where the dimension is stored. Dimensions created in drawing mode may be stored in the drawing or in the solid depending upon the setting of the *config.pro* option CREATE_DRAWING_DIMS_ONLY. The type field is either PRO_DIMENSION or PRO_REF_DIMENSION. The ID is the integer used to identify the dimension inside Pro/ENGINEER. It corresponds to the numerical part of the default symbol assigned to the dimension when it is created.

The ProDimension object also inherits from ProModelitem, which means that functions such as **ProModelitemInit()** and **ProSelectionModelitemGet()** can be used for it (**ProDimensionSymbolGet()** and **ProDimensionSymbolSet()** are recommended for this purpose, instead of **ProModelitemNameGet()** and **ProModelitemNameSet()**).

Visiting Dimensions

Functions Introduced:

- **ProSolidDimensionVisit()**
- **ProDrawingDimensionVisit()**
- **ProDimensionSymbolGet()**
- **ProDimensionValueGet()**
- **ProDimensionTypeGet()**
- **ProDimensionNomvalueGet()**
- **ProDimensionIsDisplayRoundedValue()**
- **ProDimensionDisplayRoundedValueSet()**
- **ProDimensionDisplayedValueGet()**
- **ProDimensionOverridevalueGet()**
- **ProDimensionValuedisplayGet()**
- **ProDimensionIsFractional()**
- **ProDimensionDecimalsGet()**
- **ProDimensionDenominatorGet()**

- **ProDimensionIsReldriven()**
- **ProDimensionIsRegenednegative()**
- **ProDimensionBoundGet()**
- **ProDimensionFeatureGet()**
- **ProDimensionIsAccessibleInModel()**

The two visit functions **ProSolidDimensionVisit()** and **ProDrawingDimensionVisit()** conform to the usual style of visit functions. (Refer to section Visit Functions in the chapter Fundamentals.) A dimension is stored in a solid if it is a “shown” dimension, that is, if it was created automatically by Pro/ENGINEER as part of the feature definition. A dimension will also be stored in a solid if it was created in drawing mode while the *config.pro* option **CREATE_DRAWING_DIMS_ONLY** was set to **NO**.

The function **ProDimensionSymbolGet()** returns the symbol (the name) of the specified dimension.

The function **ProDimensionValueGet()** returns the value of the dimension.

The function **ProDimensionTypeGet()** returns the type of the dimension in terms of the following values:

- **LINEAR**
- **RADIUS**
- **DIAMETER**
- **ANGLE**

The function **ProDimensionNomvalueGet()** returns the nominal value of a dimension. When passed a dimension, the function returns the nominal value even if the dimension is set to the upper or lower bound.

Use the function **ProDimensionIsDisplayRoundedValue()** to determine whether the specified dimension is set to display its rounded off value.

In Pro/TOOLKIT, a rounded off value is a decimal value that contains only the desired number of digits after the decimal point. For example, if a dimension has the stored value 10.34132 and you want to display only two digits after the decimal point, you must round off the stored value to two decimal places. Thus, rounding off converts 10.34132 to 10.34.

Use the function **ProDimensionDisplayRoundedValueSet()** to set the attribute of the given dimension to display either the rounded off value or the stored value. You can use this function for all dimensions except angular dimensions created prior to Pro/ENGINEER Wildfire 4.0, ordinate baseline dimensions, and dimensions of type DIM_IPAR_INT. For these dimensions the functions returns an error status PRO_TK_NOT_VALID.

If you choose to display the rounded off value, the function **ProDimensionDisplayedValueGet()** retrieves the displayed rounded value of the specified dimension. Otherwise, it retrieves the stored value.

The function **ProDimensionOverridevalueGet()** returns the override value for a dimension. The default override value is zero.

Note: The override value is available only for driven dimensions.

Use the function **ProDimensionValuedisplayGet()** to obtain the type of value displayed for a dimension. The valid types are:

- PRO_DIMVALUEDISPLAY_NOMINAL—Displays the actual value of the dimension along with the tolerance value.
- PRO_DIMVALUEDISPLAY_OVERRIDE—Displays the override value for the dimension along with the tolerance value.
- PRO_DIMVALUEDISPLAY_HIDE—Displays only the tolerance value for the dimension.

The function **ProDimensionIsFractional()** returns whether the dimension is expressed in terms of a fraction rather than a decimal. If the dimension is decimal, the function

ProDimensionDecimalsGet() outputs the number of decimals digits that are significant; if the dimension is fractional, the function **ProDimensionDenominatorGet()** returns the value of the largest possible denominator used to define the fractional value.

The function **ProDimensionIsRelldriven()** returns whether the dimension is driven by a relation.

The function **ProDimensionIsRegenednegative()** returns whether the dimension really has a negative value in relation to its original definition. Dimensions are always displayed in Pro/ENGINEER with positive values, and

ProDimensionValueGet() will always return a positive value, so this function is needed to show whether a dimension has been “flipped” as a result of being assigned a negative value during the last regeneration.

The function **ProDimensionBoundGet()** returns the bound status of a dimension.

The function **ProDimensionFeatureGet()** returns the feature that owns the specified dimension.

Note: For dimensions and reference dimensions in annotation elements, the function **ProDimensionFeatureGet()** returns the annotation feature that owns that element.

The function **ProDimensionIsAccessibleInModel()** identifies if a specified dimension is owned by the model. By default, the dimension is accessible in the model.

Example 2: Changing the Displayed Value of Selected Model Dimension to Rounded or Non-Rounded

```
/*-----*\
Pro/Toolkit includes
\*-----*/
#include <ProToolkit.h>
#include <ProObjects.h>
#include <ProFeature.h>
#include <ProDimension.h>
#include <ProSolid.h>
#include <ProUtil.h>
#include <ProMessage.h>
#include <PTApplsUnicodeUtils.h>
/*-----*\
Application includes
\*-----*/
#include <TestError.h>

/*=====*\
FUNCTION : UserDimDisplayRounded()
PURPOSE  : Change the displayed value of a selected model dimension to
           rounded/non-rounded.

\*=====*/
ProError UserDimDisplayRounded()
{
    ProMdl model ;           /* current model */
    ProSelection *p_sel_feat ; /* selected feature */
    ProSelection *p_sel_dim ; /* selected dimension */
    ProModelitem dim_mdllitem ; /* dimension Modelitem */
    int n_sel_feat ;         /* no. of selected features */
    int n_sel_dim ;          /* no. of selected dims */
    ProFileName msg_file ;   /* message file */
    int err ;                /* return status */
    double value ;           /* dimension value */
    double newdimval ;       /* user-entered dim value */
}
```

```

    ProCharLine str1 ;          /* message string */
    ProBoolean display_rounded; /* rounded display status */

    ProStringToWstring(msg_file, "msg_ugfund.txt") ;

/*-----*\
    Check for a model.
\*-----*/
    err = ProMdlCurrentGet(&model);
    ERROR_CHECK("UserDimDisplayRounded()", "ProMdlCurrentGet()", err);
    if (err != PRO_TK_NO_ERROR)
    {
        ProMessageDisplay(msg_file, "USER %0s F",
            "Error getting current model.");
        return(err) ;
    }

/*-----*\
    Get a feature to modify.
\*-----*/
    ProMessageDisplay(msg_file, "USER %0s F", "Select a feature.");

    err = ProSelect("feature", 1, NULL, NULL, NULL, NULL,
        &p_sel_feat, &n_sel_feat) ;
    ERROR_CHECK("UserDimDisplayRounded()", "ProSelect()", err);
    if ( err != PRO_TK_NO_ERROR )
    {
        ProMessageDisplay(msg_file, "USER %0s F",
            "Error or abort during selection.") ;
        return(err) ;
    }

    ProSelectionUnhighlight (p_sel_feat [0]);

/*-----*\
    Show the dimensions on the feature.
\*-----*/
    err = ProFeatureParamsDisplay (p_sel_feat [0], PRO_DIM_PARAM);
    ERROR_CHECK ("UserDimDisplayRounded()", "ProFeatureParamsDisplay()",
err);

/*-----*\
    Get the dimension to be changed.
\*-----*/
    ProMessageDisplay(msg_file, "USER %0s F", "Select a dimension.");

    err = ProSelect("dimension", 1, NULL, NULL, NULL, NULL,
        &p_sel_dim, &n_sel_dim) ;
    ERROR_CHECK("UserDimDisplayRounded()", "ProSelect()", err);
    if ( err != PRO_TK_NO_ERROR )
    {

```



```

        ProMessageDisplay(msg_file, "USER %0s F",
            "Error or abort during selection.") ;
        return(err) ;
    }

    err = ProSelectionModelitemGet(p_sel_dim[0], &dim_mditem);
    ERROR_CHECK("UserDimDisplayRounded()", "ProSelectionModelitemGet()",
        err);

    err = ProDimensionIsDisplayRoundedValue(&dim_mditem,
        &display_rounded );
    ERROR_CHECK("UserDimDisplayRounded()", "prodim_get_dimension()", err);

/*-----*\
    Change the dimension.
\*-----*/
    ProTKSprintf(str1, "Toggling the Display Rounded property." ) ;
    ProMessageDisplay(msg_file, "USER %0s F", str1) ;

    err = ProDimensionDisplayRoundedValueSet ( &dim_mditem,
!display_rounded);
    ERROR_CHECK("UserDimDisplayRounded()", "ProDimensionValueSet()", err);

    err = ProDimensionDisplayUpdate ( &dim_mditem );

    ProMessageDisplay(msg_file, "USER %0s F",
        "Display Rounded property toggled.");

    return(PRO_TK_NO_ERROR) ;
}

```

Modifying Dimensions

Functions Introduced:

- **ProDimensionSymbolSet()**
- **ProDimensionValueSet()**
- **ProDimensionOverridevalueSet()**
- **ProDimensionValuedisplaySet()**
- **ProDimensionDecimalsSet()**
- **ProDimensionDenominatorSet()**
- **ProDimensionBoundSet()**
- **ProDimensionDimensionReset()**
- **ProDimensionBasicSet()**

- **ProDimensionInspectionSet()**
- **ProDimensionElbowlengthSet()**

The function **ProDimensionSymbolSet()** allows you to change the symbol (the name) of a dimension. You can use this function only with solid dimensions.

The function **ProDimensionValueSet()** changes the value of a dimension. It does not allow you to change the value of any dimension whose value is driven in some other way, for example, a driven or reference dimension or a dimension that is driven by a relation.

The function **ProDimensionOverridevalueSet()** assigns the override value for a dimension. This value is restricted to real numbers. The default override value is zero.

Note: You can set the override value only for driven dimensions.

The function **ProDimensionValuedisplaySet()** sets the type of value to be displayed for a dimension.

The functions **ProDimensionDecimalsSet()** and **ProDimensionDenominatorSet()** set the displayed resolution of the dimension, depending upon whether the display is decimal or fractional. You can use these functions only with solid dimensions.

You can use the function **ProDimensionDecimalsSet()** only in decimal mode, while the function **ProDimensionDenominatorSet()** only in fraction mode to set the displayed resolution of solid dimensions.

When you call the function **ProDimensionDecimalsSet()**:

- If the attribute to display the rounded-off value is set, the displayed value gets rounded-off to the specified number of digits and the stored value remains unchanged.
- If the attribute to display the rounded-off value is not set or is non-applicable and the number of decimals specified in **ProDimensionDecimalsSet()** is less than the existing number of decimals, the stored value of the dimension is rounded-off and triggers a regeneration of the model.

Note: The input, *decimals* to the **ProDimensionDecimalsSet()** function should be a non-negative number and it should be such that when you apply either the upper or the lower values of the tolerance to the given dimension, the total number of digits before and after the decimal point in the resulting values does not exceed 13.

When you call the function **ProDimensionDenominatorSet()**:

- The stored value remains unchanged if,
 - it can be expressed as an exact fraction with the given denominator, regardless of whether the round-off attribute is set or not.
 - the stored value cannot be expressed as an exact fraction, but the round-off attribute is set. In this case, the fraction is the approximate representation of the stored value.
- The stored value changes to the nearest fraction and triggers a regeneration of the model, if it cannot be expressed as an exact fraction with the given denominator and the round-off attribute is not set.

The functions **ProDimensionBoundSet()** sets the bound status of the dimension.

The function **ProDimensionDimensionReset()** sets the dimension to the value it had at the end of the last successful regeneration. You can use this function to recover from a failed regeneration.

The function **ProDimensionBasicSet()** and the function **ProDimensionInspectionSet()** set the basic and inspection notations of the dimension. These functions are applicable to both driven and driving dimensions.

Note: The basic and inspection notations of the dimension are not available when only the tolerance value for a dimension is displayed.

The function **ProDimensionElbowlengthSet()** assigns the elbow length for the dimension.

Example 3: Modifying a Dimension

This example demonstrates how to modify a dimension. The dimensions belonging to a given feature are displayed. You can choose the dimension you wish to modify.

```
/*-----*\
    Pro/Toolkit includes
\*-----*/
#include <ProToolkit.h>
#include <ProObjects.h>
#include <ProFeature.h>
#include <ProDimension.h>
#include <PTApplsUnicodeUtils.h>
/*-----*\
    Application includes
\*-----*/
#include <TestError.h>

/*=====*\
FUNCTION : UserDimensionChange()
PURPOSE  : Change a selected model dimension.
\*=====*/
int UserDimensionChange()
{
    ProMdl model ;                /* current model */
    ProSelection *p_sel_feat ;    /* selected feature */
    ProSelection *p_sel_dim ;    /* selected dimension */
    ProModelitem dim_mdllitem ;  /* dimension Modelitem */
    int n_sel_feat ;             /* no. of selected features */
    int n_sel_dim ;              /* no. of selected dims */
    ProFileName msg_file ;       /* message file */
    int err ;                     /* return status */
    double value ;                /* dimension value */
    double newdimval ;            /* user-entered dim value */
    ProCharLine str1 ;            /* message string */

    ProStringToWstring(msg_file, "msg_ugfund.txt") ;

/*-----*\
    Check for a model.
\*-----*/
    err = ProMdlCurrentGet(&model);
    ERROR_CHECK("UserDimensionChange()", "ProMdlCurrentGet()", err);
    if (err != PRO_TK_NO_ERROR)
    {
        ProMessageDisplay(msg_file, "USER %0s F",
            "Error getting current model.");
        return(err) ;
    }
/*-----*\
    Get a feature to modify.
```

```

\*-----*/
    ProMessageDisplay(msg_file, "USER %0s F", "Select a feature.");

    err = ProSelect("feature", 1, NULL, NULL, NULL, NULL,
        &p_sel_feat, &n_sel_feat) ;
    ERROR_CHECK("UserDimensionChange()", "ProSelect()", err);
    if ( err != PRO_TK_NO_ERROR )
    {
        ProMessageDisplay(msg_file, "USER %0s F",
            "Error or abort during selection.") ;
        return(err) ;
    }

    ProSelectionUnhighlight (p_sel_feat [0]);
/*-----*\
    Show the dimensions on the feature.
\*-----*\
    err = ProFeatureParamsDisplay (p_sel_feat [0], PRO_DIM_PARAM);
    ERROR_CHECK ("UserDimensionChange()", "ProFeatureParamsDisplay()",
err);
/*-----*\
    Get the dimension to be changed.
\*-----*\
    ProMessageDisplay(msg_file, "USER %0s F", "Select a dimension.");

    err = ProSelect("dimension", 1, NULL, NULL, NULL, NULL,
        &p_sel_dim, &n_sel_dim) ;
    ERROR_CHECK("UserDimensionChange()", "ProSelect()", err);
    if ( err != PRO_TK_NO_ERROR )
    {
        ProMessageDisplay(msg_file, "USER %0s F",
            "Error or abort during selection.") ;
        return(err) ;
    }
err = ProSelectionModelitemGet (p_sel_dim[0], &dim_mdllitem);
    ERROR_CHECK("UserDimensionChange()", "ProSelectionModelitemGet()",
        err);

    err = ProDimensionValueGet(&dim_mdllitem, &value );
    ERROR_CHECK("UserDimensionChange()", "prodim_get_dimension()", err);
/*-----*\
    Change the dimension.
\*-----*\
    ProTKSprintf(str1, "Dimension value: %lf ... Enter new value: ",
        value) ;
    ProMessageDisplay(msg_file, "USER %0s F", str1) ;
    ProMessageDoubleRead(NULL, &newdimval) ;

    err = ProDimensionValueSet ( &dim_mdllitem, newdimval);
    ERROR_CHECK("UserDimensionChange()", "ProDimensionValueSet()", err);

```

```

err = ProDimensionDisplayUpdate ( &dim_mdldata );

ProMessageDisplay(msg_file, "USER %0s F",
    "Dimension changed. Regenerate model to see changes.");

return(PRO_TK_NO_ERROR) ;
}

```

Dimension Tolerances

Functions Introduced:

- **ProToleranceDefaultGet()**
- **ProDimensionDisplayedToleranceGet()**
- **ProSolidToleranceGet()**
- **ProSolidToleranceSet()**
- **ProDimensionIsToleranceDisplayed()**
- **ProDimensionToltypeGet()**
- **ProDimensionToltypeSet()**
- **ProDimensionToleranceGet()**
- **ProDimensionToleranceSet()**
- **ProDimensionTolerancedecimalsGet()**
- **ProDimensionTolerancedecimalsSet()**
- **ProDimensionTolerancedenominatorGet()**
- **ProDimensionTolerancedenominatorSet()**

The function **ProToleranceDefaultGet()** tells you the current default value for a given tolerance, that is, the value set in the configuration file to be used for new models. There is a separate tolerance for linear and angular dimensions (PROTOLERANCE_LINEAR and PROTOLERANCE_ANGULAR) and for each number of decimal places in the range 1 to 6 (12 tolerance settings in all).

If the round off attribute for the given dimension is set, the function **ProDimensionDisplayedToleranceGet()** retrieves the displayed rounded values of the upper and lower limits of the specified dimension. Otherwise, it retrieves the stored values of the tolerances as done by the function **ProSolidToleranceGet()**. For example, consider a dimension that is set to round off to two

decimal places and has the upper and lower tolerances 0.123456. By default, the tolerance values displayed are also rounded off to two decimal places. In this case, the function **ProDimensionDisplayedToleranceGet()** retrieves the upper and lower values as 0.12.

The input argument of the function **ProDimensionDisplayedToleranceGet()** is a dimension handle. The output arguments of this function are pointers to the rounded values of the upper and lower limits of the specified dimension. You must allocate a memory location for each of the output arguments. Pass a NULL pointer if you do not want to use an output argument. You cannot pass a null for both the output arguments.

The functions **ProSolidToleranceGet()** and **ProSolidToleranceSet()** let you find and set the current value of the specified dimensional tolerance. The function **ProDimensionIsToleranceDisplayed()** tells you whether the tolerances of the specified dimension are currently being displayed. The display of tolerances can be turned on by the Pro/ENGINEER user by setting Tol, ON in the environment, and turned off for individual dimensions by using **MODIFY, DIM COSMETIC, FORMAT, NOMINAL**.

The function **ProDimensionToltypeGet()** returns the display format for the specified dimension tolerance. The valid values are:

- **PRO_TOL_DEFAULT**—Displays dimensions without tolerances. Similar to the nominal option in Pro/ENGINEER.
- **PRO_TOL_LIMITS**—Displays dimension tolerances as upper and lower limits.

Note: This format is not available when only the tolerance value for a dimension is displayed.
- **PRO_TOL_PLUS_MINUS**—Displays dimensions as nominal with plus-minus tolerances. The positive and negative values are independent.
- **PRO_TOL_PLUS_MINUS_SYM**—Displays dimensions as nominal with a single value for both the positive and the negative tolerance.
- **PRO_DIM_TOL_SYM_SUPERScript**—Displays dimensions as nominal with a single value for positive and negative tolerance. The text of the tolerance is displayed in a superscript format with respect to the dimension text.

Use the function **ProDimensionToltypeSet()** to set the display format for the specified dimension tolerance.

The function **ProDimensionToleranceGet()** reports the deviation of the upper and lower tolerances from the nominal value. The values are always reported as independent upper and lower tolerance values; the actual display of the tolerance is determined by **ProDimensionToltypeGet()**. Tolerances are not applicable for reference dimensions.

Note: If the upper tolerance value is negative, it will be displayed with a '-' sign. But if the lower tolerance value is negative, it will be displayed with a '+' sign.

The function **ProDimensionToleranceSet()** sets the upper and lower tolerance limits. The values are always accepted as independent upper and lower tolerance values; the actual display of the tolerance is determined by **ProDimensionToltypeSet()**. It is not applicable to reference dimensions.

The functions **ProDimensionTolerancedecimalsGet()** and **ProDimensionTolerancedecimalsSet()** obtain and assign the number of decimal places shown for the upper and lower values of the dimension tolerance. Thus, the dimension tolerance decimals can be set independent of the number of dimension decimals. By default, the number of decimal places for tolerance values is calculated based upon the "linear_tol" settings of the model.

Note: The input, *tolerance_decimals* to the **ProDimensionTolerancedecimalsSet()** function should be a non-negative number and it should be such that when you apply either the upper or the lower values of the tolerance to the given dimension, the total number of digits before and after the decimal point in the resulting values does not exceed 13.

If the dimension tolerance value is a fraction, the functions **ProDimensionTolerancedenominatorGet()** and **ProDimensiondenominatorSet()** obtain and assign the value for the largest possible denominator for the upper and lower tolerance values. By default, this value is defined by the *config.pro* option, *dim_fraction_denominator*.

ISO/DIN Tolerance Table Use

Functions Introduced:

- **ProSolidModelclassGet()**
- **ProSolidModelclassSet()**
- **ProSolidTolclassLoad()**
- **ProDimensionTollabelGet()**
- **ProDimensionTollabelSet()**

Pro/TOOLKIT provides functions that programmatically set and return ISO/DIN tolerance table data. These functions allow changes to values before the label is set. For all other labels, use the **ProDimensionTollabelSet()** command.

The functions **ProSolidModelclassGet()** and **ProSolidModelclassSet()** respectively return or set the type of tolerance to use for a particular model. Valid settings are:

- COARSE
- FINE
- MEDIUM
- VERY_COARSE

The function **ProSolidTolclassLoad()** loads a hole or shaft ISO/DIN tolerance table into the current session memory.

The functions **ProDimensionTollabelGet()** and **ProDimensionTollabelSet()** respectively get or set the ISO/DIN tolerance table assigned to the specified dimension.

Dimension Text

Functions Introduced:

- **ProDimensionTextGet()**
- **ProDimensionTextSet()**

The function **ProDimensionTextGet()** retrieves the text of the specified dimension, if it exists.

The function **ProDimensionTextSet()** sets the text of the specified dimension. This is equivalent to the Pro/ENGINEER command **Modify, Text**.

Dimension Text Style

Functions Introduced:

- **ProDimensionTextstyleGet()**
- **ProDimensionTextstyleSet()**

The function **ProDimensionTextstyleGet()** returns the text style assigned to a specified dimension or reference dimension.

The function **ProDimensionTextstyleSet()** sets the text style assigned to a specified dimension or reference dimension.

Note: Only some of the text style properties may be assigned to dimensions.

Dimension Location

The functions described in this section extract the dimension location and geometry in 3D space for solid model dimensions.

Functions Introduced:

- **ProDimensionLocationGet()**
- **ProDimlocationFree()**
- **ProDimensionMove()**

The function **ProDimensionLocationGet()** returns the location of the elements that make up a solid dimension or reference dimension.

This function optionally takes a view used to determine the orientation of the model when calculating the dimension locations. The orientation often determines the text location, presence or absence of elbows, and other dimension location properties.

Use the function **ProDimlocationFree()** to free the structure containing the dimension location data.

The function **ProDimensionMove()** enables you to move the specified dimension to the given location within its owner model.

Dimension Entity Location

The following functions extract the locations of geometric endpoints for the dimension. You can calculate the dimension location plane, witness line, and dimension orientation vectors from these points. The location of the points is specified in the same coordinate system as the solid model.

Functions Introduced:

- **ProDimlocationTextGet()**
- **ProDimlocationArrowsGet()**
- **ProDimlocationWitnesslinesGet()**
- **ProDimlocationArrowtypesGet()**
- **ProDimlocationZExtensionlinesGet()**
- **ProDimlocationNormalGet()**

The function **ProDimlocationTextGet()** returns the location of the dimension text in model coordinates. If the dimension contains an elbow, the function returns the location of the elbow joint and the length of the elbow in model coordinates.

Note: If the value of the elbow length was originally set by the function **ProDimensionElbowlengthSet()**, then the value returned by **ProDimlocationTextGet()** is equal to the value set by the function **ProDimensionElbowlengthSet()** minus the padding around the text for all dimension types.

The function **ProDimlocationArrowsGet()** returns the location of the arrow heads for the dimension.

The function **ProDimlocationWitnesslinesGet()** returns the location of the witness line ends for the dimension.

The function **ProDimlocationArrowtypesGet()** returns the type of arrows used for the leader of a specified 3D dimension. The dimension location obtained using the function **ProDimensionLocationGet()** serves as an input argument for this function.

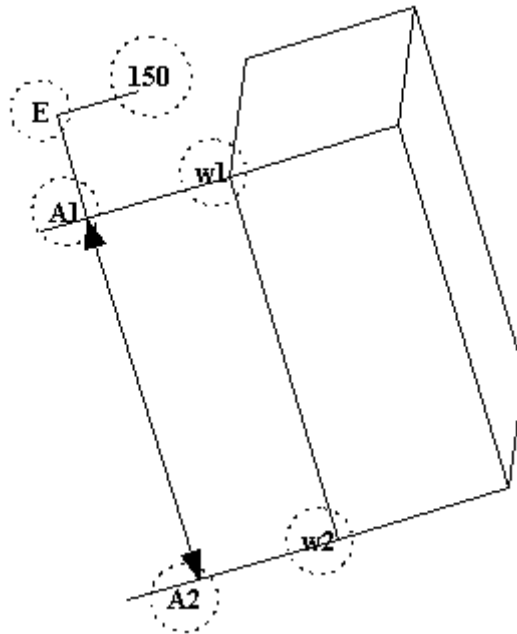
Note: In case of radial dimensions where the arrow head type cannot be changed, the function **ProDimlocationArrowtypesGet()** always returns the “Arrow head” leader type.

The function **ProDimlocationZExtensionlinesGet()** obtains the endpoints of the Z-extension line created for a specified dimension. Z-extension lines are automatically created whenever the dimension’s attachment does not intersect its reference in the Z-Direction. The Z-extension line is attached at the edge of the surface at the closest distance from the dimension witness line.

The function **ProDimlocationNormalGet()** returns the vector normal to the dimensioning plane for a radial or diameter dimension. This normal vector should correspond to the axis normal to the arc being measured by the radial or diameter dimension.

The following figures illustrate the potential location of the arrow heads and witness lines for different dimension types.

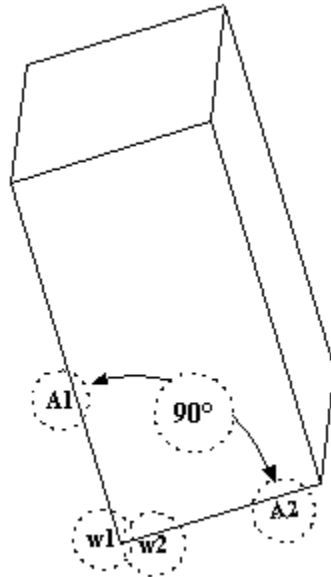
Figure 22-1: Linear and psuedo-linear dimensions



For a linear type of dimension, there are typically two arrow locations A1 and A2 as shown in Linear and psuedo-linear dimensions. w1 and w2 indicate the two witness line locations.

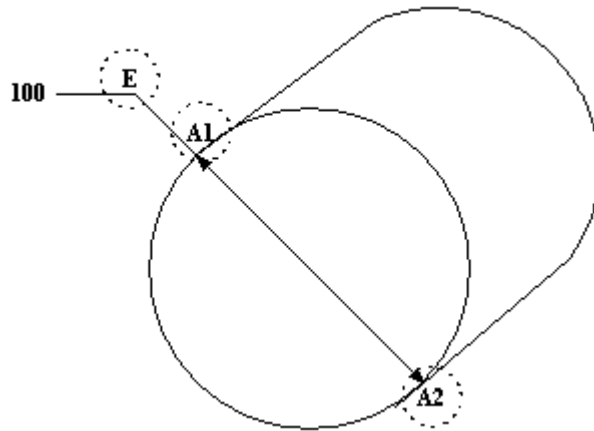
If the dimension type has an elbow joint indicated by E, the elbow length is the distance between the text and E. If the dimension does not have an elbow, the text occurs on the line between A1 and A2, and its position is returned by the function **ProDimensionTextGet()**. Pattern parameter dimensions and length-of-arc dimensions also typically return this dimension structure.

Figure 22-2: Angular dimensions



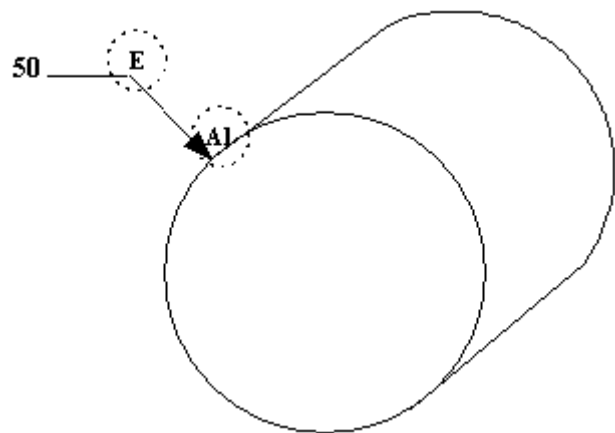
For an angular type of dimension, there are two arrow locations A1 and A2 as shown in Angular dimensions. w1 and w2 indicate the two witness line locations. For some angular dimensions the two witness line endpoints are coincident, but they are returned as independent locations. This dimension type does not have an elbow joint.

Figure 22-3: Diameter dimensions



For a diameter type of dimension, there are two arrow locations A1 and A2 as shown in Diameter dimensions. The elbow joint for this dimension is indicated by E. The elbow length is the distance between the text and the elbow joint. This dimension type does not have any witness line locations.

Figure 22-4: Radius dimensions



For a radius type dimension, there is one arrow location indicated by A1 and an elbow joint indicated by E. The elbow length is the distance between the text and the elbow joint.

The function **ProDimlocationArrowsGet()** returns a NULL value for the second arrow location. This dimension type does not have any witness line locations.

Example 4: Dimension Location Properties

This example demonstrates use of the dimension location properties. It finds linear dimensions that are oriented parallel to a specified direction vector. It shows these dimensions in the Pro/ENGINEER user interface. The orientation of the dimension is determined from the coordinates of the two dimension arrowheads.

```
/*-----*\
  Pro/Toolkit includes
\*-----*/
#include <ProToolkit.h>
#include <ProObjects.h>
#include <ProFeature.h>
#include <ProDimension.h>
#include <ProAxis.h>
#include <ProUtil.h>
#include <ProSolid.h>
#include <ProMessage.h>
#include <ProAnnotation.h>

/*-----*\
  Application includes
\*-----*/
#include <TestError.h>
#include <UtilMath.h>

typedef struct
{
  ProVector vector;          /* The direction that the dimension should
                             be parallel to */
  ProBoolean found;         /* Flag signalling if any matching
                             dimensions were found. */
} DimByVector;

/*=====*\
  FUNCTION : UserDimensionFilterByVector()
  PURPOSE  : Find linear dimensions oriented parallel to vector (a filter
             action)
\*=====*/
ProError UserDimensionFilterByVector (ProDimension* dim,
                                     ProAppData appdata)
```

```

{
    ProDimlocation dimloc;
    ProDimensiontype type;
    ProPoint3d arrow_1, arrow_2;
    int i;
    ProError status;
    ProVector dimvector;
    double dotproduct;
    DimByVector* data = (DimByVector*) appdata;

/*-----*\
    Continue only for linear dimensions
\*-----*/
    ProDimensionTypeGet (dim, &type);
    if (type != PRODIMTYPE_LINEAR)
        return (PRO_TK_CONTINUE);

/*-----*\
    Extract the location of the arrows of the dimension
\*-----*/
    status = ProDimensionLocationGet (dim, NULL, NULL, &dimloc);
    if (status != PRO_TK_NO_ERROR)
        return (PRO_TK_CONTINUE);

    ProDimlocationArrowsGet (dimloc, arrow_1, arrow_2);
    ProDimlocationFree (dimloc);

/*-----*\
    Construct, validate, and normalize the dimension direction vector
\*-----*/
    for (i = 0; i < 3; i++)
        dimvector [i] = arrow_2 [i] - arrow_1 [i];

    if (ProUtilVectorLength (dimvector) < EPSM6 )
        return (PRO_TK_CONTINUE);

    ProUtilVectorNormalize (dimvector, dimvector);

/*-----*\
    Check if the angle between the vectors is 0 or 180 degrees
\*-----*/
    dotproduct = ProUtilVectorDot (data->vector, dimvector);

    if (fabs (fabs (dotproduct) - 1.0) < EPSM6 )
    {
        data->found = PRO_B_TRUE;
        return (PRO_TK_NO_ERROR);
    }
    else

```



```

        return (PRO_TK_CONTINUE);
    }

/*=====*\
    FUNCTION : UserDimensionShow()
    PURPOSE  : Show the given dimension (a visit action)
\*=====*/
ProError UserDimensionShow (ProDimension* dim,
                           ProError error_status,
                           ProAppData appdata)

{
    ProAnnotationShow ( (ProAnnotation*)&dim, NULL, NULL);

    return (PRO_TK_NO_ERROR);
}

/*=====*\
    FUNCTION : UserDimensionsByDirectionShow()
    PURPOSE  : Highlight linear dimensions in the feature that are
               oriented parallel to a given vector.
\*=====*/
ProError UserDimensionsByDirectionShow(ProSolid solid,
                                       ProFeature* feature,
                                       ProVector vector)

{
    DimByVector data;
    ProError status;
    ProFileName msg_file;

    ProStringToWstring(msg_file, "msg_ugfund.txt");

    memcpy (data.vector, vector, sizeof (ProVector));
    data.found = PRO_B_FALSE;

    if (feature != NULL)
    {
/*-----*\
        Visit all feature dimensions.
\*-----*/
        status = ProFeatureDimensionVisit (feature,
                                           UserDimensionShow,
                                           UserDimensionFilterByVector,
                                           (ProAppData)&data);
    }
    else
    {
/*-----*\
        Visit all model dimensions.
\*-----*/

```

```

        status = ProSolidDimensionVisit (solid,
                                         PRO_B_FALSE,
                                         UserDimensionShow,
                                         UserDimensionFilterByVector,
                                         &data);
    }
    /*-----*\
    Return based on the success of the visit.
    \*-----*/
    if (status != PRO_TK_NO_ERROR || data.found == PRO_B_FALSE)
    {
        ProMessageDisplay(msg_file, "USER %0s",
                          "No dimensions found parallel to the supplied direction.");
        return (PRO_TK_E_NOT_FOUND);
    }
    else
    {
        ProMessageDisplay(msg_file, "USER %0s",
                          "Showing dimensions parallel to the supplied direction.");

        return(PRO_TK_NO_ERROR);
    }
}

```

Dimension Orientation

Functions Introduced:

- **ProDimensionPlaneSet()**
- **ProDimensionPlaneGet()**

The function **ProDimensionPlaneSet()** assigns an annotation plane as the orientation of a specified dimension stored in an annotation element.

The function **ProDimensionPlaneGet()** obtains the orientation of a specified dimension stored in an annotation element.

Note: For dimensions that are not assigned a specific orientation, the orientation obtained includes PRO_ID_NO_ANNOTATION_PLANE as its ID.

Driving Dimension Annotation Elements

You can convert driving dimensions created by features into annotation elements and place them on annotation planes. However, you can create the driving dimension annotation elements only in the features that own the dimensions. These annotation elements cannot have any user defined or system references.

In Pro/TOOLKIT, a driving dimension annotation element uses the *ProModelitem* handle with the type field PRO_DIMENSION and the appropriate dimension ID.

Functions such as **ProSolidDimensionVisit()** and **ProFeatureDimensionVisit()** can be used to find both edit dimensions and driving dimension annotation elements.

Functions Introduced:

- **ProDimensionAnnotationelemCreate()**
- **ProDimensionAnnotationelemDelete()**

The function **ProDimensionAnnotationelemCreate()** creates an annotation element for a specified driving dimension, based on the desired annotation orientation.

The function **ProDimensionAnnotationelemDelete()** removes the annotation element containing the driving dimension. It deletes all the parameters and relations associated with the annotation element.

Accessing Reference and Driven Dimensions

The functions described in this section provide additional access to reference and driven dimension annotations.

Many functions listed in the previous sections that are applicable for driving dimensions are also applicable for reference and driven dimensions.

Functions Introduced:

- **ProDimensionIsDriving()**
- **ProDimensionCreate()**
- **ProDimensionAttachmentsGet()**
- **ProDimensionAttachmentsSet()**
- **ProDimensionDelete()**
- **ProDimensionCanRegen()**

The function **ProDimensionIsDriving()** determines if a dimension is driving geometry or is driven by it. If a dimension drives geometry, its value can be modified and the model regenerated with the given change. If a dimension is driven by geometry, its value is fixed but it can be deleted and redefined as necessary. A driven dimension may also be included in an annotation element.

The function **ProDimensionCreate()** creates a new driven or reference dimension. Specify the geometric references and parameters required to construct the required dimension as the input parameters for this function. The input arguments of this function are as follows:

- *model*—Specifies the solid model.
- *dimension_type*—Specifies the type of dimension. This parameter can have the following values:
 - PRO_REF_DIMENSION—Specifies a reference dimension.
 - PRO_DIMENSION—Specifies a driven dimension.
- *annotation_plane*—Specifies the annotation plane for the dimensions.
- *attachments_arr*—Specifies the points on the model where you want to attach the dimension.

Note: The attachments structure is an array of two ProSelection entities. It is provided to support options such as **Intersect** where two entities are required (it is not supported in the Pro/ENGINEER User Interface for reference or driven dimensions). Typically, the second ProSelection in this structure should be set to NULL.

- *dsense_arr*—Specifies more information about how the dimension attaches to each attachment point of the model, that is, to what part of the entity.

- *orient_hint*—Specifies the orientation of the dimension and has one of the following values:
 - PRO_DIM_ORNT_HORIZ—Specifies a horizontal dimension.
 - PRO_DIM_ORNT_VERT—Specifies a vertical dimension.
 - PRO_DIM_ORNT_SLANTED—Specifies the shortest distance between two attachment points (available only when the dimension is attached to points).
 - PRO_DIM_ORNT_ELPS_RAD1—Specifies the start radius for a dimension on an ellipse.
 - PRO_DIM_ORNT_ELPS_RAD2—Specifies the end radius for a dimension on an ellipse.
 - PRO_DIM_ORNT_ARC_ANG—Specifies the angle of the arc for a dimension of an arc.
 - PRO_DIM_ORNT_ARC_LENGTH—Specifies the length of the arc for a dimension of an arc.
 - PRO_DIM_ORNT_LIN_TANCRV_ANG—Specifies the value to dimension the angle between the line and the tangent at a curve point (the point on the curve must be an endpoint).
 - PRO_DIM_ORNT_RAD_DIFF—Specifies the linear dimension of the radial distance between two concentric arcs or circles.
- *location*—Specifies the initial location of the dimension text.

The function **ProDimensionAttachmentsGet()** and **ProDimensionAttachmentSet()** provide access to the geometric references and parameters of the driven or reference dimension.

The function **ProDimensionDelete()** deletes the driven or reference dimension. Dimensions stored in annotation elements should be deleted using **ProAnnotationfeatElementDelete()**.

The function **ProDimensionCanRegen()** identifies if a driven dimension can be regenerated.

Accessing Ordinate and Baseline Dimensions

The functions described in this section enable you to create 3D ordinate driven dimensions in 3D models as model annotations or as annotation elements. They also provide the ability to define a baseline annotation element, and then define model ordinate dimension annotations and ordinate dimension annotation elements that reference the baseline annotation element.

Baseline Dimensions

Functions Introduced:

- **ProAnnotationfeatBaselineCreate()**
- **ProDimensionIsBaseline()**

The function **ProAnnotationfeatBaselineCreate()** creates an ordinate baseline annotation element and corresponding dimension. Specify the feature reference geometry, text location, direction and annotation plane as input arguments for this function.

The function **ProDimensionIsBaseline()** identifies whether a dimension is a baseline dimension.

Ordinate Dimensions

Function Introduced:

- **ProDimensionOrdinateCreate()**
- **ProDimensionOrdinatestandardGet()**
- **ProDimensionOrdinatestandardSet()**
- **ProDimensionOrdinatereferencesSet()**
- **ProDimensionMove()**
- **ProDimensionIsOrdinate()**

The function **ProDimensionOrdinateCreate()** creates a new model ordinate driven dimension or a model ordinate reference dimension in a solid model. It requires the input of a reference baseline annotation as well as a geometry reference. The annotation plane for the new dimension will be inherited from the baseline.

To create an ordinate driven dimension element or a model ordinate reference dimension pass the ordinate dimensions created by **ProDimensionOrdinateCreate()** to the function **ProAnnotationelemAnnotationSet()**.

The function **ProDimensionOrdinatestandardGet()** returns the display standard for the ordinate dimensions in the drawing. The style of the ordinate dimension may be as follows:

- **PRO_DIM_ORDSTD_DEFAULT**—Specifies the default style for the ordinate dimensions.
- **PRO_DIM_ORDSTD_ANSI**—Specifies the American National Standard style for the ordinate dimension. It places the related ordinate dimensions without a connecting line.
- **PRO_DIM_ORDSTD_JIS**—Specifies the Japanese Industrial Standard style for the ordinate dimension. It places the ordinate dimensions along a connecting line that is perpendicular to the baseline and starts with an open circle.
- **PRO_DIM_ORDSTD_ISO**—Specifies the International Standard of Organization style for the ordinate dimension.
- **PRO_DIM_ORDSTD_DIN**—Specifies the German Institute for Standardization style for the ordinate dimension.
- **PRO_DIM_ORDSTD_SAME_AS_3D**—Specifies the ordinate dimension style for 2D drawings. Not used in 3D ordinate dimensions.

The function **ProDimensionOrdinatestandardSet()** sets the style for the specified ordinate dimension or a set of ordinate dimensions.

The function **ProDimensionAttachmentsGet()** returns the attachment geometry for the dimension.

In order to change the dimension's attachments you must use the function **ProDimensionOrdinatereferencesSet()**.

The function **ProDimensionMove()** enables you to move the specified 3D ordinate dimension to the specified location within its owner model. For ordinate dimensions in the JIS or ISO/DIN style, all dimensions stay aligned during movement. For ordinate dimensions in the ANSI style, each dimension can be adjusted independent of the other dimensions. If the style is changed back to JIS or ISO/DIN, all the dimensions become aligned with the baseline.

The function **ProDimensionIsOrdinate()** identifies if a dimension is ordinate.

Notes

The functions in this section enable you to access the notes created in Pro/ENGINEER .

Note: These functions are applicable to solids (parts and assemblies) only. However, when notes on a solid are viewed from Drawing mode, they can also be accessed using the **ProDtlnote()** functions described in the chapter Drawings.

A note is modeled in Pro/TOOLKIT as an instance of *ProModelitem* with the type PRO_NOTE. You can select a note by supplying the selection option “note_3d” to **ProSelect()**. You can access the name of a note using the functions **ProModelitemNameGet()** and **ProModelitemNameSet()**.

Creating and Deleting Notes

Functions Introduced:

- **ProSolidNoteCreate()**
- **ProNoteDelete()**

The function **ProSolidNoteCreate()** takes as input a *ProMdl* for the solid (either a part or an assembly), a *ProModelitem* for the owner of the note, and an expandable array *ProLineList* for the note text. The function outputs a *ProNote* object for the created note.

The function **ProNoteDelete()** deletes the note specified by its *ProNote* object. Notes stored in annotation elements should be deleted using **ProAnnotationfeatElementDelete()**.

Note Properties

Functions Introduced:

- **ProNoteTextGet()**
- **ProNoteTextSet()**
- **ProNoteURLGet()**
- **ProNoteURLSet()**
- **ProNoteURLWstringGet()**
- **ProNoteURLWstringSet()**
- **ProNoteOwnerGet()**

- **ProNoteLeaderstyleGet()**
- **ProNoteLeaderstyleSet()**
- **ProNoteElbowlengthGet()**
- **ProNoteElbowlengthSet()**
- **ProNoteLineEnvelopeGet()**
- **ProNoteAttachNormtanleaderGet()**

The function **ProNoteTextGet()** returns the text of a 3D model note. The function **ProNoteTextSet()** modifies the text of an existing 3D model note.

Use the function **ProNoteUndisplay()** followed by **ProNoteDisplay()** to update the display status of the note.

The function **ProNoteURLGet()** retrieves the Uniform Resource Locator (URL) associated with the specified note, whereas **ProNoteURLSet()** sets the associated URL for the specified note.

Note: The functions **ProNoteURLGet()** and **ProNoteURLSet()** have been deprecated. Instead, use the functions **ProNoteURLWstringGet()** and **ProNoteURLWstringSet()** that return and set, respectively, the Uniform Resource Locator (URL) associated with the specified note as a wistring.

The function **ProNoteOwnerGet()** retrieves the owner of the specified note. The owner can be the solid model, a user-chosen feature, or the feature that contains the note's annotation element.

The function **ProNoteLeaderstyleGet()** returns the leader style used for the note. It can be either standard or ISO. The function **ProNoteLeaderstyleSet()** sets the leader style of the note.

The function **ProNoteElbowlengthGet()** returns the elbow properties for the specified note.

The function **ProNoteElbowlengthSet()** sets the elbow properties of the note. This is equivalent to Move Text command in Pro/ENGINEER.

The function **ProNoteLineEnvelopeGet()** returns the envelope of a line for a specified note.

The function **ProNoteAttachNormtanleaderGet()** returns the properties of a leadered note which is normal/tangent to specified note.

Visiting Notes

Function introduced:

- **ProMdlNoteVisit()**

The function **ProMdlNoteVisit()** enables you to visit all the notes in the specified solid model.

Note Text Styles

Functions Introduced:

- **ProTextStyleAlloc()**
- **ProTextStyleFree()**
- **ProNoteTextStyleGet()**
- **ProNoteTextStyleSet()**

The function **ProTextStyleAlloc()** allocates the opaque handle for a *ProTextStyle* data structure. The function **ProTextStyleFree()** frees the allocated data structure.

The function **ProNoteTextStyleGet()** enables you to retrieve the text style of a specified note, whereas **ProNoteTextStyleSet()** enables you to set the text style of the note.

Text Style Properties

Functions introduced

- **ProTextStyleHeightGet()**
- **ProTextStyleHeightSet()**
- **ProTextStyleWidthGet()**
- **ProTextStyleWidthSet()**
- **ProTextStyleAngleGet()**
- **ProTextStyleAngleSet()**
- **ProTextStyleSlantAngleGet()**
- **ProTextStyleSlantAngleSet()**
- **ProTextStyleThicknessGet()**
- **ProTextStyleThicknessSet()**
- **ProTextStyleUnderlineGet()**
- **ProTextStyleUnderlineSet()**

- **ProTextStyleMirrorGet()**
- **ProTextStyleMirrorSet()**
- **ProTextStyleJustificationGet()**
- **ProTextStyleJustificationSet()**
- **ProTextStyleColorGet()**
- **ProTextStyleColorSet()**
- **ProTextStyleIsHeightInModelUnits()**
- **ProTextStyleHeightInModelUnitsSet()**
- **ProSolidDefaulttextheightGet()**

These functions enable you to retrieve and set the properties of the specified text style. You can retrieve and set text properties such as the height, width factor, angle, slant angle, thickness, underline, mirror.

Note: The system uses the value -1.0 for properties that use default values, or that have not been set yet.

The function **ProTextStyleJustificationGet()** returns the horizontal justification for the text style object.

The function **ProTextStyleJustificationSet()** sets the horizontal justification for the text style object.

The function **ProTextStyleColorGet()** returns the color for the text style object.

The function **ProTextStyleColorSet()** sets the color for the text style object.

The functions **ProTextStyleIsHeightInModelUnits()** and **ProTextStyleHeightInModelUnitsSet()** obtain and assign whether the text height is in relation to the model units, or a fraction of the screen size. These functions are applicable only for flat-to-screen annotations.

The function **ProSolidDefaulttextheightGet()** returns the default text height for annotations and dimensions for a given solid model.

Accessing the Note Placement

The functions described in this section provide access to the properties of a 3D note.

Functions Introduced:

- **ProNotePlacementGet()**
- **ProNoteAttachFreeGet()**
- **ProSolidDispoutlineGet()**
- **ProNoteAttachLeadersGet()**
- **ProNoteAttachOnitemGet()**
- **ProNoteAttachPlaneGet()**
- **ProNoteAttachRelease()**

The function **ProNotePlacementGet()** retrieves the a ProNoteAttach structure for the given note. The ProNoteAttach object is an opaque handle that describes the location of a note and the leaders attached to it. The functions in this section enable you to set up a ProNoteAttach object and assign it to a note, and to read the ProNoteAttach information on a note.

The function **ProNoteAttachFreeGet()** retrieves the location of the note text. The note text is stored in relative model coordinates, where {0.5, 0.5, 0.5} indicates the exact center of the model's display bounding box obtained from **ProSolidDispoutlineGet()**, and {0.0, 0.0, 0.0} and {1.0, 1.0, 1.0} represent the corners of the box.

The function **ProSolidDispoutlineGet()** provides you with the maximum and minimum values of X, Y, and Z occupied by the display outline of the solid, with respect to the default provided coordinate system.

The function **ProNoteAttachLeadersGet()** returns the attachment points and properties for the leaders stored in the specified note attachment data.

Note: The function **ProNoteAttachLeadersGet()** requires the owner of the note to be displayed.

The functions **ProNoteAttachOnitemGet()** provides the location of an "On Item" note attachment data.

The functions **ProNoteAttachPlaneGet()** returns the annotation plane assigned to the note attachment data.

The function **ProNoteAttachRelease()** releases the allocated opaque handle.

Modifying 3D Note Attachments

The actual note created in Pro/ENGINEER will not be modified by the access functions until the note attachment is assigned to the note using the modification function **ProNotePlacementSet()**.

Functions Introduced:

- **ProNoteAttachAlloc()**
- **ProNoteAttachFreeSet()**
- **ProNoteAttachAddend()**
- **ProNoteAttachLeaderAdd()**
- **ProNoteAttachLeaderRemove()**
- **ProNoteAttachOnitemSet()**
- **ProNoteAttachPlaneSet()**
- **ProNotePlacementSet()**

The function **ProNoteAttachAlloc()** allocates a ProNoteAttach object for a note attachment.

To set the location of an attachment point, call the function **ProNoteAttachFreeSet()**. See the description of **ProNoteAttachFreeGet()** for an explanation of the coordinates used by this function.

The function **ProNoteAttachAddend()** adds a leader to the specified attachment. The leader points to a location on the parent model specified by an argument of type *ProSelection*.

The attachment types are specified in *ProNoteAttachAttr*. The possible values are as follows:

- PRO_NOTE_ATT_NONE
- PRO_NOTE_ATT_NORMAL
- PRO_NOTE_ATT_TANGENT

Use the function **ProNoteAttachLeaderAdd()** to specify the type of arrowhead for the leader. This function supersedes the function **ProNoteAttachAddend()**.

The function **ProNoteAttachLeaderAdd()** adds a leader to the specified attachment. The leader points to a location on the parent model specified by an argument of type ProSelection. The selection UV parameters determine the precise attachment point for the note leader. The attachment types are specified by the parameter *attr* and can have one of the following values:

- PRO_NOTE_ATT_NORMAL—Specifies a normal attachment.
- PRO_NOTE_ATT_TANGENT—Specifies a tangent attachment.

The function **ProNoteAttachLeaderRemove()** removes a leader from the note attachment data.

The function **ProNoteAttachOnitemSet()** sets the location of an "On Item" note placement. Using this function removes any leaders currently assigned to the note attachment.

The function **ProNoteAttachPlaneSet()** to the annotation plane of the notes. The annotation plane of a note in an annotation element may not be removed.

The function **ProNotePlacementSet()** assigns the a ProNoteAttach structure for the given note, thus defining or redefining the placement for the note.

Note: If modifying an existing note, the functions **ProNoteAttach*Set()** do not modify the note until the note attachment is reassigned to the note using the modification function **ProNotePlacementSet()**.

Geometric Tolerances

For more information on Geometric Tolerances refer to the chapter Annotations: Geometric Tolerances.

Accessing Set Datum Tags

The functions described in this section provide the ability to access and display set datum tag annotations in 3D models.

Functions Introduced:

- **ProSolidSetdatumtagVisit()**
- **ProSetdatumtagCreate()**
- **ProSetdatumtagReferenceGet()**
- **ProGeomitemSetdatumtagGet()**
- **ProSetdatumtagAttachmentGet()**
- **ProSetdatumtagAttachmentSet()**
- **ProSetdatumtagPlaneGet()**
- **ProSetdatumtagPlaneSet()**
- **ProSetdatumtagTextstyleGet()**
- **ProSetdatumtagTextstyleSet()**

The function **ProSolidSetdatumtagVisit()** enables you to visit the set datum tag annotations in a solid model.

The function **ProSetdatumtagCreate()** creates a new set datum tag annotation. The input arguments are as follows:

- *reference*—Specify a datum reference (plane or axis) from the model as the geometric reference for the set datum tag. If you want to make this tag reference model geometry, use Pro/TOOLKIT to create a datum plane or axis referencing the geometry first, and then use that datum in this argument.
- *annotation_plane*—Specify an Annotation Plane for the annotation.
- *attachment*—Optionally, specify the location for placement of the set datum tag. The argument can contain:
 - A dimension
 - A gtol.
 - A geometry selection to which the set datum will be attached.

The function **ProGeomitemSetdatumtagGet()** returns the set datum tag that uses the specified geometric item as the reference, if available.

The function **ProSetdatumtagAttachmentGet()** returns the item on which the datum tag is placed.

The function **ProSetdatumtagAttachmentSet()** specifies the item on which the datum tag is placed.

The function **ProSetdatumtagPlaneGet()** returns the annotation plane for the specified set datum tag.

The function **ProSetdatumtagPlaneSet()** sets the annotation plane for the set datum tag.

The function **ProSetdatumtagReferenceGet()** returns the datum specified as a reference for the set datum tag.

The function **ProSetdatumtagTextstyleGet()** returns the text style details for the set datum tag.

The function **ProSetdatumtagTextstyleSet()** sets the text style details for the set datum tag.

Accessing Set Datums for Datum Axes or Planes

The functions described in this section provide access to the “Set Datum” status of a datum axis or plane.

Note: These function support the "Set Datum" capability which existed before Set Datum Tag annotations.

Functions Introduced:

- **ProGeomitemSetdatumGet()**
- **ProGeomitemSetdatumSet()**
- **ProGeomitemSetdatumClear()**

The function **ProGeomitemSetdatumGet()** specifies whether the datum plane or axis is a “Set Datum”. This function supersedes the function **ProGeomitemIsGtolref()**.

The function **ProGeomitemSetdatumSet()** sets the datum plane or axis to be a “Set Datum”. This function supersedes the function **ProGeomitemGtolrefSet()**.

The function **ProGeomitemSetdatumClear()** removes the “Set Datum” status of a datum plane or axis. This function supersedes the function **ProGeomitemGtolrefClear()**.

Surface Finish Annotations

The functions described in this section provide read access to the properties of the surface finish object. They also allow you to create and modify surface finishes.

The style of surface finishes for releases previous to Pro/ENGINEER Wildfire 2.0 was a flat-to-screen symbol attached to a single surface. From Pro/ENGINEER Wildfire 2.0 onwards, the method for construction of surface finishes has been modified. The new style of surface finish is a symbol instance that may be attached on a surface or with a leader. The following functions support both the old and new surface finish annotations, except where specified.

Functions Introduced:

- **ProSolidSurffinishVisit()**
- **ProSurffinishCreate()**
- **ProSurffinishReferencesGet()**
- **ProSurffinishSrfcollectionGet()**
- **ProSurffinishSrfcollectionSet()**
- **ProSurffinishNameGet()**
- **ProSurffinishNameSet()**
- **ProSurffinishDataGet()**
- **ProSurffinishModify()**
- **ProSurffinishValueGet()**
- **ProSurffinishValueSet()**
- **ProSurffinishDelete()**

The function **ProSolidSurffinishVisit()** visits the surface finishes stored in the specified solid model. This function accepts a visit function **ProSurfaceVisitAction()** and a filter function **ProSurfaceFilterAction()** as the input arguments.

The function **ProSurffinishCreate()** creates a new symbol-based surface finish annotation. The function requires a symbol instance data structure for creation.

Note: The data must conform to the requirements for surface finishes, that is, attachment must be via leader to or on one or more surfaces.

You can use the standard surface finish symbol definitions from the symbol instance data structure. Use the function **ProSolidDtlSymdefRetrieve()** to retrieve the surface finish symbol definitions from the location PRO_DTLSYMDEF_SRC_SURF_FINISH_DIR. The surface finish value should be set using the variant text options for symbol instances.

The function **ProSurffinishReferencesGet()** returns the surface or surfaces referenced by the surface finish.

The function **ProSurffinishSrfcollectionGet()** obtains a surface collection which contains the references of the surface finish.

The function **ProSurffinishSrfcollectionSet()** assigns a surface collection to be the references of the surface finish. This overwrites all current surface finish references. The following types of surface collections are supported:

- One by one surface set
- Intent surface set
- Excluded surface set
- Seed and Boundary surface set
- Loop surface set
- Solid surface set
- Quilt surface set

Note: Only those surface finishes that are contained within annotation elements may use a collection of references instead of a single surface reference.

The function **ProSurffinishNameGet()** returns the name of the surface finish annotation.

The function **ProSurffinishNameSet()** sets the name of the surface finish annotation.

The function **ProSurffinishDataGet()** returns the symbol instance data for the surface finish. This function supports only new symbol-based surface finishes.

The function **ProSurffinishModify()** modifies the symbol instance data for the specified surface finish. This function supports only new symbol-based surface finishes.

The function **ProSurffinishValueGet()** retrieves the value of a surface finish annotation.

The function **ProSurffinishValueSet()** sets the value of a surface finish annotation.

The function **ProSurffinishDelete()** deletes the specified surface finish.

The functions described above supersede the Pro/Develop functions **pro_surf_finish_number()**, **pro_get_surf_finish()**, **pro_delete_surface_finish()** and **pro_set_surface_finish()**.

Symbol Annotations

The functions described in this section provide support for 3D mode symbols. Pro/TOOLKIT functions for symbol instances are used in both 2D and 3D modes. Symbols for a particular mode must conform to the requirements for that mode. Some **ProDtlsyminst** functions have been modified to accept a ProMdl object as the input instead of ProDrawing object to support 3D mode operations.

Creating, Reading and Modifying 3D Symbols

Functions Introduced:

- **ProDtlsyminstCreate()**
- **ProDtlsyminstDataGet()**
- **ProDtlsyminstModify()**
- **ProDtlsyminstdataPlaneGet()**
- **ProDtlsyminstdataPlaneSet()**

The function **ProDtlsyminstCreate()** creates a symbol instance in the specified model.

The function **ProDtlsyminstDataGet()** returns the symbol instance data.

The function **ProDtlsyminstModify()** modifies the symbol instance.

For more information about creating, accessing and modifying symbols via the ProDtlsyminst and ProDtlsyminstdata structures, refer to the Drawings chapter.

The functions **ProDtlsyminstdataPlaneGet()** and **ProDtlsyminstdataPlaneSet()** provide access to the symbol annotation plane. Annotation planes are required for 3D symbol instances but are not applicable for 2D symbol instances.

Locating 3D Symbols and Symbol Definitions

Functions Introduced:

- **ProSolidDtlSymdefVisit()**
- **ProSolidDtlSyminstVisit()**
- **ProSolidDtlSymdefEntityVisit()**
- **ProSolidDtlSymdefNoteVisit()**
- **ProSolidDtlSymdefRetrieve()**

The functions **ProSolidDtlSymdefVisit()** and **ProSolidDtlSyminstVisit()** allow traversal of symbol definitions and instances in a solid model.

The functions **ProSolidDtlSymdefEntityVisit()** and **ProSolidDtlSymdefNoteVisit()** allow traversal of items contained in a symbol definition stored in a solid model.

The function **ProSolidDtlSymdefRetrieve()** allows retrieval of a symbol definition into a given solid model. The input arguments for this function are as follows:

- *solid*—Specifies a handle to the solid model.
- *location*—Specifies the location of the symbol definition file. It can be one of the following values:
 - **PRO_DTLSYMDEF_SRC_SYSTEM**—Specifies the system symbol definition directory.
 - **PRO_DTLSTMDEF_SRC_SYMBOL_DIR**—Specifies the system surface finish symbol definition directory
 - **PRO_DTLSYMDEF_SRC_SYMBOL_DIR**—Specifies the location controlled by the configuration option `pro_symbol_dir`.
 - **PRO_DTLSYMDEF_SRC_PATH**—Specifies the absolute path to a directory containing the symbol definition.
- *filepath*—Specifies the path to the file with a symbol definition. The path is relative to the location specified in the argument *location*.
- *filename*—Specifies the name of the symbol definition file.
- *version*—Specifies the version of the symbol definition file.

- *update*—Specifies the update flag. If TRUE, the definition will be loaded even if a definition of that name already exists in the model. If FALSE, the retrieval will not take place if the definition exists in the model.

Annotations: Geometric Tolerances

The functions in this chapter allow a Pro/TOOLKIT application to read, modify, and create geometric tolerances (gtols) in a solid or drawing. We recommend that you study the Pro/ENGINEER documentation on geometric tolerances, and develop experience with manipulating geometric tolerances using the Pro/ENGINEER commands before attempting to use these functions.

Topic	Page
Geometric Tolerance Objects	23 - 2
Visiting Geometric Tolerances	23 - 2
Reading Geometric Tolerances	23 - 3
Creating a Geometric Tolerance	23 - 7
Editing a Geometric Tolerance	23 - 15
Deleting a Geometric Tolerance	23 - 16
Geometric Tolerance Layout	23 - 16
Additional Text for Geometric Tolerances	23 - 17
Geometric Tolerance Text Style	23 - 18
Prefix and Suffix for Geometric Tolerances	23 - 18
Parameters for Geometric Tolerance Attributes	23 - 19

Geometric Tolerance Objects

ProGtol

Geometric tolerances in a Pro/ENGINEER model are referenced by the data handle, *ProGtol*. This handle is identical to *ProModelitem*, in which the type field is set to PRO_GTOL. You can use **ProSelect()** with the option “gtol” to select a gtol, after which you can extract the *ProGtol* handle using **ProSelectionModelitemGet()**.

ProGtoldata

This is an opaque handle object that references an internal data structure containing a complete description of a gtol, its properties, and its state. In the initial stages of creation of the gtol, the data in the *ProGtoldata* handle is not associated with the Pro/ENGINEER model. However, while editing an existing gtol, the *ProGtoldata* object refers directly to the gtol description in the Pro/ENGINEER model. Creating and editing gtols are discussed in more detail below.

Visiting Geometric Tolerances

Function Introduced:

- **ProMdlGtolVisit()**

The function **ProMdlGtolVisit()** visits geometric tolerances stored in a part, assembly, or drawing. The forms of the visit and filter functions are similar to those of most other visit functions—they receive a *ProGtol* pointer as input argument to identify the gtol.

Reading Geometric Tolerances

Function Introduced:

- **ProGtolDataGet()**
- **ProGtoldataTypeGet()**
- **ProGtoldataModelGet()**
- **ProGtoldataOwnerGet()**
- **ProGtoldataReferenceGet()**
- **ProGtoldataPlacementGet()**
- **ProGtoldataPlaneGet()**
- **ProGtolleaderGet()**
- **ProGtolleadersFree()**
- **ProGtoldataGtoldatumrefGet()**
- **ProGtoldatumrefGet()**
- **ProGtoldataCompositeGet()**
- **ProGtoldataValueGet()**
- **ProGtoldataPerUnitGet()**
- **ProGtoldataUnitareaStyleGet()**
- **ProGtoldataMatCondGet()**
- **ProGtoldataStatTolGet()**
- **ProGtoldataDiameterGet()**
- **ProGtoldataFreeStateGet()**
- **ProGtoldataAllAroundGet()**
- **ProGtoldataTangentPlaneGet()**
- **ProGtoldataProjZoneGet()**
- **ProGtoldataProfBoundaryGet()**
- **ProGtoldataASMEY1441Get()**
- **ProGtoldataDispositionValGet()**

Annotations: Geometric
Tolerances

The function **ProGtolDataGet()** provides a *ProGtoldata* object which describes the contents of a specified *ProGtol* object in a Pro/ENGINEER model. You can unpack the *ProGtoldata* object by making calls to the other **ProGtol*Get()** functions listed above. They are described briefly below, but their meaning should be easy to understand if they are compared with the contents of the Geometric Tolerance dialog box in Pro/ENGINEER.

ProGtoldataTypeGet() provides the type of the gtol (straightness, flatness, and so on).

ProGtoldataModelGet() returns the model that contains the gtol. This is the same as the *Owner* field in the *ProGtol* object, and can be a part, assembly, or drawing.

The function **ProGtoldataOwnerGet()** returns the model that defines the origin of *ProSelection* structures used to define references inside the gtol. This will usually be the model that contains the gtol (as shown by **ProGtoldataModelGet()**); but if the gtol was created in drawing mode and added to a solid in a drawing view, the owner will be the drawing, while the model is the solid.

ProGtoldataReferenceGet() returns the geometric entity referenced by the gtol and a *ProSelection* object that identifies the reference entity.

ProGtoldataPlacementGet() provides the placement type and the items referenced by the placement. The following types of placements are available:

- PROGTOLPTYPE_DATUM – the gtol is placed on its reference datum.
- PROGTOLPTYPE_DIMENSION – the gtol is attached to a dimension. The ProDimension object will be included in the output.
- PROGTOLPTYPE_DIM_RELATED – the gtol is placed relative to a dimension, but not directly attached.
- PROGTOLPTYPE_FREENOTE – the gtol is placed as a free, unattached note in the model or drawing.
- PROGTOLPTYPE_LEADERS – the gtol is attached to geometry via one or more leaders. The leaders are represented using an opaque handle, *ProGtolleader*.
- PROGTOLPTYPE_TANLEADER – the gtol is attached to geometry via a tangent leader. The leader will be represented using a *ProGtolleader* handle.

- **PROGTOLPTYPE_NORMLEADER** – the gtol is attached to geometry via a normal leader. The leader will be represented using a *ProGtolleader* handle.
- **PROGTOLPTYPE_GTOL** – the gtol is attached to another gtol. The *ProGtol* object will be included in the output.

To unpack the information in the *ProGtolleader* handle, use the function **ProGtolleaderGet()**. After reading the leaders, free the leader array by calling **ProGtolleadersFree()**.

The function **ProGtoldataPlaneGet()** provides access to the annotation plane assigned to the geometric tolerance. Annotation planes are required for gtols in solid models when the placement type is **PROGTOLPTYPE_FREENOTE**, **PROGTOLPTYPE_LEADERS**, **PROGTOLPTYPE_TANLEADER**, and **PROGTOLPTYPE_NORMLEADER**. Refer to the section **Annotation Orientation** for more information on Annotation Planes.

The functions **ProGtoldataGtoldatumrefGet()**, **ProGtoldatumrefGet()**, and **ProGtoldataCompositeGet()** provide information on the gtol datum references.

ProGtoldataGtoldatumrefGet() returns the primary, secondary, and tertiary datum references for a gtol. Use the function **ProGtoldatumrefGet()** to unpack this object and get the basic and compound datums and material conditions.

ProGtoldataCompositeGet() reports whether the gtol is a composite one, and, if so, the composite value and reference type.

The functions **ProGtoldataValueGet()**, **ProGtoldataPerUnitGet()**, **ProGtoldataUnitareaStyleGet()**, and **ProGtoldataMatCondGet()** provide information on the gtol value and material condition symbol.

The function **ProGtoldataUnitareaStyleGet()** gets the unit area style for a gtol which can be of the following types:

Unit Area Style	Gtol display format
PROGTOLUNITAREASTYLE_X (default type)	

Unit Area Style	Gtol display format
PROGTOLUNITAREASTYLE_BOX	
PROGTOLUNITAREASTYLE_DIA	

This function enables the user to display the unit area of a geometric tolerance with an image preceding the unit area value. This display is in accordance with JIS B0602-1964 standard.

Note: The unit area style for a gtol is applicable only for PROGTOLTYPE_FLATNESS.

The functions **ProGtoldataStatTolGet()**, **ProGtoldataDiameterGet()**, **ProGtoldataFreeStateGet()**, **ProGtoldataAllAroundGet()**, **ProGtoldataTangentPlaneGet()**, **ProGtoldataProjZoneGet()**, and **ProGtoldataProfBoundaryGet()** provide information on the different symbolic modifiers available to the gtol.

The function **ProGtoldataASMEY1441Get()** specifies if surface profile gtols are in accordance with the ASME Y 14.41 standards.

The function **ProGtoldataDispositionValGet()** gets the value of the unequal disposition for profile gtols. An unequal disposition implies that the tolerance specified in the gtol is applied unequally about the selected profile.

Creating a Geometric Tolerance

Functions Introduced:

- **ProGtoldataAlloc()**
- **ProGtoldataTypeSet()**
- **ProGtoldataModelSet()**
- **ProGtoldataReferenceSet()**
- **ProGtolleaderAlloc()**
- **ProGtoldataPlacementSet()**
- **ProGtoldataPlaneSet()**
- **ProGtolleaderFree()**
- **ProGtoldatumrefAlloc()**
- **ProGtoldataGtoldatumrefSet()**
- **ProGtoldatumrefFree()**
- **ProGtoldataCompositeSet()**
- **ProGtoldataValueSet()**
- **ProGtoldataPerUnitSet()**
- **ProGtoldataUnitareaStyleSet()**
- **ProGtoldataMatCondSet()**
- **ProGtoldataStatTolSet()**
- **ProGtoldataDiameterSet()**
- **ProGtoldataFreeStateSet()**
- **ProGtoldataAllAroundSet()**
- **ProGtoldataTangentPlaneSet()**
- **ProGtoldataProjZoneSet()**
- **ProGtoldataProfBoundarySet()**
- **ProGeomitemGtolrefSet()**
- **ProGtolCreate()**

The basic steps in creating a gtol are:

1. Allocate a *ProGtoldata* structure using **ProGtoldataAlloc()**.
2. Set tolerance properties using the **ProGtoldata*Set()** functions.

3. Create the tolerance using **ProGtolCreate()**.
4. Free the *ProGtoldata* using **ProGtoldataFree()**.

Be careful about the sequence in which applications call the **ProGtol*Set()** functions. Each function tries to maintain the internal consistency of the gtol description, and may:

- Not set the value if it conflicts with other existing settings, or requires other settings that have not yet been made
- Modify (usually clear) any existing settings that conflict with the setting being made

As an example of the first case, if you set the gtol placement using **ProGtoldataPlacementSet()** before setting the reference with **ProGtoldataReferenceSet()**, then the former function will fail with the error status PRO_TK_CANT_MODIFY because it cannot verify the placement in relation to the reference.

As an example of the second case, if you modify the gtol reference using **ProGtoldataReferenceSet()** in a gtol that already has a placement set, the placement becomes unavailable because it depends upon the reference.

The function **ProGtoldataPlaneSet()** sets the annotation plane for the given gtol. Any gtol that is placed with placement types PROGTOLPTYPE_FREENOTE, PROGTOLPTYPE_LEADERS, PROGTOLPTYPE_TANLEADER, and PROGTOLPTYPE_NORMLEADER must be placed on an annotation plane before the call to the function **ProGtolCreate()**. Refer to the section Annotation Orientation for more information on Annotation Planes.

The **ProGtol*Set()** functions perform exactly the same operations that result from selecting the corresponding buttons in Pro/ENGINEER's Geometric Tolerance dialog box. To ensure that you call the **ProGtol*Set()** functions in the correct sequence, first experiment with creating the desired gtol interactively in Pro/ENGINEER, and then use a sequence of **ProGtol*Set()** functions that reflects the Pro/ENGINEER button sequence.

Call the functions in the order listed at the head of this section; this is the same as progressing from left to right through the options on each tab, and left to right through the tabs, on the Pro/ENGINEER Geometrical Tolerance dialog box.

A useful feature of the **ProGtol*Set()** functions is that each has an output argument of type **ProGtoldataStatus**, an enumerated type whose value indicates whether the gtol is fully defined, and if not, what other minimum setting is needed next. In the example of datum creation below, the value of the tolerance status from some of the function calls is as follows:

Function	GtoldataStatus
ProGtoldataReferenceSet()	PROGTOLSTATUS_PLACEMENT
ProGtoldataPlacementSet()	PROGTOLSTATUS_REF_DTM_NEEDED
ProGtoldataGtoldatumrefSet()	PROGTOLSTATUS_COMPLETE

If the function **ProGtoldataPlacementSet()** specifies one or more leaders, the leaders are described by a separate opaque object called *ProGtolleader*. This object is allocated and freed by calls to **ProGtolleaderAlloc()** and **ProGtolleaderFree()**. Similarly, the datum references are given to **ProGtoldataGtoldatumrefSet()** in the form of objects of type *ProGtoldatumref*, which you allocate and free with **ProGtoldatumrefAlloc()** and **ProGtoldatumrefFree()**.

The function **ProGtoldataUnitareaStyleSet()** sets the unit area style for a gtol.

Note:

- When modifying the values of an existing flatness gtol, the display of the unit area is preserved. You can modify the area and tolerance values as long as the gtol type is flatness.
- Changing the type of the gtol (from flatness to any other type of gtol) will cause the style format of the gtol to be lost. The style will not be restored if you change the gtol back to a flatness type gtol.

Example 1: Attaching a Position GTOL to Solid Planes Parallel to a Datum Plane

This example shows how the function **UsrPlanesTol()** implements a command to attach a position gtol to each solid plane surface that is parallel to a selected reference datum. If the selected datum is not a gtol reference, you can choose to make it a reference.

```

#include <ProToolkit.h>
#include <ProSurface.h>
#include <ProArray.h>
#include <ProGtol.h>
#include <ProUtil.h>
#include <ProModelitem.h>
#include <ProMessage.h>
#include <ProAnnotation.h>
#include <ProSolid.h>
#include <ProDisplist.h>
#include <ProWindows.h>

#include <UtilMessage.h>
#include <PTApplsUnicodeUtils.h>
#define EPSM6 0.0001
/*-----*\
Data structure for finding parallel solid plane surfaces
\*-----*/
typedef struct planes_data
{
    ProGeomitem reference;
    ProAnnotationPlane ap;
    ProVector normal;
    double tolerance;
} Planesdata_t;
/*=====*\
FUNCTION: UsrPlanePositiontolSet()
PURPOSE: To add a position gtol to the specified surface
\*=====*/
int UsrPlanePositiontolSet(
    ProSelection surface, /* The surface */
    ProVector pos, /* The position of the gtol */
    ProGeomitem *reference, /* The datum reference */
    ProAnnotationPlane* ap, /* The annotation plane */
    double tolerance) /* The tolerance value */
{
    ProError status;
    ProGtoldata gdata;
    ProGtoldataStatus gstatus;
    ProGtolleader leader, *leaders;
    ProName wname;
    ProCharName name;
    ProModelitem modelitem;
    ProGtoldatumref datumref;
    ProSelection ref;
    ProGtol gtol;
/*-----*\
Allocate the gtol data structure
\*-----*/
    ProGtoldataAlloc(reference->owner, &gdata);
/*-----*\

```



```

Set the gtol type
\*-----*/
    ProGtoldataTypeSet(gdata, PROGTOLTYPE_POSITION, &gstatus);
/*-----*\
Set the gtol model
\*-----*/
    ProGtoldataModelSet(gdata, reference->owner, &gstatus);
/*-----*\
Set the reference to the surface
\*-----*/
    ProGtoldataReferenceSet(gdata, PROGTOLRTYPE_SURF, surface, &gstatus);
/*-----*\
Allocate a leader which is attached to the surface
\*-----*/
    ProGtolleaderAlloc(PROLEADERTYPE_ARROWHEAD, surface, &leader);
/*-----*\
Set up an array of leaders with the one leader in it
\*-----*/
    ProArrayAlloc(0, sizeof(ProGtolleader), 1, (ProArray*)&leaders);
    ProArrayObjectAdd((ProArray*)&leaders, -1, 1, &leader);
/*-----*\
Set the placement using the leader and the specified position
\*-----*/
    ProGtoldataPlacementSet(gdata, PROGTOLPTYPE_LEADERS, NULL, leaders,
                             pos, NULL, &gstatus);
/*-----*\
Set the annotation plane
\*-----*/
    ProGtoldataPlaneSet(gdata, ap);
/*-----*\
Free the leader
\*-----*/
    ProGtolleaderFree(&leader);
/*-----*\
Set up a ProSelection for the datum, and set it as the basic reference
\*-----*/
    ProSelectionAlloc(NULL, reference, &ref);
    ProGtoldatumrefAlloc(ref, PROGTOLMATCOND_DEFAULT_RFS, NULL,
                          PROGTOLMATCOND_DEFAULT_RFS, &datumref);
    ProGtoldataGtoldatumrefSet(gdata, datumref, NULL, NULL, &gstatus);
    ProGtoldatumrefFree(&datumref);
/*-----*\
Set the tolerance value
\*-----*/
    ProSelectionModelitemGet(surface, &modelitem);
    ProTKSprintf(name, "surf%d", modelitem.id); ProStringToWstring(wname,
                             name);
    ProGtoldataValueSet(gdata, PRO_B_TRUE, tolerance, wname, &gstatus);
/*-----*\
Create the tolerance
\*-----*/

```

```

    status = ProGtolCreate(gdata, &gtol);
/*-----*\
Free the gtol data
\*-----*/
    ProGtoldataFree(&gdata);
    return(status == PRO_TK_NO_ERROR ? 1 : 0);
}
/*=====*\
FUNCTION: UsrSurfAction()
PURPOSE: Action function called when visiting solid surfaces to
         attach gtol to.
\*=====*/
ProError UsrSurfAction(
    ProSurface surface,
    ProError filt_status,
    ProAppData data)
{
    Planesdata_t *pdata=(Planesdata_t*)data;
    ProVector normal, cross, pos;
    ProUvParam uv;
    ProSrftype stype;
    int id;
    ProModelitem modelitem;
    ProSelection sel;
/*-----*\
If the surface is not a plane, skip it.
\*-----*/
    ProSurfaceTypeGet(surface, &stype);
    if(stype != PRO_SRF_PLANE)
        return(PRO_TK_NO_ERROR);
/*-----*\
If the surface is not parallel to the reference datum, skip it.
\*-----*/
    uv[0]=uv[1]=0.0;
    ProSurfaceXYZdataEval(surface, uv, pos, NULL, NULL, normal);
    ProUtilVectorCross(normal, pdata->normal, cross);
    if(fabs(ProUtilVectorLength(cross)) > EPSM6)
        return(PRO_TK_NO_ERROR);
/*-----*\
Set the position of the gtol to be the point for zero UV, offset by the
outward normal.
\*-----*/
    pos[0] += normal[0];
    pos[1] += normal[1];
    pos[2] += normal[2];
/*-----*\
Add the gtol to the surface
\*-----*/
    ProSurfaceIdGet(surface, &id);
    ProModelitemInit(pdata->reference.owner, id, PRO_SURFACE, &modelitem);
    ProSelectionAlloc(NULL, &modelitem, &sel);

```

```

    UsrPlanePositiontolSet(sel, pos, &pdata->reference, &pdata->ap,
                           pdata->tolerance);
    return(PRO_TK_NO_ERROR);
}
/*=====*\
FUNCTION: UsrPlanesTol()
PURPOSE:  Command to add a position gtol to all solid planes that are
          parallel to a selected datum. Makes the selected datum
          into a gtol reference if required.
\*=====*/
int UsrPlanesTol()
{
    ProError status;
    ProSelection *sel;
    int n_sel;
    ProGeomitem datum;
    ProName wname;
    ProBoolean ref_datum, is_in_dim;
    ProDimension dim;
    Planesdata_t data;
    ProUvParam uv;
    ProSurface surface;
    ProFileName msgfil;
    ProAnnotationPlane ap;
    ProVector normal;
    ProModelitem ap_datum;
    ProSurface ap_surf;
    ProGeomitemdata* gdata;

    ProStringToWstring (msgfil, "msg_uggtol.txt");
/*-----*\
Select the datum
\*-----*/
    ProMessageDisplay(msgfil,
                      "USER Select a datum plane for gtol references");
    if(ProSelect("datum",1,NULL,NULL,NULL,NULL,&sel,&n_sel)
        != PRO_TK_NO_ERROR || n_sel < 0)
        return(0);

    ProSelectionModelitemGet(sel[0], &datum);
/*-----*\
Convert it's type to be a DATUM_PLANE
\*-----*/
    ProModelitemInit(datum.owner, datum.id, PRO_DATUM_PLANE, &datum);
    ProModelitemNameGet(&datum, wname);
/*-----*\
Is the datum a gtol reference?
\*-----*/
    ProGeomitemIsGtolref(&datum, &ref_datum, &is_in_dim, &dim);
/*-----*\
If so, say so; if not, ask whether it should be made one

```

```

\*-----*/
    if(ref_datum)
        ProMessageDisplay(msgfil,"USER %0w is already a reference datum",
                           wname);
    else
    {
        ProMessageDisplay(msgfil,"USER %0w is not a reference datum."
                           "Do you wish to set it (yes/no)?|||yes", wname);
        ref_datum = ProUtilYesnoGet("YES");
    }
/*-----*\
    If "no" then exit, else set the datum ad a gtol reference
\*-----*/
    if(!ref_datum)
        return(0);
    ProGeomitemSetdatumSet (&datum, NULL);
}
/*-----*\
    Remember the reference
\*-----*/
    memcpy(&data.reference, &datum, sizeof(ProGeomitem));
/*-----*\
    Calculate the normal, used in checking for parallel surfaces
\*-----*/
    ProSurfaceInit(datum.owner, datum.id, &surface);
    uv[0]=uv[1]=0.0;
    ProSurfaceXYZdataEval(surface, uv, NULL, NULL, NULL, data.normal);
/*-----*\
    Ask the user for the annotation plane
\*-----*/
    ProMessageDisplay(msgfil,"USER Select the annotation plane to use");
    if (ProSelect ("datum", 1, NULL, NULL, NULL, NULL, &sel, &n_sel))
        return (0);
/*-----*\
    Create the annotation plane using the default normal vector
\*-----*/
    status = ProSelectionModelitemGet (sel[0], &ap_datum);
    status = ProGeomitemToSurface (&ap_datum, &ap_surf);
    status = ProSurfaceDataGet (ap_surf, &gdata);
    memcpy (normal, gdata->data.p_surface_data->srf_shape.plane.e3, sizeof
            (ProVector));
    ProGeomitemdataFree (&gdata);
    status = ProAnnotationplaneCreate (sel[0], normal, &ap);
    memcpy(&data.ap, &ap, sizeof(ProAnnotationPlane));
/*-----*\
    Ask the user for the tolerance value
\*-----*/
    ProMessageDisplay(msgfil,"USER Enter the tolerance value|||0.1");
    status = ProMessageDoubleRead(NULL, &data.tolerance);
    if(status == PRO_TK_MSG_USER_QUIT)
        return(0);
    if(status != PRO_TK_NO_ERROR)

```

```

    data.tolerance = 0.1;
/*-----*\
Visit all the solid surfaces, attaching a gtol to each one which is
plane and parallel to the reference.
/*-----*/
    ProSolidSurfaceVisit(datum.owner, UsrSurfAction, NULL, &data);

/*-----*\
Repaint the model to show the added geometric tolerances
/*-----*/
    ProDisplistInvalidate (datum.owner);
    ProWindowRepaint (PRO_VALUE_UNUSED);
    return(1);
}

```

Editing a Geometric Tolerance

Functions Introduced:

- **ProGtolEditInit()**
- **ProGtolEditCommit()**
- **ProGtolEditCancel()**
- **ProGtoldataFree()**
- **ProGtolValueSet()**
- **ProGtoldataNumdigitsSet()**
- **ProGtoldataNumdigitsGet()**

A call to **ProGtolEditInit()** returns a pointer to the *ProGtoldata* data handle that describes the gtol in the Pro/ENGINEER model to be edited.

Note: After the call to **ProGtolEditInit()**, the data handle returned is not a copy of the data, but is directly tied to Pro/ENGINEER's representation of the gtol. Calls to the **ProGtol*Set()** functions on the data handle returned have an immediate effect on the gtol in Pro/ENGINEER. However, you can cancel the effect of all modifications done using **ProGtol*Set()** functions by calling **ProGtolEditCancel()**. This restores the gtol from a backup made during the call to **ProGtolEditInit()**.

Call **ProGtolEditCommit()** to notify Pro/ENGINEER that you have finished modifying the gtol using the **ProGtol*Set()** functions.

After calling **ProGtolEditCommit()** or **ProGtolEditCancel()**, call **ProGtoldataFree()** to free the workspace *ProGtoldata* structure.

When modifying a gtol, it is important to be careful of **ProGtol*Set()** functions that may affect or depend on other settings. For example, if you modify the reference of a gtol using these functions, the placement will become unavailable. So if you want to switch the reference and the leader attachment of a gtol to a different entity, but otherwise keep the placement the same, you should call the relevant functions in the following sequence:

1. **ProGtoldataReferenceGet()**
2. **ProGtoldataPlacementGet()**
3. **ProGtoldataReferenceSet()**
4. **ProGtoldataPlacementSet()**

This ensures that you do not lose the placement information when **ProGtoldataReferenceSet()** is called.

The function **ProGtolValueSet()** assigns the geometric tolerance value without requiring the use of **ProGtolEditInit()**. It is applicable only for gtols that have an overall value. This method is the only set method that supports modification of gtol values copied into another model via an inheritance feature.

The function **ProGtoldataNumdigitsSet()** assigns the number of significant digits for a specified gtol value.

The function **ProGtoldataNumdigitsGet()** returns the number of significant digits assigned to the specified gtol value.

Deleting a Geometric Tolerance

Function Introduced:

- **ProGtolDelete()**

The function **ProGtolDelete** permanently removes a gtol.

Geometric Tolerance Layout

The functions described in this section provide access to the layout for the text and symbols in a geometric tolerance.

Functions Introduced:

- **ProGtolElbowlengthGet()**
- **ProGtolLineEnvelopeGet()**

The function **ProGtolElbowlengthGet()** returns the length and direction of the geometric tolerance leader elbow.

The function **ProGtolLineEnvelopeGet()** returns the bounding box coordinates for one line from the geometric tolerance.

Additional Text for Geometric Tolerances

You can place multi-line additional text to the right of and above a geometric tolerance control frame while creating and editing a gtol in both drawing and model modes.

Functions Introduced:

- **ProGtolRightTextGet()**
- **ProGtolRightTextSet()**
- **ProGtolTopTextGet()**
- **ProGtolTopTextSet()**

The function **ProGtolRightTextGet()** retrieves the text added to the right of the specified geometric tolerance.

The function **ProGtolRightTextSet()** assigns the text added to the right of the specified geometric tolerance.

The function **ProGtolTopTextGet()** obtains the text added to the top of the specified geometric tolerance.

The function **ProGtolTopTextSet()** assigns the text added to the top of the specified geometric tolerance.

Note:

- If the additional text extends over multiple lines, the output string will contain ‘\n’ characters indicating line breaks.
- The text added to the top of a gtol cannot extend beyond the length of the geometric tolerance control frame.

Geometric Tolerance Text Style

The functions described in this section access the text style properties of a geometric tolerance.

Functions Introduced:

- **ProGtolTextstyleGet()**
- **ProGtolTextstyleSet()**
- **ProGtoltextTextstyleGet()**
- **ProGtoltextTextstyleSet()**

The function **ProGtolTextstyleGet()** returns the text style assigned of a specified geometric tolerance.

The function **ProGtolTextstyleSet()** assigns the text style of a specified geometric tolerance.

The function **ProGtoltextTextstyleGet()** retrieves the text style of the additional text applied to the specified geometric tolerance.

The function **ProGtoltextTextstyleSet()** assigns the text style of the additional text applied to the specified geometric tolerance.

Prefix and Suffix for Geometric Tolerances

You can easily add a prefix and suffix to a geometric tolerance in both drawing and model modes. They have the same text style as the geometric tolerance text.

A prefix will be placed before the tolerance value; a suffix will be placed after the material condition, if one exists.

Functions Introduced:

- **ProGtolPrefixGet()**
- **ProGtolPrefixSet()**
- **ProGtolSuffixGet()**
- **ProGtolSuffixSet()**

The function **ProGtolPrefixGet()** obtains the prefix text for the specified geometric tolerance.

The function **ProGtolPrefixSet()** assigns the prefix set for the specified geometric tolerance.

The function **ProGtolSuffixGet()** obtains the suffix text for the specified geometric tolerance.

The function **ProGtolSuffixSet()** assigns the suffix text for the specified geometric tolerance.

Parameters for Geometric Tolerance Attributes

System parameters are automatically generated for the attributes of a geometric tolerance upon the creation of the geometric tolerance. These parameters are used for downstream processes such as Coordinate Measuring Machine (CMM) operations, and in driving other annotation and feature relationships.

Note: Parameters are generated only for geometric tolerances created within annotation features because 2D or 3D geometric tolerances created outside annotation elements do not have a placeholder for parameters.

The following table lists the various system parameters and the gtol attributes for which the parameters are generated:

Parameter	Gtol Attribute
PTC_GTOL_PRIMAY_TOL	Primary tolerance value
PTC_GTOL_TYPE	Geometric tolerance type
PTC_GTOL_MATERIAL_CONDITION	Material condition for the primary tolerance value
PTC_GTOL_COMPOSITE_TOL	Composite tolerance value; available only if the gtol type is Surface or Position
PTC_GTOL_PERUNIT_TOL	Pre-unit tolerance value; available only if the gtol type is Straightness, Perpendicular, Surface, Parallel, or Flatness
PTC_GTOL_UNITLENGTH_TOL	Unit length value; available only if the gtol type is Straightness, Perpendicular, Surface, or Parallel
PTC_GTOL_UNITAREA_TOL	Unit area value; available only if the gtol type is Flatness

Parameter	Gtol Attribute
PTC_GTOL_PROJECTEDTOLZONE_TOL	Projected tolerance zone value; available only if the gtol type is Angular, Perpendicular, Parallel, or Position
PTC_GTOL_UNEQUALLYDISPOSED_TOL	Unequally disposed tolerance value; available only if the gtol type is Surface

You can access the parameters of gtol attributes by using the Pro/TOOLKIT parameter functions **ProParameter*Get()** and **ProParameter*Set()**, and the UDF placement functions related to variable parameters **ProUdfvarparam*()** and **ProUdfdataVarparamAdd()**. Refer to the chapter Core: Parameters for more information on parameters.

Annotations: Designated Area Feature

This chapter describes how to access designated area features through Pro/TOOLKIT.

Topic	Page
Introduction to Designated Area Feature	24 - 2
Feature Element Tree for the Designated Area	24 - 2
Accessing Designated Area Properties	24 - 5

Introduction to Designated Area Feature

A Designated Area is a “cosmetic surface” that can be referenced by annotations (including driven dimensions), and propagates as you add geometry to the model. It is used to indicate an area that needs to be closely examined or treated differently.

The designated area is made up of sets of chains constructed by a selection of edges or curves. The curves might lie on a solid (by default), a quilt, or on a plane. If the chains lie on multiple object types, then you must decide on one object type to place the target area.

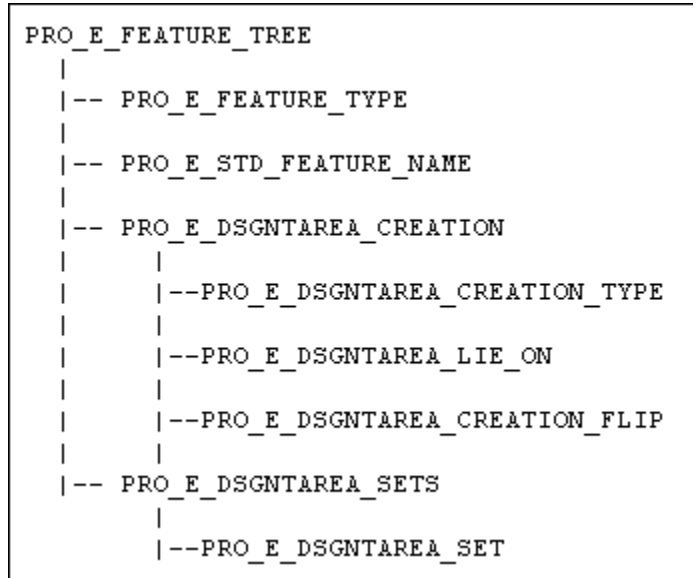
You can attach an annotation to the created surface or to its boundaries. You can also include the designated area as a reference in a data sharing feature, if its parent surface is included.

Geometry of this feature can be accessed using standard Pro/TOOLKIT functions such as **ProFeatureGeomitemVisit()** and **ProSolidQuiltVisit()**.

Feature Element Tree for the Designated Area

The element tree for the Designated Area feature is documented in the header file *ProDesignatedArea.h*. The following figure demonstrates the feature element tree structure:

Figure 24-1: Feature Element Tree for a Designated Area



Annotations:
Designated Area

The following list details special information about the elements in this tree:

- PRO_E_FEATURE_TYPE—Specifies the feature type, should always have the value PRO_FEAT_DSGNT_AREA.
- PRO_E_STD_FEATURE_NAME—Specifies the name of the designated area feature.
- PRO_E_DSGNTAREA_CREATION—Specifies the data used for creation of the designated area. It consists of the following elements:
 - PRO_E_DSGNTAREA_CREATION_TYPE—Specifies whether the designated area lies on a solid, a quilt, or on a plane, where the new surface will be constructed from a chain not related to existing surfaces. It can have one of the following values:
 - PRO_DSGNTA_CR_SOLID—Specifies that the feature is created on a solid model.
 - PRO_DSGNTA_CR_QUILT—Specifies that the feature is created on a selected quilt.
 - PRO_DSGNTA_CR_AIR—Specifies that the feature is created on a plane from a chain of curves not lying on any model surface.

- PRO_E_DSGNTAREA_LIE_ON—Specifies the placement reference of the feature. If the type is PRO_DSGNTA_CR_QUILT, the feature must include the placement quilt. If the type is PRO_DSGNTA_CR_SOLID, it can optionally include a single surface of the solid, which will be the only surface of the solid used in constructing the feature. If you do not specify a single surface, all solid surfaces will be included in the designated area feature references.
- PRO_E_DSGNTAREA_CREATION_FLIP—Specifies the flip option to switch between inside and outside the boundary chains, and can be set to either TRUE or FALSE.
- PRO_E_DSGNTAREA_SETS—Specifies an array of compound elements of the type PRO_E_DSGNTAREA_SET that contain the designated area boundaries.

Element Details of PRO_E_DSGNTAREA_SET

Figure 24-2: PRO_E_DSGNTAREA_SETS

```

|-- PRO_E_DSGNTAREA_SETS
|
|--PRO_E_DSGNTAREA_SET
|
|--PRO_E_DSGNTAREA_SET_REFS
|
|--PRO_E_STD_CHAIN_HOLDER
|
|--PRO_E_STD_CURVE_COLLECTION_APPL

```

Each `PRO_E_DSGNTAREA_SET` contains the following element:

`PRO_E_DSGNTAREA_SET_REFS`—Specifies the set of references to be used for creation of the designated area and consists of the following element:

- `PRO_E_STD_CHAIN HOLDER`—Specifies the set of chains and can consists of one or more of the following element:
 - `PRO_E_STD_CURVE_COLLECTION_APPL`—Specifies the collection of curves to be used as reference. Each curve set must consist of a closed loop. For more information about curve collections, refer to the chapter, User Interface: Curve and Surface Collection.

Accessing Designated Area Properties

The following functions determine the appearance of the designated area by controlling its boundary properties such as the line style and the appearance of the surface.

Functions Introduced:

- `ProDesignatedareaStatusGet()`
- `ProDesignatedareaLinestyleGet()`
- `ProDesignatedareaLinestyleSet()`
- `ProDesignatedareaColorGet()`
- `ProDesignatedareaColorSet()`

The function `ProDesignatedareaStatusGet()` identifies the current status of the geometry of the created designated area. The input value to this function is a quilt from the designated area, or from a data sharing feature referencing the designated area.

The output geometry can have one of the following values:

- `PRO_DSGNTAREA_STATUS_ACTIVE`—Specifies that the geometry is active in the indicated model.
- `PRO_DSGNTAREA_STATUS_INACTIVE`—Specifies that the geometry is inactive due to geometry features occurring after the designated area feature in the regeneration order.
- `PRO_DSGNTAREA_STATUS_OUT_OF_COPIED_GEOM`—Specifies that the parent geometry used from the trimming was cut out from the model.

The function **ProDesignatedareaLinestyleGet()** returns the line style used for the boundary of the designated area. Use the function **ProDesignatedareaLinestyleSet()** to set the line style.

The function **ProDesignatedareaColorGet()** returns the color used for the boundary of the designated area. Use the function **ProDesignatedareaColorSet()** to set the color.

Data Management: Windchill and Pro/INTRALINK Operations

Pro/ENGINEER has the capability to be directly connected to Windchill solutions, including Windchill Foundation, ProjectLink, PDMLink, and ProductPoint servers. This access allows users to manage and control the product data seamlessly from within Pro/ENGINEER.

This chapter lists Pro/TOOLKIT APIs that support Windchill servers and server operations in a connected Pro/ENGINEER session.

Topic	Page
Introduction	25 - 2
Accessing a Windchill Server from a Pro/ENGINEER Session	25 - 3
Accessing the Workspace	25 - 6
Workflow to Register a Server	25 - 8
Aliased URL	25 - 8
Server Operations	25 - 10
Utility APIs	25 - 26
Sample Batch Workflow	25 - 27

Introduction

The functions introduced in this chapter provide support for the basic Windchill server (Windchill Foundation, ProjectLink, PDMLink, and ProductPoint) operations from within Pro/ENGINEER. With these functions, operations such as registering a Windchill server, managing workspaces, and check in or check out of objects will be possible via Pro/TOOLKIT. Windchill ProductPoint does not have the concept of a workspace. New objects are directly stored to the user-specified folder in the Windchill ProductPointer Server. New iterations of the objects are stored in the same folder as the previous iteration. Some functions related to workspace operations may, therefore, not be supported for customizations using Windchill ProductPoint.

The functions specific to Windchill ProductPoint operations are included in the header file *ProWPP.h*. Windchill ProductPoint is available only on Windows. For all other platforms these functions return `PRO_TK_UNSUPPORTED`.

The capabilities of the APIs described in this chapter are similar to the operations available from within the Pro/ENGINEER Wildfire client, with some restrictions.

Non-Interactive Mode Operations

Some of the APIs specified in this section operate only in batch mode and cannot be used in the normal Pro/ENGINEER interactive mode. This restriction is mainly centered around the Pro/TOOLKIT registered servers, that is, servers registered by Pro/TOOLKIT are not available in the Pro/ENGINEER Server Registry or in other locations in the Pro/ENGINEER user interface such as the Folder Navigator and embedded browser. If a Pro/TOOLKIT customization requires the user to have interactive access to the server, the server must be registered via the normal Pro/ENGINEER techniques, that is, either by entry in the Server Registry or by automatic registration of a previously registered server.

All of these APIs are supported from a non-interactive, that is, batch mode application or asynchronous application. For more information about batch mode and asynchronous mode, refer to the chapter “Core: Asynchronous Mode”.

Accessing a Windchill Server from a Pro/ENGINEER Session

Pro/ENGINEER allows you to register Windchill servers as a connection between the Windchill database and Pro/ENGINEER. Although the represented Windchill database can be from Windchill Foundation, Windchill ProjectLink, Windchill PDMLink, or Windchill ProductPoint, all types of databases are represented in the same way.

You can use the following identifiers when referring to Windchill servers in Pro/TOOLKIT:

- **Codebase URL**—This is the root portion of the URL that is used to connect to a Windchill server. For example `http://wcserver.company.com/Windchill`.
- **Server Alias**—A server alias is used to refer to the server after it has been registered. The alias is also used to construct paths to files in the server workspaces and commonspace. The server alias is chosen by the user or application and it need not have any direct relationship to the codebase URL. An alias can be any normal name, such as `my_alias`.

Accessing Information Before Registering a Server

To start working with a Windchill server, you must establish a connection by registering the server in Pro/ENGINEER. The functions described in this section enable you to connect to a Windchill server and access information related to the server.

Functions Introduced:

- **ProBrowserAuthenticate()**
- **ProServerClassGet()**
- **ProServerVersionGet()**
- **ProServerContextsCollect()**
- **ProServerWorkspacesCollect()**

Use the function **ProBrowserAuthenticate()** to set the authentication context using a valid username and password. A successful call to this function allows the Pro/ENGINEER session to register with any server accepting the username and password combination. A successful call to this function also ensures that an

authentication dialog box does not appear during the registration process. You can call this function any number of times to set the authentication context for any number of Windchill servers, provided that you register the appropriate servers or servers immediately after setting the context.

The function **ProServerClassGet()** returns the class of the server. The values are:

- **Windchill**—Denotes either a Windchill Classic PDM server or a Windchill PDMLink server.
- **ProjectLink**—Denotes Windchill ProjectLink type of servers.
- **productpoint**—Denotes a Windchill ProductPoint server.

This function accepts the server codebase URL as the input.

The function **ProServerVersionGet()** returns the version of Windchill that is configured on the server, for example, "7.0" or "8.0." This function accepts the server codebase URL as the input.

Note: **ProServerVersionGet()** works only for Windchill servers and returns the `PRO_TK_UNSUPPORTED` error, if the server is not a Windchill server.

The function **ProServerContextsCollect()** gives a list of all the available contexts for a specified server. A context is used to associate a workspace with a product, project, or library. This function accepts the server codebase URL as the input.

The function **ProServerWorkspacesCollect()** returns the list of available workspaces for the specified server. The workspace data returned contains the name of the workspace and its context. This function accepts the server codebase URL as the input.

Note: This function is not supported for Windchill ProductPoint.

Registering and Activating a Server

The functions described in this section are restricted to the non-interactive mode only. Refer to the section “Non-Interactive Mode Operations” for more information.

Functions Introduced:

- **ProServerRegister()**
- **ProServerActivate()**
- **ProServerUnregister()**

The function **ProServerRegister()** registers the specified server with the codebase URL.

Note: While working with the Windchill ProductPoint server, specify the value of the input argument *workspace* as `NULL` for this function.

A successful call to **ProBrowserAuthenticate()** with a valid username and password is essential for **ProServerRegister()** to register the server without launching the authentication dialog box. Registration of the server establishes the server alias.

The function **ProServerActivate()** sets the specified server as the active server in the Pro/ENGINEER session.

The function **ProServerUnregister()** unregisters the specified server. This is similar to **Server Registry>Delete** through the user interface.

Accessing Information From a Registered Server

Functions Introduced:

- **ProServerContextGet()**
- **ProServerAliasGet()**
- **ProServersCollect()**
- **ProServerLocationGet()**
- **ProServerTargetfolderGet()**
- **ProServerTargetfolderSet()**

The function **ProServerContextGet()** returns the active context of the active server.

Note: This function is not supported while working with a Windchill ProductPoint server.

The function **ProServerAliasGet()** returns the alias of a server if you specify the codebase URL.

The function **ProServersCollect()** returns a list of the aliases of all the registered servers.

The function **ProServerLocationGet()** returns the codebase URL for the server if you specify the alias.

The function **ProServerTargetfolderGet()** returns a location on the Windchill ProductPoint server where you can save new product items. Specify the location of the target folder on the Windchill ProductPoint server using the function **ProServerTargetfolderSet()**. These functions are applicable only when working with a Windchill ProductPoint server.

Accessing the Workspace

For every workspace, a new distinct storage location is maintained in the user's personal folder on the server (server-side workspace) and on the client (client-side workspace cache). Together, the server-side workspace and the client-side workspace cache make up the workspace.

Note: Windchill ProductPoint does not have the concept of a workspace or active workspace. Therefore, many functions in this section are not applicable for this product.

Workspace Data

Functions Introduced:

- **ProServerworkspacedataAlloc()**
- **ProServerworkspacedataNameGet()**
- **ProServerworkspacedataContextGet()**
- **ProServerworkspacedataFree()**
- **ProServerworkspacedataProarrayFree()**

The workspace data is an opaque handle that contains the name and context of the workspace. The function **ProServerWorkspacesCollect()** returns an array of workspace data. Workspace data is also required for the function **ProServerWorkspaceCreate()** to create a workspace with a given name and a specific context.

The function **ProServerworkspacedataAlloc()** creates a workspace data structure to describe a workspace. The workspace contains the name of the workspace and the context in which the workspace is stored.

The function **ProServerworkspacedataNameGet()** retrieves the name of the workspace from the workspace data.

The function **ProServerworkspacedataContextGet()** retrieves the context of the workspace from the workspace data.

Use the function **ProServerworkspacedataFree()** to free the workspace data structure from memory.

Use the function **ProServerworkspacedataProarrayFree()** to free the workspace data array from the memory.

Creating and Modifying the Workspace

Functions Introduced

- **ProServerWorkspaceCreate()**
- **ProServerWorkspaceGet()**
- **ProServerWorkspaceSet()**
- **ProServerWorkspaceDelete()**

All functions described in this section, except **ProServerWorkspaceGet()** are permitted only in the non-interactive mode. Refer to the section “Non-Interactive Mode Operations” for more information.

The function **ProServerWorkspaceCreate()** creates and activates a new workspace.

The function **ProServerWorkspaceGet()** retrieves the name of the active workspace.

The function **ProServerWorkspaceSet()** sets a specified workspace as an active workspace.

The function **ProWorkspaceDelete()** deletes the specified workspace. The function deletes the workspace only if the following conditions are met:

- The workspace is not the active workspace.
- The workspace does not contain any checked out objects.

Use one of the following techniques to delete an active workspace:

- Make the required workspace inactive using **ProServerWorkspaceSet()** with the name of some other workspace and then call **ProServerWorkspaceDelete()**.
- Unregister the server using **ProServerUnregister()** and delete the workspace using the codebase URL instead of the alias.

Workflow to Register a Server

To Register a Server with an Existing Workspace

Perform the following steps to register a Windchill server with an existing workspace:

1. Set the appropriate authentication context using the function **ProBrowserAuthenticate()** with a valid username and password.
2. Look up the list of workspaces using the function **ProServerWorkspacesCollect()**. If you already know the name of the workspace on the server, then ignore this step.
3. Register the workspace using the function **ProServerRegister()** with an existing workspace name on the server.
4. Activate the server using the function **ProServerActivate()**.

To Register a Server with a New Workspace

Perform the following steps to register a Windchill server with a new workspace:

1. Perform steps 1 to 4 in the preceding section to register the Windchill server with an existing workspace.
2. Use the function **ProServerContextsCollect()** to choose the required context for the server.
3. Create a new workspace with the required context using the function **ProServerWorkspaceCreate()**. This function automatically makes the created workspace active.

Note: You can create a workspace only after the server is registered.

Aliased URL

An aliased URL serves as a handle to the server objects. You can access the server objects in the commonspace (shared folders) and the workspace using the aliased URL. An aliased URL is a unique identifier for the server object and the format is as follows:

- Object in workspace has a prefix `wtws`

wtws://<server_alias>/<workspace_name>/<object_server_name>

where <object_server_name> includes
<object_name>.<object_extension>

For example,

wtws://my_server/my_workspace/abcd.prt,
wtws://my_server/my_workspace/intf_file.igs

where

<server_alias> is my_server

<workspace_name> is my_workspace

- Object in commonspace has a prefix wtpub

wtpub://<server_alias>/<folder_location>/<object_server_name>

For example,

wtpub://my_server/path/to/cs_folder/abcd.prt

where

<server_alias> = my_server

<folder_location> = path/to/cs_folder

Note:

- <object_server_name> must be in lowercase.
- The APIs are case-sensitive to the aliased URL.
- <object_extension> should not contain Pro/ENGINEER versions, for example, .1 or .2, and so on.
- For Windchill ProductPoint servers, you can specify a large number of URL variations as long as the base server URL is included. For example,
 - wpp://<Server_Alias>/ProdA/ProENGINEER/Document/jan.prt
 - <Server_Alias>/ProdA/ProENGINEER/Documents/jan.prt

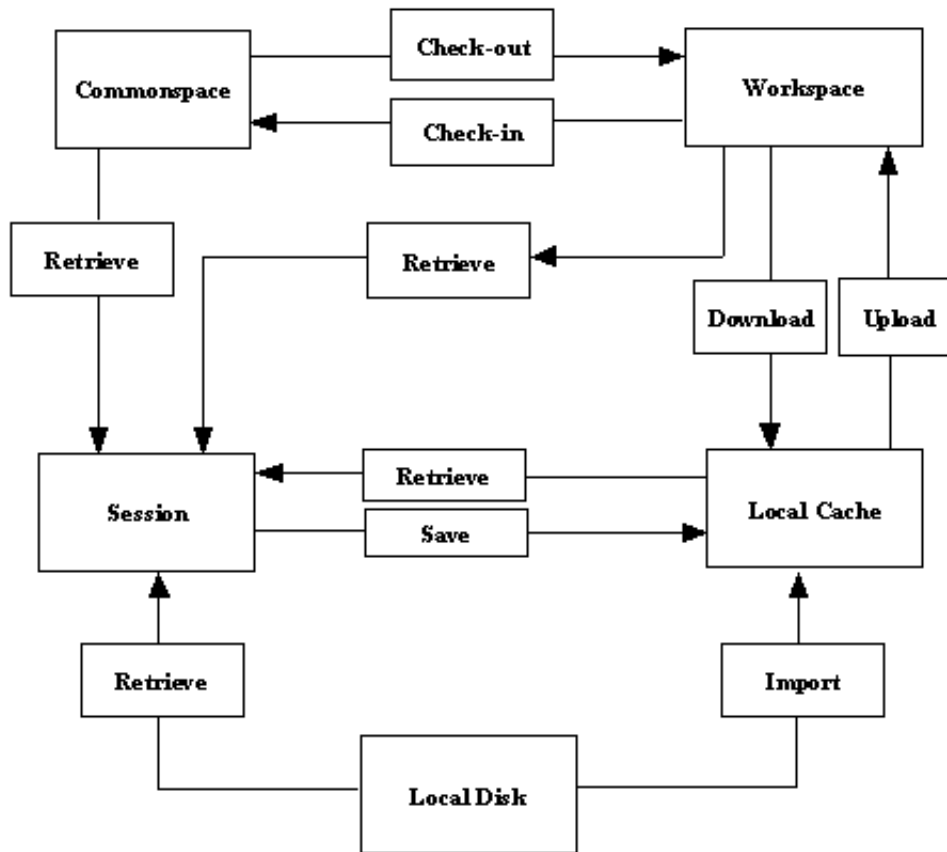
You can also specify only the part name and the object will be accessed from the server, if the server is the default location being explored.

Server Operations

After registering the Windchill server with Pro/ENGINEER, you can start accessing the data on the Windchill servers. The Pro/ENGINEER interaction with Windchill servers leverages the following locations:

- Commonspace (Shared folders)
- Workspace (Server-side workspace)
- Workspace local cache (Client-side workspace)
- Pro/ENGINEER session
- Local disk

The functions described in this section enable you to perform the basic server operations. The following diagram illustrates how data is transferred among these locations.



Save

Functions Introduced:

- **ProMdlSave()**
- **ProServerSynchronize()**
- **ProServerSynchronizationstateGet()**

The function **ProMdlSave()** stores the object from the session in the local workspace cache, when a server is active. For Windchill ProductPoint servers, this function saves an existing model to the location from where it was retrieved. To save a new object to a specified location on the ProductPoint server, first use the function **ProServerTargetSet()** to set the target folder location on the server and then call the function **ProMdlSave()**. If you do not set the target folder location, the function **ProMdlSave()** saves the new objects to the top-level folder of the active product or context.

The function **ProServerSynchronize()** synchronizes the objects in the local cache with the contents in the Windchill ProductPoint server. The function provides an array of conflicts, if any, encountered during the synchronization operation.

The function **ProServerSynchronizationstateGet()** specifies if the contents of the Windchill ProductPoint server are synchronized with the local cache. This function returns true if the server is synchronized, and false, if otherwise.

Upload

An upload transfers Pro/ENGINEER files and any other dependencies from the local workspace cache to the server-side workspace.

Function Introduced:

- **ProServerObjectsUpload()**

The function **ProServerObjectsUpload()** uploads the specified object to the workspace. The object to be uploaded must be present in the current Pro/ENGINEER session. Additionally, ensure that you save the object using the function **ProMdlSave()** before you upload it.

Note: To upload all the objects to the workspace without retrieving them in the current Pro/ENGINEER session, use the function **ProServerObjectsCheckin()** with the checkin option *upload_only* set to PRO_B_TRUE.

CheckIn

After you have finished working on objects in your workspace, you can share the design changes with other users. The checkin operation copies the information and files associated with all changed objects from the workspace to the Windchill database.

Note: The functions described in this section are not applicable to Windchill ProductPoint server operations.

Functions Introduced

- **ProServerObjectsCheckin()**
- **ProServercheckinoptsAlloc()**
- **ProServercheckinoptsDeflocationSet()**
- **ProServercheckinoptsLocationAdd()**
- **ProServercheckinoptsBaselineSet()**
- **ProServercheckinoptsKeepcheckedoutSet()**
- **ProServercheckinoptsAutoresolveSet()**
- **ProServercheckinoptsUploadonlySet()**
- **ProServercheckinoptsFree()**

The function **ProServerObjectsCheckin()** checks in or uploads an object to the database. The object to be checked in or uploaded must be present in the current Pro/ENGINEER session. Changes made to the object are not included unless you save the object to the workspace using the function **ProMdlSave()** before it is checked in or uploaded.

Note: **ProServerObjectsCheckin()** checks in the target object by default. To only upload the object, set the checkin option *upload_only* to PRO_B_TRUE.

If you pass NULL as the value of the input argument *options*, the checkin operation is similar to the **Auto Check-In** option in Pro/ENGINEER. For more details on **Auto Check-In**, refer to the online help for Pro/ENGINEER.

By using an appropriately constructed *options* argument, you can control the checkin or upload operation. The APIs described in this section help in constructing the *options* argument.

The function **ProServercheckinoptsAlloc()** allocates a set of checkin or upload options for the object. These options are as follows:

- **Default location**—Specifies the default *folder_location* on the server for the automatic checkin operation. Use the function **ProServercheckinoptsDeflocationSet()** to set this location.
- **Server location**—Specifies the *folder_location* on the server in which an object will be checked in or uploaded. Use the function **ProServercheckinoptsLocationAdd()** to set this location.

- **Baseline**—Specifies the baseline information for the objects upon checkin. This information does not apply to upload operations. Use the function **ProServercheckinoptsBaselineSet()** to create a new baseline. The baseline information for a checkin operation is as follows:
 - *baseline_name*—Specifies the name of the baseline.
 - *baseline_number*—Specifies the number of the baseline.

The default format for the baseline name and baseline number is “Username + time (GMT) in milliseconds”

 - *baseline_location*—Specifies the location of the baseline.
 - *baseline_lifecycle*—Specifies the name of the lifecycle.
- *keep_checked_out*—If this option is set to `PRO_B_TRUE`, then the contents of the selected object are checked in to the Windchill server and automatically checked out again for further modification. The default value is `PRO_B_FALSE`. This option does not apply to upload operations. Use the function **ProServercheckinoptsKeepcheckedoutSet()** to set the *keep_checked_out* flag.
- *autoresolve*—Specifies the option used to automatically resolve objects that have not been completely checked in or uploaded to the database. The autoresolve options specified by the enumerated type *ProServerAutoresolveOption* are as follows:
 - `PRO_SERVER_DONT_AUTORESOLVE`—Model references missing from the workspace are not automatically resolved. This may result in a conflict upon checkin. This option is used by default.
 - `PRO_SERVER_AUTORESOLVE_IGNORE`—Missing references are automatically resolved by ignoring them.
 - `SERVER_AUTORESOLVE_UPDATE_IGNORE`—Missing references are automatically resolved by updating them in the database and ignoring them if not found.

Use the function **ProServercheckinoptsAutoresolveSet()** to assign the appropriate *autoresolve* option.

- *upload_only*—Specifies the option to fully check in the target object or only upload the object to the server. Set this option to `PRO_B_TRUE` to only upload and not check in the target objects and to `PRO_B_FALSE` to upload and check in the objects. By default, this option is `PRO_B_FALSE`, if not explicitly set, to cause a checkin. Use the function **ProServercheckinoptsUploadonlySet()** to set the *upload_only* flag.

Use the function **ProServercheckinoptsFree()** to free the memory of the checkin options.

Notification Functions

A Pro/TOOLKIT notification is called before you attempt to check in a model through the Pro/ENGINEER user interface. The notification functions are established in a session using the function **ProNotificationSet()**.

Function Introduced:

- **ProCheckinUIPreAction()**

The notification function **ProCheckinUIPreAction()** is called when you select **File>Auto Checkin**, **File>Custom Checkin**, or the corresponding options on the **Model Tree** menu in the Pro/ENGINEER user interface. It is called before any action is performed by the corresponding menu commands. This function is available by calling **ProNotificationSet()** with the value of the notify type as `PRO_CHECKIN_UI_PRE`.

Retrieval

Standard Pro/TOOLKIT provides several functions that are capable of retrieving models. When using these functions with Windchill servers, remember that these functions do not check out the object to allow modifications.

Functions Introduced:

- **ProMdlRetrieve()**
- **ProMdlLoad()**

The function **ProMdlRetrieve()** loads an object into a session given its name and type. This function searches for the object in the active workspace, the local directory, and any other paths specified by the configuration option `search_path`.

The function **ProMdlLoad()** loads an object into session given its path. The path can be a disk path, a workspace path, or a commonspace path. Refer to the section “Aliased URL” for examples of these types of paths. For Windchill ProductPoint servers, this function supports the `instance<generic>` notation for the name of the object.

Checkout and Download

To modify an object from the commonspace, you must check out the object. The process of checking out communicates your intention to modify a design to the Windchill server. The object in the database is locked, so that other users can obtain read-only copies of the object, and are prevented from modifying the object while you have it checked out.

Checkout is often accompanied by a download action, where the objects are brought from the server-side workspace to the local workspace cache. In Pro/TOOLKIT, both operations are covered by the same set of functions.

Note: The functions described in this section are not applicable to Windchill ProductPoint server operations.

Functions Introduced:

- **ProServerObjectsCheckout()**
- **ProServerMultiobjectsCheckout()**
- **ProServercheckoutoptsAlloc()**
- **ProServercheckoutoptsFree()**
- **ProServercheckoutoptsDependencySet()**
- **ProServercheckoutoptsIncludeinstancesSet()**
- **ProServercheckoutoptsVersionSet()**
- **ProServercheckoutoptsDownloadSet()**
- **ProServercheckoutoptsReadonlySet()**

The function **ProServerObjectsCheckout()** checks out and optionally downloads the object to the workspace based on the configuration specifications of the workspace. It takes the following two potential identifiers for the model to checkout:

- **ProMdl handle**—Specifies the object to be checked out. This is applicable if the model has already been retrieved without checking out.

- **Aliased URL**—Specifies the commonspace path of the object.

Use the function **ProServerMultiobjectsCheckout()** to check out and download a *ProArray* of objects to the workspace based on the configuration specifications of the workspace.

If you specify the value of the input argument *checkout* as `PRO_B_TRUE` in the above functions, the selected object is checked out. Otherwise, the object is downloaded without being checked out. The download action enables you to bring read-only copies of objects into your workspace. This allows you to examine the object without placing a lock on it.

If you pass `NULL` as the value of the input argument *options* in the above functions, then the default Pro/ENGINEER checkout rules apply.

The function **ProServercheckoutoptsAlloc()** allocates a set of checkout options for the object. These options are as follows:

- *dependency*—Specifies the dependency rule used while checking out dependents of the object selected for checkout. The function **ProServercheckoutoptsDependencySet()** sets the dependency rule for checkout. The types of dependencies specified by the enumerated type *ProServerDependency* are as follows:
 - `PRO_SERVER_DEPENDENCY_ALL`—All objects that are dependent on the selected object are checked out.
 - `PRO_SERVER_DEPENDENCY_REQUIRED`—All models required to successfully retrieve the originally-selected model from the CAD application are selected for checkout.
 - `PRO_SERVER_DEPENDENCY_NONE`—None of the dependent objects are checked out.
- *include_option*—Specifies the rule for including instances from the family table during checkout. The function **ProServercheckoutoptsIncludeinstancesSet()** sets the flag to include instances during checkout. The type of instances specified by the enumerated type *ProServerInclude* are as follows:
 - `PRO_SERVER_INCLUDE_ALL`—All the instances of the selected object are checked out.
 - `PRO_SERVER_INCLUDE_SELECTED`—The application selects the instance members from the family table to be included during checkout.

- **PRO_SERVER_INCLUDE_NONE**—No additional instances from the family table are added to the object list.
- *version*—Specifies the version of the object that is checked out or downloaded to the workspace. If *version* is not set, the object is checked out according to the current workspace configuration. The function **ProServercheckoutoptsVersionSet()** sets the version of the object.
- *download*—Specifies the checkout type as download or link. The value `download` specifies that the object content is downloaded and checked out, while `link` specifies that only the metadata is downloaded and checked out. Use the function **ProServercheckoutoptsDownloadSet()** to set this option.
- *readonly*—Specifies the checkout type as a read-only checkout. This option is applicable only if the checkout type is `link`. Use the function **ProServercheckoutoptsReadonlySet()** to set the *readonly* flag to `PRO_B_TRUE`. Use this function only while downloading.

Use the function **ProServercheckoutoptsFree()** to free the memory of the checkout options.

The following truth table explains the dependencies of the different control factors in **ProServerObjectsCheckout()** and the effect of different combinations on the end result.

Argument <i>checkout</i> in ProServerObjectsCheckout()	ProServerchecko uptoptsDownload Set()	ProServercheckout optsReadonlySet()	Result
PRO_B_TRUE	PRO_B_TRUE	NA	Object is checked out and its content is downloaded.
PRO_B_TRUE	PRO_B_FALSE	NA	Object is checked out but content is not downloaded.
PRO_B_FALSE	NA	PRO_B_TRUE	Object is downloaded as read-only.

Argument <i>checkout</i> in ProServerObjectsCheckout()	ProServercheckoutoptsDownloadSet()	ProServercheckoutoptsReadonlySet()	Result
PRO_B_FALSE	NA	PRO_B_FALSE	This combination is invalid and is not supported.

Undo Checkout

Function Introduced:

- **ProServerObjectsUndocheckout()**

Use the function **ProServerObjectsUndocheckout()** to undo a checkout of the specified object. When you undo a checkout, the changes that you have made to the content and metadata of the object are discarded and the content, as stored in the server, is downloaded to the workspace. This function is applicable only for the model in the active Pro/ENGINEER session.

Import and Export

Pro/TOOLKIT provides you with the capability of transferring specified objects to and from a workspace. Import and export operations must take place in a session with no models.

Functions Introduced:

- **ProWsimpexFolderoptionSet()**
- **ProCurrentWorkspaceImport()**
- **ProCurrentWorkspaceExport()**
- **ProCurrentWorkspaceImpexMessagesGet()**
- **ProWsimpexmessageDataGet()**
- **ProWsimpexmessageArrayFree()**
- **ProWsexportSecondarycontentoptionSet()**

The function **ProWsimpexFolderoptionSet()** sets the target folder to import data or the source folder to the Windchill ProductPoint servers or to export data from these servers. Set the target folder location using this function before calls to **ProCurrentWorkspaceImport()** and **ProCurrentWorkspaceExport()**. This function is used for Windchill ProductPoint servers only.

The function **ProCurrentWorkspaceImport()** imports specified objects from disk to the current workspace in a linked session of Pro/ENGINEER. For Windchill ProductPoint servers this function copies files from the local disk to the specified target folder on the server.

The function **ProCurrentWorkspaceExport()** exports the specified objects from the current workspace to a location on disk in a linked session of Pro/ENGINEER. For Windchill ProductPoint servers this function exports files from the specified source folder on the server to a disk.

Both **ProCurrentWorkspaceImport()** and **ProCurrentWorkspaceExport()** allow you to specify a dependency criterion to process the following items:

- All external dependencies
- Only required dependencies
- No external dependencies

All warnings, conflicts, or errors generated during import or export operations are logged in the *proimpex.errors* file created in the Pro/ENGINEER working directory. Alternatively, you can obtain this information using the function

ProCurrentWorkspaceImpexMessagesGet(). This function returns a *ProArray* of messages generated by the last call to **ProCurrentWorkspaceImport()** or **ProCurrentWorkspaceExport()**.

The function **ProWsimpexmessageDataGet()** extracts the contents of the message generated by **ProCurrentWorkspaceImport()** or **ProCurrentWorkspaceExport()**. The message contains the following items:

- *type*—Specifies the severity of the message in the form of the enumerated type *ProWSImpexMessageType*. The severity is one of the following types:
 - PRO_WSIMPEX_MSG_INFO—Specifies an informational type of message.
 - PRO_WSIMPEX_MSG_WARNING—Specifies a low severity problem that can be resolved according to the configured rules.
 - PRO_WSIMPEX_MSG_CONFLICT—Specifies a conflict that can be overridden.

- **PRO_WSIMPEX_MSG_ERROR**—Specifies a conflict that cannot be overridden or a serious problem that prevents processing of an object.
- *object*—Specifies the object name or the name of the object path described in the message.
- *description*—Specifies the description of the problem or the message information.
- *resolution*—Specifies the resolution applied to resolve a conflict that can be overridden. This is applicable when the message is of the type **PRO_WSIMPEX_MSG_CONFLICT**.
- *succeeded*—Determines whether the resolution succeeded or not. This is applicable when the message is of the type **PRO_WSIMPEX_MSG_CONFLICT**.

Use the function **ProWsimpexmessageArrayFree()** to free the memory allocated to the array of messages returned by **ProCurrentWorkspaceImpexMessagesGet()**.

The function **ProWsexportSecondarycontentoptionSet()** sets the *ProBoolean* option that controls the export of secondary contents. If this option is set to **PRO_B_TRUE**, secondary contents are exported along with the primary Pro/ENGINEER model files. By default, it is **PRO_B_TRUE**.

File Copy

Pro/TOOLKIT provides you with the capability of copying a file from the workspace or target folder to a location on the disk and vice-versa.

Functions Introduced:

- **ProFileCopyToWS()**
- **ProFileCopyFromWS()**

Use the function **ProFileCopyToWS()** to copy a file from disk to the workspace. The file can optionally be added as secondary content to a given workspace file. For Windchill ProductPoint servers, use this function to copy a viewable file from disk as a new item in the target folder specified by the function **ProServerTargetfolderSet()**. If the viewable file is added as secondary content, a dependency is created between the Pro/ENGINEER model and the viewable file.

Use the function **ProFileCopyFromWS()** to copy a file from the workspace to a location on disk. For Windchill ProductPoint servers, use this function to copy a single file from the current target folder specified by the function **ProServerTargetFolderSet()** to the local disk.

Note: When importing or exporting Pro/ENGINEER models, PTC recommends that you use **ProCurrentWorkspaceImport()** and **ProCurrentWorkspaceExport()**, respectively, to perform the operation. Functions that copy individual files do not traverse Pro/ENGINEER model dependencies, and therefore do not copy a fully retrievable set of models at the same time. Additionally, only the functions **ProCurrentWorkspaceImport()** and **ProCurrentWorkspaceExport()** provide full metadata exchange and support. That means **ProCurrentWorkspaceImport()** can communicate all the Pro/ENGINEER designated parameters, dependencies, and family table information to a PDM system while **ProCurrentWorkspaceExport()** can update exported Pro/ENGINEER data with PDM system changes to designated and system parameters, dependencies, and family table information. Hence PTC recommends the use of **ProFileCopyToWS()** and **ProFileCopyFromWS()** to process only non-Pro/ENGINEER files.

Server Object Status

Function Introduced:

- **ProServerObjectIsModified()**

The function **ProServerObjectIsModified()** verifies the current status of the object in the workspace as well as in the local workspace cache. The status of the object is as follows:

- `checkout_status`—Specifies whether the object is checked out for modification.
- `modifiedInWS`—Specifies whether the object has been modified in the workspace since checkout. The value of this argument is `false` if the newly created object has not been uploaded.

- **modifiedLocally**—Specifies whether the object has been modified in the local workspace cache since checkout. The value of this argument is `false` if the newly created object has not been saved after modifying it in the local workspace cache.

Note:

- This function is not applicable for Windchill ProductPoint server operations.
- The function **ProServerObjectStatusGet()** is deprecated. Use the function **ProServerObjectIsModified()** instead.

Object Lock Status

In comparison with other Windchill servers, Windchill ProductPoint does not have the concept of a workspace. This means that as soon as changes are saved to the server, they are visible to all users with access to work-in-progress versions. The functions described in this section enable you to establish an exclusive lock when modifying a server-managed part in Pro/ENGINEER. The functions described in this section are applicable only for Windchill ProductPoint server operations.

Functions Introduced:

- **ProServerObjectsLock()**
- **ProServerlockConflictinfoGet()**
- **ProServerObjectIsLocked()**
- **ProServerObjectsAreLocked()**
- **ProServerlockStatusinfoGet()**
- **ProServerlockStatusFree()**
- **ProServerlockStatusArrayFree()**
- **ProServerObjectsUnlock()**
- **ProServerlockConflictArrayFree()**

The function **ProServerObjectsLock()** establishes an explicit lock on the specified objects on the server. Specify the full path, name, and extension for the input objects. The function returns an array of conflicts, if any, that occurred during the lock operation. Use the function **ProServerlockConflictinfoGet()** to extract the details of the conflicts that occurred during the lock operation. It provides the name of the object and the type of conflict that occurred during the lock operation.

The function **ProServerObjectIsLocked()** checks the lock status of the specified object on the Windchill ProductPoint server. Specify the full path, name, and extension for the input object. The function **ProServerObjectsAreLocked()** checks the lock status of a set of objects on the Windchill ProductPoint server. Specify the full path, name, and extension for the input objects.

Both these functions provide a handle to the server lock status which is passed as the input to the function **ProServerlockStatusinfoGet()**. This function provides the following details as output arguments:

- *object*—Specifies the name of the object including the extension.
- *lock_status*—Specifies the status of the lock on the object on the server.
 - PRO_OBJ_LOCK_STAT_UNSET—Specifies that no lock has been applied on the object.
 - PRO_OBJ_LOCK_STAT_HARDLOCK—Specifies that objects are locked and that the current user is not the owner of the locks. Therefore, this user cannot modify or release the lock.
 - PRO_OBJ_LOCK_STAT_SOFTLOCK—Specifies that the objects are locked and the current user is the owner of the locks. Therefore, this user can release the lock.
 - PRO_OBJ_LOCK_STAT_UNLOCKED—Specifies that the explicit or implicit lock has been removed from the object and it is available for editing.
- *message*—Specifies the status of the object on the server. It provides the name of the user who locked the object and the time of locking.

Use the function **ProServerlockStatusFree()** and **ProServerlockStatusArrayFree()** to free the server status from the memory for functions **ProServerObjectIsLocked()** and **ProServerObjectsAreLocked()**, respectively.

The function **ProServerObjectsUnlock()** unlocks a set of objects that have been explicitly locked on the product server. The function returns an array of conflicts, if any, that occurred during the unlock operation.

Use the function **ProServerlockConflictArrayFree()** to free the memory allocated for the array of conflicts output by **ProServerObjectsLock()** or **ProServerObjectsUnlock()**.

Delete Objects

Function Introduced:

- **ProServerObjectsRemove()**

The function **ProServerObjectsRemove()** deletes the array of objects from the workspace. When passed with the *mode_names* array as *NULL*, this function removes all the objects in the active workspace.

Conflicts During Server Operations

Functions Introduced:

- **ProServerconflictsDescriptionGet()**
- **ProServerconflictsFree()**

Conflict objects are provided to capture the error condition while performing the following server operations using the specified APIs:

Operation	API	Error Object
Checkin an object or workspace	ProServerObjectsCheckin()	ProServerCheckinConflicts
Checkout an object	ProServerObjectsCheckout()	ProServerCheckoutConflicts
Undo checkout of an object	ProServerObjectsUndocheckout()	ProServerUndoCheckout Conflicts
Upload object	ProServerObjectsUpload()	ProServerUploadConflicts
Download object	ProServerObjectsCheckout() (with download as PRO_B_TRUE)	ProServerCheckoutConflicts
Delete workspace	ProServerWorkspaceDelete()	ProServerDeleteConflicts
Remove object	ProServerObjectsRemove()	ProServerRemoveConflicts

These APIs return a common status *PRO_TK_CHECKOUT_CONFLICT* and a conflict object *ProServerConflicts*. The conflict object is used to get more details about the error condition.

Use the function **ProServerconflictsDescriptionGet()** to extract details of the error condition. This description is similar to the description displayed by the Pro/ENGINEER HTML User Interface in the conflict report.

The function **ProServerconflictsFree()** frees the memory of the conflict structure returned by the functions.

Utility APIs

The functions specified in this section enable you to obtain the handle to the server objects to access them. The handle may be the Aliased URL or the model name of the http URL. These utilities enable the conversion of one type of handle to the other type.

Functions Introduced:

- **ProServerModelNameToAliasedURL()**
- **ProServerAliasedURLToModelName()**
- **ProServerAliasedURLToURL()**

The function **ProServerModelNameToAliasedURL()** enables you to search for a server object by its name. Specify the complete file name of the object as the input, for example, `test_part.prt`. The function returns the aliased URL for a model on the server. For more information regarding the aliased URL, refer to the section “Aliased URL”. During the search operation, the workspace takes precedence over the shared space.

You can also use this function to search for files that are not in the Pro/ENGINEER format. For example, `my_text.txt`, `prodev.dat`, `intf_file.stp`, and so on.

The function **ProServerAliasedURLToModelName()** returns the name of the object from the given aliased URL on the server.

The function **ProServerAliaseURLToURL()** converts an aliased URL to a standard URL to the objects on the server.

For example, `wtws://my_alias/Wildfire/abcd.prt` is converted to an appropriate URL on the server as `http://server.mycompany.com/Windchill`.

For Windchill ProductPoint, the aliased URL `wpp://<Server_Alias>/ProdA/ProENGINEER/Document/jan.prt` is converted to an appropriate URL on server, for example, `http://server.mycompany.com/`.

Sample Batch Workflow

A typical workflow using the Windchill APIs for an asynchronous non-graphical application is as follows:

1. Start a Pro/ENGINEER session using the function **ProEngineerConnectionStart()**.
2. Authenticate the browser using the function **ProBrowserAuthenticate()**.
3. Register the server with the new workspace using the function **ProServerRegister()**.
4. Activate the server using the function **ProServerActivate()**.
5. Check out and retrieve the model from the vault URL using the function **ProServerObjectsCheckout()** followed by **ProMdlRetrieve()**.
6. Modify the model according to the application logic.
7. Save the model to the workspace using the function **ProMdlSave()**.
8. Check in the modified model back to the server using the function **ProServerObjectsCheckin()**.
9. After processing all models, unregister from the server using the function **ProServerUnregister()**.
10. Delete the workspace using **ProServerWorkspaceDelete()**.
11. Stop Pro/ENGINEER using the function **ProEngineerEnd()**.

Interface: Data Exchange

This chapter describes various methods of importing and exporting files in Pro/TOOLKIT.

Topic	Page
Exporting Information Files	26 - 2
Exporting 2D Models	26 - 5
Automatic Printing of 3D Models	26 - 16
Exporting 3D Models	26 - 21
Shrinkwrap Export	26 - 31
Exporting to PDF and U3D	26 - 37
Importing Parameter Files	26 - 46
Importing 2D Models	26 - 48
Importing 3D Models	26 - 48

Exporting Information Files

Functions Introduced:

- **ProOutputFileWrite()**

The function **ProOutputFileWrite()** is used to create files of several types from data in Pro/ENGINEER. This function operates only on the current object. The file types are declared in *ProUtil.h*. The export formats and their type constants are as listed in the following table.

Export Format	Pro/TOOLKIT function	Type Constant
Bills of material	ProOutputFileWrite()	PRO_BOM_FILE
Drawing setup file		PRO_DWG_SETUP_FILE
Feature identifier		PRO_FEAT_INFO, PRO_FEAT_INFO
Material file (currently assigned material)		PRO_MATERIAL_FILE
CL Data output, NC Sequence file		PRO_MFG_FEAT_CL
CL Data operation file		PRO_MFG_OPER_CL
Information on Pro/ENGINEER Objects		PRO_MODEL_INFO
Program file		PRO_PROGRAM_FILE
Cable Parameters file		PRO_CABLE_PARAMS_FILE
Connector Parameters file		PRO_CONNECTOR_PARAMS_FILE
Spool file		PRO_SPOOL_FILE
Difference Report file		PRO_DIFF_REPORT_FILE
IGES file		PRO_IGES_FILE
DXF file		PRO_DXF_FILE
DWG file		PRO_DWG_FILE
Render file		PRO_RENDER_FILE
SLA ASCII file		PRO_SLA_ASCII_FILE

Export Format	Pro/TOOLKIT function	Type Constant
SLA Binary file	ProOutputFileWrite()	PRO_SLA_BINARY_FILE
INVENTOR file		PRO_INVENTOR_FILE
CATIA facets file		PRO_CATIAFACETS_FILE
IGES 3D file		PRO_IGES_3D_FILE
STEP file		PRO_STEP_FILE
VDA file		PRO_VDA_FILE
SET file		PRO_SET_FILE
FIAT file		PRO_FIAT_FILE
CATIA file		PRO_CATIA_FILE
CATIA DIRECT file		PRO_CATIA_DIRECT_FILE
ACIS file		PRO_ACIS_FILE
CGM file		PRO_CGM_FILE

The option **PRO_RELATION_FILE** creates a file that contains a list of all the model relations and parameters.

To access parameters on the connector entry ports, you must call the function **ProOutputFileWrite()** with the option **PRO_CONNECTOR_PARAMS_FILE**. The function writes a text file to disk. This text file is in the same format as the file that you edit when using the Pro/ENGINEER command **Connector>Modify Parameters>Mod Param**.

To generate and export a difference report to text or CSV format, call the function **ProOutputFileWrite()** with the option **PRO_DIFF_REPORT_FILE**.

Note: As this report is generated and exported from the Pro/ENGINEER embedded browser, using this output type will cause Pro/ENGINEER to show the difference report in the browser.

For some of the options used with **ProOutputFileWrite()**, you must provide some more information, using the last four arguments. The following list shows the arguments to be set and when:

- For **PRO_RENDER_FILE**, **PRO_INVENTOR_FILE**, **PRO_CATIAFACETS_FILE**, **PRO_SLA_ASCII_FILE**, and **PRO_SLA_BINARY_FILE**, set the following argument:
 - *arg1*—The name of the coordinate system. If this NULL, the function uses the default coordinate system.

- For PRO_SPOOL_FILE, set *arg1* to the spool name.
- For PRO_FEAT_INFO, PRO_MFG_FEAT_CL, and PRO_MFG_OPER_CL, set the following argument:
 - arg2*—The integer identifier of the feature.
- For PRO_IGES_3D_FILE, PRO_STEP_FILE, PRO_VDA_FILE, PRO_SET_FILE, PRO_FIAT_FILE, PRO_CATIA_FILE, PRO_CATIA_DIRECT_FILE, or PRO_ACIS_FILE, set the following argument:
 - arg2*—The integer pointer to an odd or even number.
- For PRO_CGM_FILE, set the following arguments:
 - *arg2*—Represents the integer pointer to the export type PRO_EXPORT_CGM_CLEAR_TEXT or PRO_EXPORT_CGM_MIL_SPEC.
 - *arg3*—Represents the integer pointer to the scalar type PRO_EXPORT_CGM_ABSTRACT or PRO_EXPORT_CGM_METRIC.
- For PRO_CONNECTOR_PARAMS, set the following arguments:
 - *arg1*—Represents the integer pointer to *ProIdTable*. *ProIdTable* is an integer array of component identifiers.
 - *arg2*—Represents the integer pointer to the number of component identifiers.
- For PRO_CABLE_PARAMS_FILE, set the following arguments:
 - *arg1*—Represents a *ProSolid* (part pointer).
 - *arg2*—Represents the cable name.
- For PRO_DIFF_REPORT_FILE, set the following argument:
 - arg4*—Represents the model to which the input model is compared to generate the difference report.

Exporting 2D Models

Functions Introduced:

- **Pro2dExport()**
- **Pro2dExportdataAlloc()**
- **Pro2dExportdataFree()**
- **Pro2dExportdataSheetoptionSet()**
- **Pro2dExportdataSheetsSet()**
- **Pro2dExportdataModelspacesheetSet()**
- **ProProductviewexportoptsAlloc()**
- **ProProductviewexportoptsFree()**
- **ProProductviewexportoptsFormatSet()**
- **ProProductviewFormattedExport()**
- **ProPrintPrinterOptionsGet()**
- **ProPrintMdlOptionsGet()**
- **ProPrintPlacementOptionsGet()**
- **ProPrintPCFOptionsGet()**
- **ProPrintExecute()**

Interface: Data
Exchange

Export Format	Pro/TOOLKIT Functions	Type Constant
STEP	Pro2dExport()	PRO_STEP_FILE
SET		PRO_SET_FILE
IGES		PRO_IGES_FILE
MEDUSA		PRO_MEDUSA_FILE
DXF		PRO_DXF_FILE
DWG		PRO_DWG_FILE
CGM		PRO_CGM_FILE
TIFF		PRO_SNAP_TIFF_FILE
Stheno		PRO_STHENO_FILE
DXF	ProOutputFileWrite()	PRO_DXF_FILE
DWG		PRO_DWG_FILE
CGM file		PRO_CGM_FILE

Export Format	Pro/TOOLKIT Functions	Type Constant
PVS file, Plot file	ProProductviewFormattedExport()	PRO_PV_FORMAT_PVS
ED file, Plot file		PRO_PV_FORMAT_ED
EDZ file		PRO_PV_FORMAT_EDZ
PVZ file		PRO_PV_FORMAT_PVZ
Plot file	ProPrintExecute()	N/A

The function **Pro2dExport()** exports existing two-dimensional models into a single object file. The exported model can be a single drawing, layout or diagram, or multiple sheets of a drawing. It supports the STEP, SET, IGES, Medusa, DXF, CGM, TIFF, Stheno, and DWG formats. The interface file obtained using the function **Pro2dExport()** is controlled by one of its input argument *data*, an instance of the *Pro2dExportdata* object. Note that the *data* argument is optional; you do not have to specify it when exporting only the current sheet of the 2D model. Additionally, several Pro/ENGINEER configuration options related to entity type export options can affect the results of the export operation. Refer to the Pro/ENGINEER Online Help for details on the configuration options.

Example 1 Publishing a Drawing

The following example shows how to publish a drawing in the given context.

```

/-----\
FILE      : UgDwgPublishContext.c
PURPOSE   : Pro/TOOLKIT User Guide Example - drawing Publish/Export context
\-----/
/*----- Pro/Toolkit includes -----*/
#include <ProToolkit.h>
#include <ProMdl.h>
#include <ProUtil.h>
#include <ProMessage.h>

/*----- Application includes -----*/
#include <TestError.h>

#define MSGFIL L##"msg_ugdrawing.txt"
/*-----*\
FUNCTION   : UgDwgPublishContext()
PURPOSE    : Publishes a drawing In given context
\-----*/

```

```

ProError UgDwgPublishContext_step()
{
    ProError status;
    ProPath export_filename;
    ProMdl drawing_to_export = NULL;
    ProMdlType mdl_type;

    status = ProMdlCurrentGet ((ProMdl*)&drawing_to_export);
    ERROR_CHECK("UgDwgPublishContext_step", "ProMdlCurrentGet", status);

    if (status == PRO_TK_NO_ERROR)
    {
        status = ProMdlTypeGet(drawing_to_export, &mdl_type);
        ERROR_CHECK("UgDwgPublishContext_step", "ProMdlTypeGet", status);

        if(mdl_type == PRO_MDL_DRAWING)
        {
            ProStringToWstring( export_filename ,
                                "../exported_drawing_step.stp" );

            status = Pro2dExport ( PRO_STEP_FILE, export_filename,
                                   drawing_to_export, NULL );
            ERROR_CHECK("UgDwgPublishContext_step", "Pro2dExport", status);

            if(status == PRO_TK_NO_ERROR)
                ProMessageDisplay(MSGFIL, "USER %0s has been created",
                                   "STEP file exported_drawing_step.stp");
        }
        else
            status = ProMessageDisplay(MSGFIL, "USER %0s",
                                       "Please open the drawing to publish");
    }

    else
        status = ProMessageDisplay(MSGFIL, "USER %0s",
                                   "Please open the drawing to publish");

    return(status);
}

ProError UgDwgPublishContext_iges()
{
    ProError status;
    ProPath export_filename;
    ProMdl drawing_to_export = NULL;
    ProMdlType mdl_type;

    status = ProMdlCurrentGet ((ProMdl*)&drawing_to_export);
    ERROR_CHECK("UgDwgPublishContext_iges", "ProMdlCurrentGet", status);

```

```

if (status == PRO_TK_NO_ERROR)
{
    status = ProMdlTypeGet(drawing_to_export, &mdl_type);
    ERROR_CHECK("UgDwgPublishContext_step", "ProMdlTypeGet", status);

    if(mdl_type == PRO_MDL_DRAWING)
    {
        ProStringToWstring( export_filename ,
                           "../exported_drawing_iges.igs" );

        status = Pro2dExport ( PRO_IGES_FILE, export_filename,
                               drawing_to_export, NULL );
        ERROR_CHECK("UgDwgPublishContext_iges", "Pro2dExport", status);

        if(status == PRO_TK_NO_ERROR)
            status = ProMessageDisplay(MSGFIL, "USER %0s has been created",
                                       "IGES file exported_drawing_iges.igs");
    }
    else
        status = ProMessageDisplay(MSGFIL, "USER %0s",
                                   "Please open the drawing to publish");
}
else
    status = ProMessageDisplay(MSGFIL, "USER %0s",
                              "Please open the drawing to publish");

return(status);
}

ProError UgDwgPublishContext_dxf()
{
    ProError status;
    ProPath export_filename;
    ProMdl drawing_to_export = NULL;
    ProMdlType mdl_type;

    status = ProMdlCurrentGet ((ProMdl*)&drawing_to_export);
    ERROR_CHECK("UgDwgPublishContext_dxf", "ProMdlCurrentGet", status);

    if (status == PRO_TK_NO_ERROR)
    {
        status = ProMdlTypeGet(drawing_to_export, &mdl_type);
        ERROR_CHECK("UgDwgPublishContext_step", "ProMdlTypeGet", status);

        if(mdl_type == PRO_MDL_DRAWING)
        {
            ProStringToWstring( export_filename , "../exported_drawing_dxf.dxf" );

            status = Pro2dExport ( PRO_DXF_FILE, export_filename,
                                   drawing_to_export, NULL );
            ERROR_CHECK("UgDwgPublishContext_dxf", "Pro2dExport", status);
        }
    }
}

```

```

        if(status == PRO_TK_NO_ERROR)
            status = ProMessageDisplay(MSGFIL, "USER %0s has been created",
                                      "DXF file exported_drawing_dxf.dxf");
    }
    else
        status = ProMessageDisplay(MSGFIL, "USER %0s",
                                   "Please open the drawing to publish");
    }
    else
        status = ProMessageDisplay(MSGFIL, "USER %0s",
                                   "Please open the drawing to publish");
    return(status);
}

ProError UgDwgPublishContext_set()
{
    ProError status;
    ProPath export_filename;
    ProMdl drawing_to_export = NULL;
    ProMdlType mdl_type;

    status = ProMdlCurrentGet ((ProMdl*)&drawing_to_export);
    ERROR_CHECK("UgDwgPublishContext_set", "ProMdlCurrentGet", status);

    if (status == PRO_TK_NO_ERROR)
    {
        status = ProMdlTypeGet(drawing_to_export, &mdl_type);
        ERROR_CHECK("UgDwgPublishContext_step", "ProMdlTypeGet", status);

        if(mdl_type == PRO_MDL_DRAWING)
        {
            ProStringToWstring( export_filename , "../exported_drawing_set.set" );

            status = Pro2dExport (PRO_SET_FILE , export_filename,
                                drawing_to_export, NULL );

            ERROR_CHECK("UgDwgPublishContext_set", "Pro2dExport", status);

            if(status == PRO_TK_NO_ERROR)
                status = ProMessageDisplay(MSGFIL, "USER %0s has been created",
                                          "SET file exported_drawing_set.set");
        }
        else
            status = ProMessageDisplay(MSGFIL, "USER %0s",
                                       "Please open the drawing to publish");
    }
    else
        status = ProMessageDisplay(MSGFIL, "USER %0s",
                                   "Please open the drawing to publish");
    return(status);
}

```

```

}

ProError UgDwgPublishContext_medusa()
{
    ProError status;
    ProPath export_filename;
    ProMdl drawing_to_export = NULL;
    ProMdlType mdl_type;

    status = ProMdlCurrentGet ((ProMdl*)&drawing_to_export);
    ERROR_CHECK("UgDwgPublishContext_medusa", "ProMdlCurrentGet", status);

    if (status == PRO_TK_NO_ERROR)
    {
        status = ProMdlTypeGet(drawing_to_export, &mdl_type);
        ERROR_CHECK("UgDwgPublishContext_step", "ProMdlTypeGet", status);

        if(mdl_type == PRO_MDL_DRAWING)
        {
            ProStringToWstring( export_filename , "../exported_drawing_medusa" );

            status = Pro2dExport ( PRO_MEDUSA_FILE, export_filename,
            drawing_to_export, NULL );
            ERROR_CHECK("UgDwgPublishContext_medusa", "Pro2dExport", status);

            if(status == PRO_TK_NO_ERROR)
                status = ProMessageDisplay(MSGFIL, "USER %0s has been created",
                "medusa file s.exported_drawing_medusa");
        }
        else
            status = ProMessageDisplay(MSGFIL, "USER %0s",
            "Please open the drawing to publish");
    }
    else
        status = ProMessageDisplay(MSGFIL, "USER %0s",
        "Please open the drawing to publish");

    return(status);
}

ProError UgDwgPublishContext_cgm()
{
    ProError status;
    ProPath export_filename;
    ProMdl drawing_to_export = NULL;
    ProMdlType mdl_type;

    status = ProMdlCurrentGet ((ProMdl*)&drawing_to_export);
    ERROR_CHECK("UgDwgPublishContext_cgm", "ProMdlCurrentGet", status);

    if (status == PRO_TK_NO_ERROR)
    {

```

```

        status = ProMdlTypeGet(drawing_to_export, &mdl_type);
        ERROR_CHECK("UgDwgPublishContext_step", "ProMdlTypeGet", status);

        if(mdl_type == PRO_MDL_DRAWING)
        {
ProStringToWstring( export_filename , "./exported_drawing_cgm.cgm" );

        status = Pro2dExport ( PRO_CGM_FILE, export_filename,
drawing_to_export, NULL );
        ERROR_CHECK("UgDwgPublishContext_cgm", "Pro2dExport", status);

        if(status == PRO_TK_NO_ERROR)
            status = ProMessageDisplay(MSGFIL, "USER %0s has been created",
                                        "CGM file exported_drawing_cgm.cgm");
        }
        else
            status = ProMessageDisplay(MSGFIL, "USER %0s",
                                        "Please open the drawing to publish");
        }
        else
            status = ProMessageDisplay(MSGFIL, "USER %0s",
                                        "Please open the drawing to publish");
        return(status);
    }

ProError UgDwgPublishContext_tiff()
{
    ProError status;
    ProPath export_filename;
    ProMdl drawing_to_export = NULL;
    ProMdlType mdl_type;

    status = ProMdlCurrentGet ((ProMdl*)&drawing_to_export);
    ERROR_CHECK("UgDwgPublishContext_tiff", "ProMdlCurrentGet", status);

    if (status == PRO_TK_NO_ERROR)
    {
        status = ProMdlTypeGet(drawing_to_export, &mdl_type);
        ERROR_CHECK("UgDwgPublishContext_step", "ProMdlTypeGet", status);

        if(mdl_type == PRO_MDL_DRAWING)
        {
ProStringToWstring( export_filename , "./exported_drawing_tiff.tif" );

        status = Pro2dExport ( PRO_SNAP_TIFF_FILE, export_filename,
drawing_to_export, NULL );
        ERROR_CHECK("UgDwgPublishContext_tiff", "Pro2dExport", status);

        if(status == PRO_TK_NO_ERROR)
            status = ProMessageDisplay(MSGFIL, "USER %0s has been created",
                                        "TIFF file exported_drawing_tiff.tif");
        }
    }
}

```

```

    }
    else
        status = ProMessageDisplay(MSGFIL, "USER %0s",
                                   "Please open the drawing to publish");
}
else
    status = ProMessageDisplay(MSGFIL, "USER %0s",
                               "Please open the drawing to publish");
return(status);
}

```

The functions **Pro2dExportdataAlloc()** and **Pro2dExportdataFree()** allocate and free the memory for the *Pro2dExportdata* object containing the 2D export options and flags.

The function **Pro2dExportdataSheetoptionSet()** assigns the sheet export option for export to the specified 2D format. The sheet export option can take one of the following values:

- **PRO2DEXPORT_CURRENT_TO_MODEL_SPACE**—Specifies that only the drawing's current sheet will be exported as model space into the specified 2D format. This is the default value.
- **PRO2DEXPORT_CURRENT_TO_PAPER_SPACE**—Specifies that only the drawing's current sheet will be exported as paper space into the specified 2D format. This value is only available for formats that support the concept of model space and paper space, for example, DXF and DWG.
- **PRO2DEXPORT_ALL**—Specifies that all the sheets in a drawing will be exported into the specified 2D format as paper space, if applicable to the format type.
- **PRO2DEXPORT_SELECTED**—Specifies that selected sheets in a drawing will be exported as paper space and one sheet will be exported as model space.

The function **Pro2dExportdataSheetsSet()** assigns the sheet numbers to be exported as paper space to the specified 2D export format file. Use this function only if the sheet export option is set to **PRO2DEXPORT_SELECTED**.

The function **Pro2dExportdataModelspacesheetSet()** assigns the sheet number to be exported as model space. Use this function only if the export format supports the concept of model space and paper space, and if the sheet export option is set to **PRO2DEXPORT_SELECTED**.

The functions **ProProductviewexportoptsAlloc()** and **ProProductviewexportoptsFree()** allocate and free the memory assigned to the *ProProductviewExportOptions* object containing the ProductView export formats.

The function **ProProductviewexportoptsFormatSet()** assigns the flag specifying the ProductView export format.

The function **ProProductviewFormattedExport()** exports a drawing to one of the following user-defined ProductView formats:

- PRO_PV_FORMAT_PVS
- PRO_PV_FORMAT_ED
- PRO_PV_FORMAT_EDZ
- PRO_PV_FORMAT_PVZ

Use the function **ProPrintPrinterOptionsGet()** to get the options for a specified printer. Specify the printer type as the input argument for this function. The supported printer types are:

- POSTSCRIPT—Generic Postscript
- COLORPOSTSC—Color Postscript
- MS_PRINT_MGR—MS Print Manager

Note: For a list of all supported printers, please refer to the **Add Printer Type** list in Pro/ENGINEER's **Printer Configuration** dialog box.

The function gets the initialized printer options. The options include the file related options, print command options and printer specific options as follows:

- File Related:
 - *save_to_file*—Saves a plot to a file.
 - *save_method*—Specifies if the plot is to be appended to a file, saved to a single file, or saved to multiple files.
 - *filename*—Specifies the name of the file to which the plot is saved.
 - *delete_after*—Deletes the plot file after printing.
- Print Command:
 - *send_to_printer*—Sends the plot directly to the printer.
 - *print_command*—Specifies the complete command that you want to use for printing.

- *pen_table*—Specifies the complete path to the file containing the pen table.
 - *paper_size*—Specifies the size of the paper to be printed.
 - *quantity*—Specifies the number of copies to be printed.
 - **Printer Specific:**
 - *sw_handshake*—Specifies the type of the handshake initialization sequence. Specify the value as `True` to set the initialization sequence to Software and as `False` to set it to Hardware.
- Note:** Consult your system administrator for more information on handshaking.
- *roll_media*—Specifies whether to use roll-media or cut-sheet.
 - *use_ttf*—Specifies whether to use TrueType font or stroked text.
 - *slew*—Specifies the speed of the pen in centimeters per second in X and Y direction.
 - *rotate_plot*—Specifies that the plot is to be rotated by 90 degrees.

Use the function **ProPrintMdlOptionsGet()** to get the initialized model options for the model to be printed. The available model options are:

- *mdl*—Specifies the model to be printed.
- *quality*—Determines the quality of the model to be printed. It checks for no line, no overlap, simple overlap, and complex overlap.

The model options specific to drawing objects are:

- *use_drawing_size*—Overrides the size of the paper specified in the Printer options.
- *draw_format*—Prints the drawing format used.
- *segmented*—If true, that is the value is set to a boolean of 1, the printer prints drawing full size, but in segments that are compatible with the selected paper size. This option is available only if you are plotting a single page.
- *layer_only*—Prints the specified layer only.
- *layer_name*—Prints the name of the layer.
- *sheets*—Prints the current sheet, all sheets, or selected sheets.

- *range*—An array of two integers specifying the start and end sheet numbers.

The model option specific to solid objects is:

- *use_solid_scale*—Prints using the scale used in the solid model.

Use the function **ProPrintPlacementOptionsGet()** to get the current print placement options such as print scale, offset, zoom, and so on. The options available for the object placement are:

- Placement Options:
 - *scale*—Specifies the scale used for the selected plot.
 - *offset*—An array of two doubles representing the offset from the lower-left corner of the plot.
 - *keep_panzoom*—Maintains the pan and zoom values of a window.
- Clipping Options:
 - *clip_plot*—Specifies whether you want to clip the plot.
 - *shift_to_ll_corner*—Shifts the clip area to the lower-left corner of the plot
 - *clip_area*—Two dimensional array of four double representing the area that is clipped. The range of the values of this option is 0.0 through 1.0.
- Label Options:
 - *place_label*—Specifies whether you want to place the label on the plot.
 - *label_height*—Height of the label in inches.

Use the function **ProPrintPCFOptionsGet()** to get the print options from a specified Plotter Configuration File. Specify the name of the plotter configuration file and the name of the model to be printed. The function gets the printer options, model options and placement options.

Use the function **ProPrintExecute()** to print a Pro/ENGINEER window using the specified printer options, model options and placement options. The drawing must be displayed in a window to be successfully printed.

Automatic Printing of 3D Models

Pro/TOOLKIT provides the capability of automatically creating and plotting a drawing of a solid model. The Pro/TOOLKIT application needs only to supply instructions for the "quick print" activity, and Pro/ENGINEER will automatically create the drawing, print it, and then discard it.

Functions Introduced:

- **ProQuickprintoptionsAlloc()**
- **ProQuickprintoptionsFree()**
- **ProQuickprintoptionsLayouttypeSet()**
- **ProQuickprintoptionsOrientationSet()**
- **ProQuickprintoptionsSizeSet()**
- **ProQuickprintoptionsViewAdd()**
- **ProQuickprintoptionsThreeviewlayoutSet()**
- **ProQuickprintoptionsProjectionsSet()**
- **ProQuickprintoptionsTemplateSet()**
- **ProQuickprintoptionsPrintFlatToScreenAnnotsSet()**
- **ProQuickprintExecute()**

The function **ProQuickprintoptionsAlloc()** allocates a quick print options handle.

Use the function **ProQuickprintoptionsFree()** to free a quick print options handle.

Use the function **ProQuickprintoptionsLayouttypeSet()** to assign the layout type for the quick print. You can either specify a drawing layout using the instructions or use a template to define the drawing. The following are the available layout types:

- **PRO_QPRINT_LAYOUT_PROJ**—Use a projected view-type layout.
- **PRO_QPRINT_LAYOUT_MANUAL**—Use a manually arranged layout.
- **PRO_QPRINT_LAYOUT_TEMPLATE**—Use a drawing template to define the layout. If this option is used, only the template name is needed to define the print; other options are not used.

Use the function **ProQuickprintoptionsOrientationSet()** to assign the sheet orientation for the quick print operation. The following are the available sheet orientation types:

- PRODEV_ORIENTATION_PORTRAIT
- PRODEV_ORIENTATION_LANDSCAPE

Use the function **ProQuickprintoptionsSizeSet()** to assign the size of the print for the quick print operation. *PlotPaperSize* specifies the paper size and can be any of the following types:

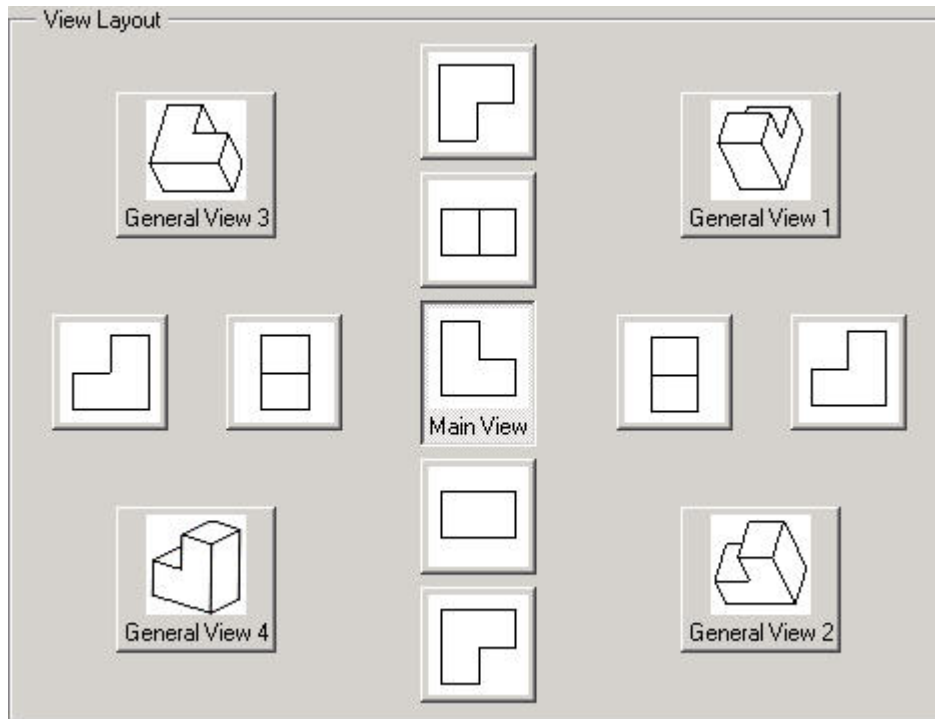
- A_SIZE_PLOT
- B_SIZE_PLOT
- C_SIZE_PLOT
- D_SIZE_PLOT
- E_SIZE_PLOT
- A4_SIZE_PLOT
- A3_SIZE_PLOT
- A2_SIZE_PLOT
- A1_SIZE_PLOT
- A0_SIZE_PLOT
- F_SIZE_PLOT

Note: Variable size plots are not supported by this utility.

Use the function **ProQuickprintoptionsViewAdd()** to add a new general view. The input arguments of this function are:

- *options*—Specifies the options handle.
- *location*—Specifies the location of the view being added for projected view layout. This option is ignored for a manual view layout. It can be of the following types:
 - PRO_QPRINTPROJ_GENVIEW_MAIN
 - PRO_QPRINTPROJ_GENVIEW_NW
 - PRO_QPRINTPROJ_GENVIEW_SW
 - PRO_QPRINTPROJ_GENVIEW_SE
 - PRO_QPRINTPROJ_GENVIEW_NE

The general view location options are analogous to the locations in the quick print user_interface:



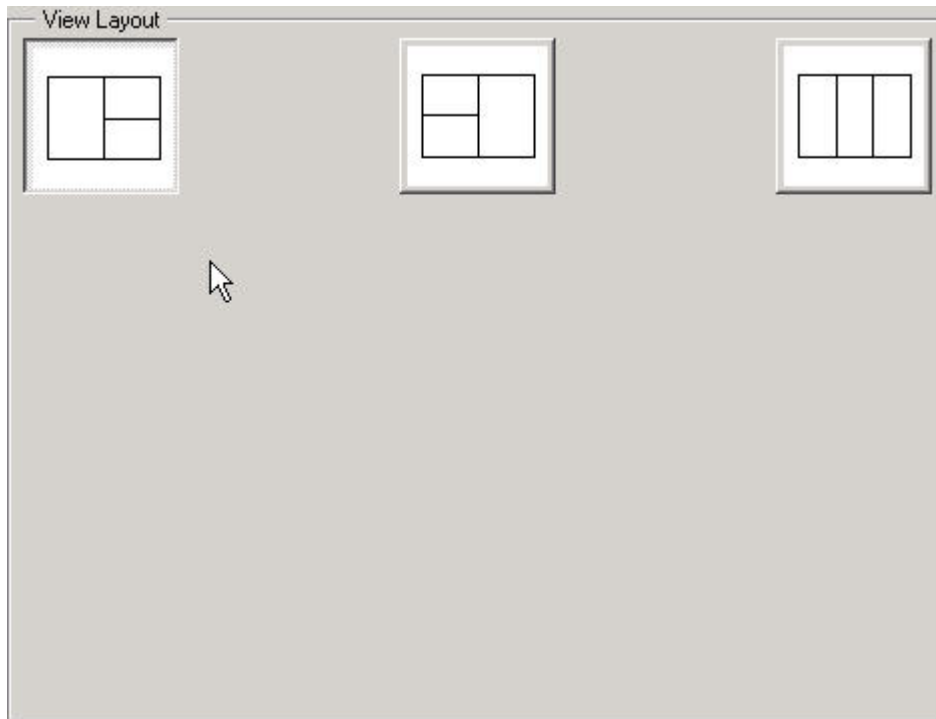
Note: For manual view layouts, the order in which the views are added determine their final location in the drawing. For this configuration, PTC does not support using more than four views.

- *view_name*—Specifies the name of the saved model view.
- *scale*—Specifies the view scale.
- *display_style*—Specifies the view display style to use and is of the following types:
 - PRO_DISPSTYLE_DEFAULT
 - PRO_DISPSTYLE_WIREFRAME
 - PRO_DISPSTYLE_HIDDEN_LINE
 - PRO_DISPSTYLE_NO_HIDDEN

Use the function **ProQuickprintoptionsThreeviewlayoutSet()** to assign the layout type when three views are being used in a manual layout (PRO_QPRINT_LAYOUT_MANUAL). The layout can be either of the following types:

- PRO_QPRINTMANUAL_3VIEW_1_23VERT
- PRO_QPRINTMANUAL_3VIEW_23_VERT1
- PRO_QPRINTMANUAL_3VIEW_123_HORIZ

These options correspond to the diagrams in the user interface:



Interface: Data
Exchange

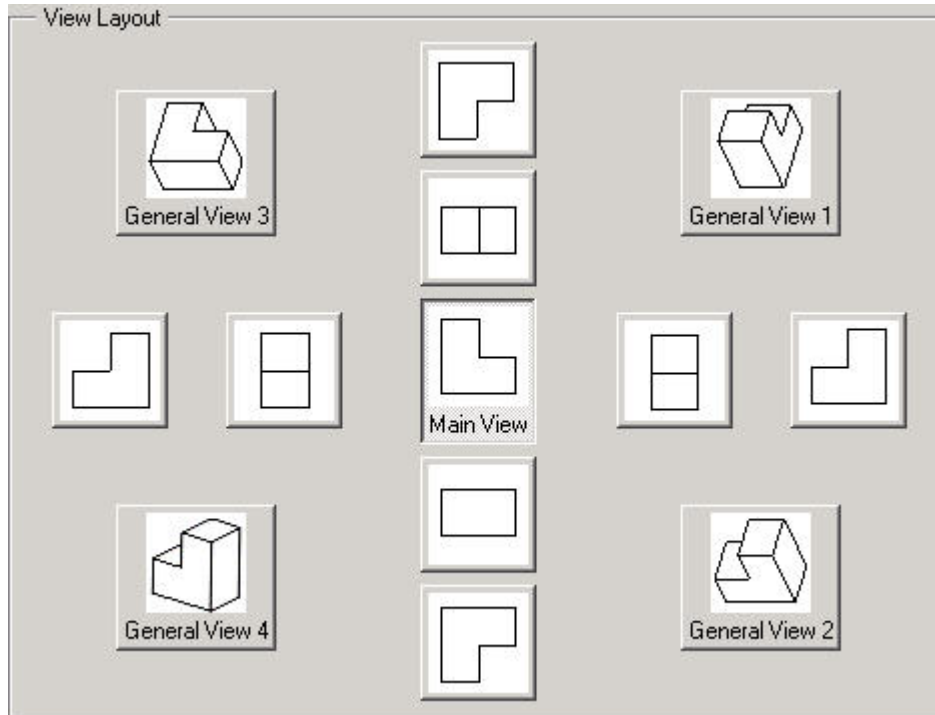
Use the function **ProQuickprintoptionsProjectionsSet()** to assign the projected views to be included in the quick print operation. The function applies only to PRO_QPRINT_LAYOUT_PROJ. The projections are of the following types:

- PRO_QPRINTPROJ_TOP_VIEW
- PRO_QPRINTPROJ_RIGHT_VIEW
- PRO_QPRINTPROJ_LEFT_VIEW
- PRO_QPRINTPROJ_BOTTOM_VIEW
- PRO_QPRINTPROJ_BACK_NORTH
- PRO_QPRINTPROJ_BACK_EAST

- PRO_QPRINTPROJ_BACK_SOUTH
- PRO_QPRINTPROJ_BACK_WEST

Note: Projection views takes the same view scale and display style as the main view.

The options correspond to the projected members of the diagram in the user interface:



Use the function **ProQuickprintoptionsTemplateSet()** to assign the path to the drawing template file to be used for the quick print operation. The function applies only to layout type PRO_QPRINT_LAYOUT_TEMPLATE.

Note: The quick print operation shows the exact image of the model as is shown on-screen. Therefore, if the drawing template has drawing views set with display options such as view clipping, simplified representations, or layers, these settings are ignored while plotting. The resulting plot reflects whatever is seen on-screen.

Use the function

ProQuickprintOptionsPrintFlatToScreenAnnotsSet() to set the *ProBoolean* flag to quick print flat-to-screen annotations. The flat-to-screen annotations created at screen locations in the Pro/ENGINEER graphics window are printed at their relative locations in the drawing. You can quick print only flat-to-screen annotations such as notes, symbols, and surface finish symbols.

Once the instructions have been prepared, use the function **ProQuickprintExecute()** to execute a quick print for a given solid model. It has the following input arguments:

- *solid*—Specifies the solid model to be printed.
- *pcf_path*—Specifies the path to the plotter configuration file to use. If no path is specified, then the path will have the value of the configuration option `quickprint_plotter_config_file`.
- *options*—Specifies the details of the quick print operation given by the *ProQuickprintOptions* handle.

Exporting 3D Models

Pro/TOOLKIT provides export capabilities for three dimensional geometry to various formats.

Functions Introduced:

- **ProIntf3DFileWrite()**
- **ProOutputBrepRepresentationAlloc()**
- **ProOutputBrepRepresentationFlagsSet()**
- **ProOutputBrepRepresentationIsSupported()**
- **ProOutputBrepRepresentationFree()**
- **ProOutputInclusionAlloc()**
- **ProOutputInclusionFacetparamsSet()**
- **ProOutputInclusionWithOptionsSet()**
- **ProOutputInclusionFlagsSet()**
- **ProOutputInclusionFree()**
- **ProOutputLayerOptionsAlloc()**
- **ProOutputLayerOptionsAutoidSet()**
- **ProOutputLayerOptionsSetupfileSet()**

- **ProOutputLayerOptionsFree()**
- **ProOutputAssemblyConfigurationIsSupported()**
- **ProRasterFileWrite()**
- **ProIntfSliceFileExport()**
- **ProIntfSliceFileWithOptionsExport()**
- **ProExportVRML()**
- **ProProductviewexportoptsAlloc()**
- **ProProductviewexportoptsFree()**
- **ProProductviewexportoptsFormatSet()**
- **ProProductviewFormattedExport()**

Export Format	Pro/TOOLKIT Functions	Type Constant
STEP file (Standard for the Exchange of Product Model Data)	ProIntf3DFileWrite()	PRO_INTF_EXPORT_STEP
SET file (Standard for Exchange and Transfer)		PRO_INTF_EXPORT_SET
VDA file		PRO_INTF_EXPORT_VDA
IGES (3D) file		PRO_INTF_EXPORT_IGES
CATIA (.model) file		PRO_INTF_EXPORT_CATIA_MODEL
SAT file (ACIS format for Pro/ENGINEER)		PRO_INTF_EXPORT_SAT
NEUTRAL file (ASCII text)		PRO_INTF_EXPORT_NEUTRAL
CADDS file		PRO_INTF_EXPORT_CADDS
CATIA (.session) file		PRO_INTF_EXPORT_CATIA_SESSION
Parasolid file		PRO_INTF_EXPORT_PARASOLID
UG file		PRO_INTF_EXPORT_UG

Export Format	Pro/TOOLKIT Functions	Type Constant
CATIA V5 Part file	ProIntfSliceFileExport()	PRO_INTF_EXPORT_CATIA_PART
CATIA V5 Assembly file		PRO_INTF_EXPORT_CATIA_PRODUCT
JT Open format		PRO_INTF_EXPORT_JT
CATIA Graphical Representation (CGR) format		PRO_INTF_EXPORT_CATIA_CGR
CATIA facets file		PRO_CATIAFACETS_FILE
INVENTOR file		PRO_INVENTOR_FILE
Render file		PRO_RENDER_FILE
SLA ASCII file		PRO_SLA_ASCII_FILE
SLA Binary file		PRO_SLA_BINARY_FILE
JPEG file	ProRasterFileWrite()	PRORASTERTYPE_JPEG
BMP file		PRORASTERTYPE_BMP
TIFF file		PRORASTERTYPE_TIFF
EPS file (Postscript)		PRORASTERTYPE_EPS
PVS file, OL file (a separate OL file is created for each PART in an assembly)	ProProductviewFormattedExport()	PRO_PV_FORMAT_PVS
ED file, OL file (a separate OL file is created for each PART in an assembly)		PRO_PV_FORMAT_ED
EDZ file		PRO_PV_FORMAT_EDZ
PVZ file		PRO_PV_FORMAT_PVZ
VRML	ProExportVRML	N/A
Shrinkwrap	ProSolidShrinkwrapCreate()	N/A

The function **ProIntf3DFileWrite()** exports a Pro/ENGINEER model to the specified output format. The following types of output formats are supported:

- STEP
- SET
- VDA
- IGES
- CATIA MODEL
- SAT (ACIS format in Pro/ENGINEER)
- NEUTRAL
- CADDs
- CATIA SESSION
- PARASOLID
- UG
- CATIA V5
- JT Open
- CATIA Graphical Representation

While exporting the model you can specify the structure and contents of the output files as:

- Flat File—Exports all of the geometry of the assembly to a single file as if it were a part. This is similar to the Single File format in Pro/ENGINEER for STEP output.
- Single File—Exports an assembly structure to a file with external references to component files. This file contains only top-level geometry. This is similar to the Dittos format in Pro/ENGINEER for CATIA, Separate Parts Only for STEP and One Level for IGES outputs.
- Multi Files—Exports an assembly structure to a single file and the components to component files. It creates component parts and subassemblies with their respective geometry and external references. This option supports all levels of hierarchy. This is similar to All Levels format for IGES and Separate All Parts for STEP in Pro/ENGINEER.
- Parts—Exports an assembly as multiple files containing geometry information of its components and assembly features. This is similar to All Parts format for IGES in Pro/ENGINEER.

Some output formats support only certain types of assembly configurations. The default assembly configuration is a flat file.

Use the function

ProOutputAssemblyConfigurationIsSupported() to check if the specified assembly configuration is valid for the particular model and the specified export format. This function must be called before exporting the model to the specified output format using the function **ProIntf3DFileWrite()** except for the CADDs and STEP2D formats.

The function **ProOutputBrepRepresentationAlloc()** allocates memory for the geometric representation data structure. This data structure represents the types of geometry supported by the export operation. The types of geometric representations are:

- Wireframe
- Surfaces
- Solid
- Quilts (Shell in Pro/ENGINEER)

These correspond to the options shown in the Pro/ENGINEER dialog box for export. Note that some formats allow a combination of types to be input.

The function **ProOutputBrepRepresentationFlagsSet()** sets the flags for the geometric representation data structure. It specifies the type of geometry to be exported.

The function **ProOutputBrepRepresentationIsSupported()** checks if the specified geometric representation is valid for a particular export format. This function should be called before exporting the model to the specified output format using the function **ProIntf3DFileWrite()**, to check if the planned configuration is supported by the Pro/ENGINEER interface options.

The function **ProOutputBrepRepresentationFree()** frees the memory allocated for the geometry data structure.

The function **ProOutputInclusionAlloc()** allocates memory for the inclusion structure to be used while exporting the model.

The function **ProOutputInclusionFlagsSet()** determines whether to include certain entities during export. The types of entities are:

- Datums—Determines whether datum curves are included when exporting files. If the flag `include_datums` is set to true the datum curve and point information is included during export. The default value is false.

- **Blanked**—Determines whether entities on blanked layers are exported. If the flag `include_blanked` is set to true, entities on blanked layers are exported. The default value is false.
- **Facets**—Determines whether faceted geometry is included when exporting the models. The default value of the flag `include_facetted` is false.

The function **ProOutputInclusionFacetparamsSet()** assigns the parameters to use while exporting the model to a faceted format such as `PRO_INTF_EXPORT_CATIA_CGR`. These parameters are as follows:

- *chord_height*—The chord height to use for the exported facets.
- *angle_control*—The angle control to use for the exported facets.

Note: The function **ProOutputInclusionFacetparamsSet()** has been deprecated. Use the function **ProOutputInclusionWithOptionsSet()** instead.

Use the function **ProOutputInclusionWithOptionsSet()** to set the parameters and configuration flags used while exporting the model to a faceted format such as `PRO_INTF_EXPORT_CATIA_CGR`. The input arguments are as follows:

- *parameters*—Specifies a *ProArray* of parameters that consists of the following three elements:
 - *chord_height*—The chord height of the exported facets.
 - *angle_control*—The angle control of the exported facets. Specify a value between 0.0 to 1.0. If the angle control is out of bounds, Pro/ENGINEER changes it to the closest limit without returning an error.
 - *step_size*—The step size of the exported facets. If the step size is less or equal to 0, it is ignored.

Note: If the chord height or step size are too small or too big, then Pro/ENGINEER resets it to the smallest or biggest acceptable value, respectively, without returning an error.
- *config_flags*—Specifies the configuration flags that control the export operation. They are as follows:
 - `PRO_FACET_STEP_SIZE_OFF`—Switches off the step size control.
 - `PRO_FACET_STEP_SIZE_ADJUST`—Adjusts the step size according to the component size.

- **PRO_FACET_CHORD_HEIGHT_ADJUST**—Adjusts the chord height according to the component size.
- **PRO_FACET_USE_CONFIG**—If this flag is set, values of the flags **PRO_FACET_STEP_SIZE_OFF**, **PRO_FACET_STEP_SIZE_ADJUST**, and **PRO_FACET_CHORD_HEIGHT_ADJUST** are ignored and the configuration settings from the Pro/ENGINEER user interface are used during the export operation.
- **PRO_FACET_CHORD_HEIGHT_DEFAULT**—Uses the default value set in the Pro/ENGINEER user interface for the chord height.
- **PRO_FACET_ANGLE_CONTROL_DEFAULT**—Uses the default value set in the Pro/ENGINEER user interface for the angle control.
- **PRO_FACET_STEP_SIZE_DEFAULT**—Uses the default value set in the Pro/ENGINEER user interface for the step size.

Note: The behavior of the function **ProOutputInclusionWithOptionsSet()** is similar to the function **ProOutputInclusionFacetparamsSet()** only if the configuration flag **PRO_FACET_STEP_SIZE_OFF** is set.

The function **ProOutputInclusionFree()** frees the memory allocated for the inclusion structure.

The function **ProOutputLayerOptionsAlloc()** allocates memory for the layer options data structure. The layer options are:

- **AutoId**—A flag indicating whether layers should be automatically assigned numerical ids when exporting.
- **LayerSetupFile**—The layer setup file contains the name of the layer, its display status, the interface ID and number of sub layers.

Specify the name and complete path of the layer setup file. This file contains the layer assignment information.

The function **ProOutputLayerOptionsAutoidSet()** enables you to set or remove an interface layer ID. If true, automatically assigns interface ids to layers not assigned ids and exports them. The default value is false.

Use the function **ProOutputLayerOptionsSetupfileSet()** to specify the name and complete path of the layer setup file.

The function **ProOutputLayerOptionsFree()** frees the memory allocated for the layer options structure.

Use function **ProRasterFileWrite()** to create a standard Pro/ENGINEER raster output file. Note that this function does not support output of drawings (2-dimensional objects) in Drawing mode.

The function **ProIntfSliceFileExport()** exports 5 of the 7 “tesellated” formats (STL, Render, Inventor, CatiaFacets, Optegra Visualizer, and RCS.) Note that *3dPaint* requires additional input and is not handled by this function. These formats require the maximum chord height, angle control, and transformation to be specified for the model being exported. If the specified model is an assembly, the last input argument of the function is the component path; if the model is a part, this argument is NULL.

Note: The function **ProIntfSliceFileExport()** has been deprecated. Use the function **ProIntfSliceFileWithOptionsExport()** instead.

The function **ProIntfSliceFileWithOptionsExport()** exports to “tesellated” formats such as STL, Render, Inventor, CatiaFacets, and Optegra Visualizer based on the values of a *ProArray* of parameters and two configuration flags. These parameters and configuration flags are same as the ones assigned by the function **ProOutputInclusionWithOptionsSet()** described earlier in this section. Refer to its description for more information on the parameters and configuration flags.

Note: The behavior of the function **ProIntfSliceFileWithOptionsExport()** is similar to the function **ProIntfSliceFileExport()** if the configuration flag PRO_FACET_STEP_SIZE_OFF is set.

The function **ProExportVRML()** exports a solid from a Pro/ENGINEER session, or a Pro/ENGINEER solid stored in a file, into a directory of VRML files. This output directory contains assembly structure data, part and assembly names, and geometrical data representing the parts. This function accepts as input only Pro/ENGINEER assemblies or parts.

ProExportVRML() supports creation of multiple output files from either parts or assemblies. If you export an assembly, the function creates an output file for each member of the assembly and one for the assembly itself. Default file names are:

asm1_a.wrl, asm2_a.wrl, ... asmN_a.wrl

where *asm* is the assembly name.

If you export parts, **ProExportVRML()** creates an output file for each part. Default names are *part_p.wrl*, where *part* is the part name. For more information on **ProExportVRML()**, refer to “Exporting Files to VRML” or “Batch Utilities” in the “Interface” section of Pro/ENGINEER help.

The functions **ProProductviewexportoptsAlloc()** and **ProProductviewexportoptsFree()** allocate and free the memory assigned to the *ProProductviewExportOptions* object containing the ProductView export formats.

The function **ProProductviewexportoptsFormatSet()** assigns the flag specifying the ProductView export format.

The function **ProProductviewFormattedExport()** exports a part or an assembly to one of the following user-defined ProductView formats.

- PRO_PV_FORMAT_PVS
- PRO_PV_FORMAT_ED
- PRO_PV_FORMAT_EDZ
- PRO_PV_FORMAT_PVZ

Example 2: To Export a Model File to IGES Format

This example exports a model file to IGES format using options similar to those seen in the UI.

```
#include <ProToolkit.h>
#include <ProMenu.h>
#include <ProMdl.h>
#include <ProIntf3Dexport.h>
#include <ProShrinkwrap.h>
#include <TestError.h>

/*=====*\
FUNCTION: UserSolidIGESExportWithFlags()
PURPOSE: Exports a model to IGES
\*=====*/

int UserSolidIGESExportWithFlags(ProSolid solid,
                                ProSelection reference_csys,
                                ProPath output_file)
{
    ProError status;
    ProIntf3DexportType output_format = PRO_INTF_EXPORT_IGES;
    ProOutputAssemblyConfiguration asm_config =
        PRO_OUTPUT_ASSEMBLY_MULTI_FILES;
    ProOutputBrepRepresentation brep_flags;
```

```

ProOutputInclusion inclusion;
ProOutputLayerOptions layer_options;
ProBoolean is_supported;

/*-----*\
Verify that this is a permitted assembly configuration option
\*-----*/
status = ProOutputAssemblyConfigurationIsSupported (output_format,
                                                    asm_config,
                                                    &is_supported);

if (status != PRO_TK_NO_ERROR || !is_supported)
{
    ProMessageDisplay (MSGFIL,
                      "USER Assembly configuration is not supported!");
    return PRO_TK_BAD_INPUTS;
}

/*-----*\
Preallocate geometry flags for export
\*-----*/
status = ProOutputBrepRepresentationAlloc (&brep_flags);

status = ProOutputBrepRepresentationFlagsSet (brep_flags,
                                              PRO_B_FALSE,
                                              PRO_B_TRUE,
                                              PRO_B_FALSE,
                                              PRO_B_FALSE);

/*-----*\
Verify that these flags are supported for this export type
\*-----*/
status = ProOutputBrepRepresentationIsSupported (output_format,
                                                  brep_flags,
                                                  &is_supported);

if (status != PRO_TK_NO_ERROR || !is_supported)
{
    ProMessageDisplay (MSGFIL,
                      "USER B-rep flag combination is not supported!");
    ProOutputBrepRepresentationFree (brep_flags);

    return PRO_TK_BAD_INPUTS;
}

/*-----*\
Allocate inclusion flags and layer options
\*-----*/
status = ProOutputInclusionAlloc (&inclusion);

status = ProOutputInclusionFlagsSet (inclusion,
                                   PRO_B_TRUE,
                                   PRO_B_FALSE,
                                   PRO_B_FALSE);

```

```

    status = ProOutputLayerOptionsAlloc (&layer_options);

    status = ProOutputLayerOptionsAutoidSet (layer_options, PRO_B_TRUE);

/*-----*\
Export IGES file(s)
\*-----*/
    ProIntf3DFileWrite (solid, output_format, output_file,
        asm_config, reference_csys, brep_flags,
        inclusion, layer_options);

/*-----*\
Free input argument memory
\*-----*/
    ProOutputBrepRepresentationFree (brep_flags);
    ProOutputLayerOptionsFree (layer_options);
    ProOutputInclusionFree (inclusion);

    return PRO_TK_NO_ERROR;
}

```

Shrinkwrap Export

To improve performance in large assembly design, you can export lightweight representations of models called Shrinkwrap models. A shrinkwrap model is based on the external surfaces of the source part or assembly model and captures the outer shape of the source model.

You can create the following types of non associative exported Shrinkwrap models:

- **Surface Subset**—This type consists of a subset of the original model's surfaces.
- **Faceted Solid**—This type is a faceted solid representing the original solid.
- **Merged Solid**—The external components from the reference assembly model are merged into a single part representing the solid geometry in all collected components.

Export Format	Pro/TOOLKIT Functions	Type Constant
Shrinkwrap	ProSolidShrinkwrapCreate()	N/A

Function Introduced:

- **ProSolidShrinkwrapCreate()**

You can export the specified solid model as a Shrinkwrap model using the function **ProSolidShrinkwrapCreate()**. This function requires:

- The model to be exported as shrinkwrap
- The template model where the Shrinkwrap geometry will be created.
- The name of the exported file if the export format is VRML or STL.

Setting Shrinkwrap Options

Functions Introduced:

- **ProShrinkwrapoptionsAlloc()**
- **ProShrinkwrapoptionsFree()**
- **ProShrinkwrapoptionsQualitySet()**
- **ProShrinkwrapoptionsAutoholefillingSet()**
- **ProShrinkwrapoptionsIgnoreskeletonsSet()**
- **ProShrinkwrapoptionsIgnorequiltsSet()**
- **ProShrinkwrapoptionsAssignmasspropsSet()**
- **ProShrinkwrapoptionsDatumrefsSet()**

The function **ProShrinkwrapoptionsAlloc()** allocates memory for the structure defining the shrinkwrap options. The types of shrinkwrap methods are:

- PRO_SWCREATE_SURF_SUBSET—Surface Subset
- PRO_SWCREATE_FACETED_SOLID—Faceted Solid
- PRO_SWCREATE_MERGED_SOLID—Merged Solid

The function returns the options handle which is used to set the members of the structure defining the shrinkwrap options.

The function **ProShrinkwrapoptionsFree()** frees the memory allocated by the function **ProShrinkwrapoptionsAlloc()**.

The function **ProShrinkwrapoptionsQualitySet()** specifies the quality level for the system to use when identifying surfaces or components that will contribute to the Shrinkwrap model. Quality ranges from 1 which produces the coarsest representation of the model in the fastest time, to 10 which produces the most exact representation. The default value is true.

The function **ProShrinkwrapoptionsAutoholefillingSet()** sets a flag that forces Pro/ENGINEER to identify all holes and surfaces that intersect a single surface and fills those holes during shrinkwrap. The default value is true.

The function **ProShrinkwrapoptionsIgnoreskeletonsSet()** determines whether the skeleton model geometry must be included in the Shrinkwrap model.

ProShrinkwrapoptionsIgnorequiltsSet() determines whether external quilts will be included in the Shrinkwrap model.

ProShrinkwrapoptionsAssignmasspropsSet() assigns mass properties to the Shrinkwrap model. The default value is false and the mass properties of the original model is assigned to the Shrinkwrap model. If the value is set to true, the user will have to assign a value for the mass properties.

ProShrinkwrapoptionsDatumrefsSet() selects the datum planes, points, curves, axes, and coordinate system references to be included from the Shrinkwrap model.

Surface Subset Options

Functions Introduced:

- **ProShrinkwrapoptionsIgnoresmallsurfsSet()**
- **ProShrinkwrapoptionsAdditional SurfacesSet()**

The function **ProShrinkwrapoptionsIgnoresmallsurfsSet()** sets a flag that forces Pro/ENGINEER to skip surfaces smaller than a certain size. The default value of this argument is false. The size of the surface is specified as a percentage of the model's size.

Use **ProShrinkwrapoptionsAdditional SurfacesSet()** to select individual surfaces to be included in the Shrinkwrap model.

Faceted Solid Options

Functions Introduced:

- **ProShrinkwrapoptionsFacetedformatSet()**
- **ProShrinkwrapoptionsFramesFileSet()**

Use the function **ProShrinkwrapoptionsFacetedformatSet()** to specify the output file format of the Shrinkwrap model. The types of output format are:

- PRO_SWFACETED_PART—Pro/ENGINEER part with normal geometry. This is the default format type.
- PRO_SWFACETED_LIGHTWEIGHT_PART—Lightweight Pro/ENGINEER part with lightweight, faceted geometry.
- PRO_SWFACETED_STL—An STL file
- PRO_SWFACETED_VRML—A VRML file

The function **ProShrinkwrapoptionsFramesFileSet()** enables you to select a frame file to create a faceted solid motion envelope model that represents the full motion of the mechanism captured in the frame file. Specify the name and complete path of the frame file.

Merged Solid Options

Function Introduced:

- **ProShrinkwrapoptionsAdditionalcomponentsSet()**

Use the function **ProShrinkwrapoptionsAdditionalcomponentsSet()** to select individual components of the assembly to be merged into the Shrinkwrap model.

Example 3: To Export a Model to VRML Format

This example creates a faceted shrinkwrap model in VRML format

```
#include <ProToolkit.h>
#include <ProMenu.h>
#include <ProMdl.h>
#include <ProShrinkwrap.h>

#include <TestError.h>

/*****
FUNCTION: UserShrinkwrapExportToVRML()
PURPOSE: Exports a model to VRML shrinkwrap
*****/
```

```

int UserShrinkwrapExportToVRML(ProSolid solid,
                               ProName output_file_name)
{
    ProError status;
    ProShrinkwrapOptions options;

    /*-----*\
    Preallocate needed input arguments
    \*-----*/
    status = ProShrinkwrapoptionsAlloc (PRO_SWCREATE_FACETED_SOLID,
                                       &options);

    status = ProShrinkwrapoptionsAutoholefillingSet (options,
                                                    PRO_B_FALSE);

    status = ProShrinkwrapoptionsFacetedformatSet (options,
                                                  PRO_SWFACETED_VRML);

    /*-----*\
    Export shrinkwrap file
    \*-----*/
    status = ProSolidShrinkwrapCreate (solid, NULL, output_file_name,
                                       options);

    /*-----*\
    Free input argument memory
    \*-----*/
    ProShrinkwrapoptionsFree (options);

    return PRO_TK_NO_ERROR;
}

```

Example 4: To Create a Shrinkwrap Part Model as a Merged Solid

This example demonstrates how to create a new empty model and copy the merged solid shrinkwrap information into it.

```

#include <ProToolkit.h>
#include <ProMenu.h>
#include <ProMdl.h>
#include <ProIntf3Dexport.h>
#include <ProShrinkwrap.h>
#include <ProFaminstance.h>
#include <ProSolid.h>
#include <ProToolkitDll.h>
#include <ProSimprep.h>
#include <ProDrawing.h>
#include <ProUtil.h>

```

```

#include <ProModelitem.h>
#include <ProWstring.h>
#include <ProTKRunTime.h>
#include <ProMessage.h>
#include <UtilString.h>

#include <TestError.h>

/*=====*\
FUNCTION: UserShrinkwrapCreateInNewModel()
PURPOSE:  Creates a new Shrinkwrap part model as a merged solid.
\*=====*/

int UserShrinkwrapCreateInNewModel(ProSolid solid,
                                   ProName new_model_name)
{
    ProError status;
    ProShrinkwrapOptions options;
    ProSolid new_model;

/*-----*\
Preallocate needed input arguments
\*-----*/
    status = ProShrinkwrapoptionsAlloc (PRO_SWCREATE_MERGED_SOLID,
                                       &options);

    status = ProShrinkwrapoptionsAssignmasspropsSet (options,
                                                    PRO_B_TRUE);

/*-----*\
Export shrinkwrap file
\*-----*/
    status = ProSolidCreate (new_model_name, (ProType) PRO_MDL_PART,
&new_model);

    status = ProSolidShrinkwrapCreate (solid, new_model, NULL, options);

/*-----*\
Free input argument memory
\*-----*/
    ProShrinkwrapoptionsFree (options);

    return PRO_TK_NO_ERROR;
}
#endif MSGFIL

```


Exporting to PDF and U3D

The functions described in this section support the export of Pro/ENGINEER drawings and solid models to Portable Document Format (PDF) and U3D format. You can export a drawing or a 2D model as a 2D raster image embedded in a PDF file. You can export Pro/ENGINEER solid models in the following ways:

- As a U3D model embedded in a one-page PDF file
- As 2D raster images embedded in the pages of a PDF file representing saved views
- As a standalone U3D file

While exporting multiple sheets of a Pro/ENGINEER drawing to a PDF file, you can choose to export all sheets, the current sheet, or selected sheets.

These functions also allow you to insert a variety of non-geometric information to improve document content, navigation, and search.

Functions Introduced:

- **ProPDFOptionsAlloc()**
- **ProPDFOptionsIntpropertySet()**
- **ProPDFOptionsBoolpropertySet()**
- **ProPDFOptionsStringpropertySet()**
- **ProPDFOptionsDoublepropertySet()**
- **ProPDFExport()**
- **ProPDFWithProfileExport()**
- **ProPDFOptionsFree()**

The function **ProPDFOptionsAlloc()** allocates memory for the PDF options structure.

The function **ProPDFOptionsIntpropertySet()** sets the value for an integer or enum property of the PDF options structure. The types of export options are as follows:

- **PRO_PDFOPT_EXPORT_MODE**—Enables you to select the object to be exported to PDF and the export format. The values are:
 - **PRO_PDF_2D_DRAWING**—Specifies that only 2D drawings will be exported to PDF. This is the default.

- PRO_PDF_3D_AS_NAMED_VIEWS—Specifies that Pro/ENGINEER models will be exported as 2D raster images embedded in PDF files.
- PRO_PDF_3D_AS_U3D_PDF—Specifies that Pro/ENGINEER models will be exported as U3D models embedded in one-page PDF files.
- PRO_PDF_3D_AS_U3D—Specifies that a Pro/ENGINEER model will be exported as a U3D (*.u3d) file. This value ignores the options available for the PDF options structure.
- PRO_PDFOPT_PDF_SAVE—Enables you to specify the PDF format while exporting 2D drawings and solid models. The values are:
 - PRO_PDF_ARCHIVE_1—Exports the 2D drawing to the PDF/A format. This type is applicable only for 2D drawings, that is, when you set the option PRO_PDFOPT_EXPORT_MODE to PRO_PDF_2D_DRAWING.

If you set the option PRO_PDFOPT_PDF_SAVE to PRO_PDF_ARCHIVE_1, the following options are set as shown below:

 - PRO_PDFOPT_LAYER_MODE is set to PRO_PDF_LAYERS_NONE.
 - PRO_PDFOPT_HYPERLINKS is set to FALSE, that is, hyperlinks are not created in the PDF.
 - Shaded views will not be transparent and may overlap other data.
 - PRO_PDFOPT_PASSWORD_TO_OPEN is set to NULL.
 - PRO_PDFOPT_MASTER_PASSWORD is set to NULL.- PRO_PDF_FULL—Exports the object to the standard PDF format. This is the default value.
- PRO_PDFOPT_FONT_STROKE—Enables you to switch between using TrueType fonts in the resulting document and drawing or "stroking" text as line segments. The values are:
 - PRO_PDF_USE_TRUE_TYPE_FONTS—Specifies TrueType fonts. This is the default.
 - PRO_PDF_STROKE_ALL_FONTS—Specifies the option to stroke all fonts.
- PRO_PDFOPT_COLOR_DEPTH—Enables you to choose between color, grayscale, or monochrome output. The values are:

- PRO_PDF_CD_COLOR—Specifies color output. This is the default.
- PRO_PDF_CD_GRAY—Specifies grayscale output.
- PRO_PDF_CD_MONO—Specifies monochrome output.
- PRO_PDFOPT_HIDDENLINE_MODE—Enables you to set the style for hidden lines in the resulting PDF document. The values are:
 - PRO_PDF_HLM_DASHED—Specifies dashed hidden lines. This is the default.
 - PRO_PDF_HLM_SOLID—Specifies solid hidden lines.
- PRO_PDFOPT_RASTER_DPI—Enables you to set the resolution for the output of any shaded views in DPI. The value is restricted to the values in *ProDotsPerInch*, and the default is PRORASTERDPI_300.
- PRO_PDFOPT_LAYER_MODE—Enables you to set the availability of layers in the document. The values are:
 - PRO_PDF_LAYERS_VISIBLE—Exports only visible layers in a drawing.
 - PRO_PDF_LAYERS_NONE—Exports only the visible entities in the drawing, but not the layers on which they are placed.
 - PRO_PDF_LAYERS_ALL—Exports the visible layers and entities. This is the default.
- PRO_PDFOPT_PARAM_MODE—Enables you to set the availability of model parameters as searchable metadata in the PDF document. The values are:
 - PRO_PDF_PARAMS_DESIGNATED—Exports only the specified model parameters in the PDF metadata.
 - PRO_PDF_PARAMS_NONE—Exports the drawing to PDF without the model parameters.
 - PRO_PDF_PARAMS_ALL—Exports the drawing and the model parameters to PDF. This is the default.
- PRO_PDFOPT_ALLOW_MODE—Enables you to set the security settings for the PDF document. This option must be set if PRO_PDFOPT_RESTRICT_OPERATIONS is set to true. The values are:
 - PRO_PDF_RESTRICT_NONE—Specifies that the user can perform any of the permitted viewer operations on the PDF document. This is the default.

- PRO_PDF_RESTRICT_FORMS_SIGNING—Restricts the user from adding digital signatures to the PDF document.
- PRO_PDF_RESTRICT_INSERT_DELETE_ROTATE—Restricts the user from inserting, deleting, or rotating the pages in the PDF document.
- PRO_PDF_RESTRICT_COMMENT_FORM_SIGNING—Restricts the user from adding or editing comments in the PDF document.
- PRO_PDF_RESTRICT_EXTRACTING—Restricts the user from extracting pages from the PDF document.
- PRO_PDFOPT_ALLOW_PRINTING_MODE—Enables you to set the print resolution. The values are:
 - PRO_PDF_PRINTING_LOW_RES—Specifies low resolution for printing.
 - PRO_PDF_PRINTING_HIGH_RES—Specifies high resolution for printing. This is the default.
- PRO_PDFOPT_LINECAP—Enables you to control the treatment of the ends of the geometry lines exported to PDF. The values are:
 - PRO_PDF_LINECAP_BUTT—Specifies the butt cap square end. This is the default.
 - PRO_PDF_LINECAP_ROUND—Specifies the round cap end.
 - PRO_PDF_LINECAP_PROJECTING_SQUARE—Specifies the projecting square cap end.
- PRO_PDFOPT_LINEJOIN—Enables you to control the treatment of the joined corners of connected lines exported to PDF. The values are:
 - PRO_PDF_LINEJOIN_MITER—Specifies the miter join. This is the default.
 - PRO_PDF_LINEJOIN_ROUND—Specifies the round join.
 - PRO_PDF_LINEJOIN_BEVEL—Specifies the bevel join.
- PRO_PDFOPT_SHEETS—Enables you to specify the sheets from a Pro/ENGINEER drawing that are to be exported to PDF. The values are:
 - PRINT_CURRENT_SHEET—Specifies that only the current sheet will be exported to the PDF file.
 - PRINT_ALL_SHEETS—Specifies that all the sheets will be exported to the PDF file. This is the default.

- PRINT_SELECTED_SHEETS—Specifies that sheets of a specified range will be exported to the PDF file. If this value is assigned, then the value of the string property PRO_PDFOPT_SHEET_RANGE must also be included.
- PRO_PDFOPT_LIGHT_DEFAULT—Enables you to set the default lighting style used while exporting Pro/ENGINEER models in the U3D format to a one-page PDF file. The values are:
 - PRO_PDF_U3D_LIGHT_NONE—Specifies no lights.
 - PRO_PDF_U3D_LIGHT_WHITE—Specifies white lights.
 - PRO_PDF_U3D_LIGHT_DAY—Specifies day lights.
 - PRO_PDF_U3D_LIGHT_BRIGHT— Specifies bright lights.
 - PRO_PDF_U3D_LIGHT_PRIMARY—Specifies primary color lights.
 - PRO_PDF_U3D_LIGHT_NIGHT—Specifies night lights.
 - PRO_PDF_U3D_LIGHT_BLUE—Specifies blue lights.
 - PRO_PDF_U3D_LIGHT_RED—Specifies red lights.
 - PRO_PDF_U3D_LIGHT_CUBE—Specifies cube lights.
 - PRO_PDF_U3D_LIGHT_CAD—Specifies CAD optimized lights. This is the default value.
 - PRO_PDF_U3D_LIGHT_HEADLAMP—Specifies headlamp lights.
- PRO_PDFOPT_RENDER_STYLE_DEFAULT—Enables you to set the default rendering style used while exporting Pro/ENGINEER models in the U3D format to a one-page PDF file. The values are:
 - PRO_PDF_U3D_RENDER_BOUNDING_BOX—Specifies bounding box rendering.
 - PRO_PDF_U3D_RENDER_TRANSPARENT_BOUNDING_BOX—Specifies transparent bounding box rendering.
 - PRO_PDF_U3D_RENDER_TRANSPARENT_BOUNDING_BOX_OUTLINE—Specifies transparent bounding box outline rendering.
 - PRO_PDF_U3D_RENDER_VERTICES—Specifies vertices rendering.
 - PRO_PDF_U3D_RENDER_SHADED_VERTICES—Specifies shaded vertices rendering.

- PRO_PDF_U3D_RENDER_WIREFRAME—Specifies wireframe rendering.
- PRO_PDF_U3D_RENDER_SHADED_WIREFRAME—Specifies shaded wireframe rendering.
- PRO_PDF_U3D_RENDER_SOLID—Specifies solid rendering. This is the default.
- PRO_PDF_U3D_RENDER_TRANSPARENT—Specifies transparent rendering.
- PRO_PDF_U3D_RENDER_SOLID_WIREFRAME—Specifies solid wireframe rendering.
- PRO_PDF_U3D_RENDER_TRANSPARENT_WIREFRAME—Specifies transparent wireframe rendering.
- PRO_PDF_U3D_RENDER_ILLUSTRATION—Specifies illustrated rendering.
- PRO_PDF_U3D_RENDER_SOLID_OUTLINE—Specifies solid outlined rendering.
- PRO_PDF_U3D_RENDER_SHADED_ILLUSTRATION—Specifies shaded illustrated rendering.
- PRO_PDF_U3D_RENDER_HIDDEN_WIREFRAME—Specifies hidden wireframe rendering.
- PRO_PDFOPT_SIZE—Enables you to specify the page size of the exported PDF file. The values are restricted to the value of *ProPlotPaperSize*. If the value is VARIABLE_SIZE_IN_MM_PLOT or VARIABLE_SIZE_PLOT, the size must be specified in the PRO_PDFOPT_HEIGHT and PRO_PDFOPT_WIDTH properties.
- PRO_PDFOPT_ORIENTATION—Enables you to specify the orientation of the pages in the exported PDF file. The values are:
 - PRO_ORIENTATION_PORTRAIT—Exports the pages in portrait orientation. This is the default.
 - PRO_ORIENTATION_LANDSCAPE—Exports the pages in landscape orientation.

The option PRO_PDFOPT_ORIENTATION is not available if the property PRO_PDFOPT_SIZE is set to VARIABLE_SIZE_IN_MM_PLOT or VARIABLE_SIZE_PLOT.
- PRO_PDFOPT_VIEW_TO_EXPORT—Enables you to specify the view or views to be exported to the PDF file. The values are:

- `PRO_PDF_VIEW_SELECT_CURRENT`—Exports the current graphical area to a one-page PDF file.
- `PRO_PDF_VIEW_SELECT_BY_NAME`—Exports the selected view to a one-page PDF file with the view name printed at the bottom center of the view port. If this value is assigned, then the value of the string property `PRO_PDFOPT_SELECTED_VIEW` must also be included.
- `PRO_PDF_VIEW_SELECT_ALL`—Exports all the views to a multi-page PDF file. Each page contains one view with the view name displayed at the bottom center of the view port.

The function **ProPDFoptionsBoolpropertySet()** sets the value for a boolean property of the PDF options structure. The types of export options are as follows:

- `PRO_PDFOPT_SEARCHABLE_TEXT`—If true, stroked text is searchable. The default value is true.
- `PRO_PDFOPT_LAUNCH_VIEWER`—If true, launches the Adobe Acrobat Reader. The default value is true.
- `PRO_PDFOPT_HYPERLINKS`—Sets Web hyperlinks to be exported as label text only or sets the underlying hyperlink URLs as active. The default value is true, specifying that the hyperlinks are active.
- `PRO_PDFOPT_BOOKMARK_ZONES`—If true, adds bookmarks to the PDF showing zoomed in regions or zones in the drawing sheet. The zone on an A4-size drawing sheet is ignored.
- `PRO_PDFOPT_BOOKMARK_VIEWS`—If true, adds bookmarks to the PDF document showing zoomed in views on the drawing.
- `PRO_PDFOPT_BOOKMARK_SHEETS`—If true, adds bookmarks to the PDF document showing each of the drawing sheets.
- `PRO_PDFOPT_BOOKMARK_FLAG_NOTES`—If true, adds bookmarks to the PDF document showing the text of the flag note.
- `PRO_PDFOPT_RESTRICT_OPERATIONS`—If true, enables you to restrict or limit operations on the PDF document using the enum `ProPDFRestrictOperationsMode`. The default is false.
- `PRO_PDFOPT_ALLOW_EDITING`—If true, allows you to edit the PDF document. The default value is true.
- `PRO_PDFOPT_ALLOW_ADD_OR_REMOVE_PAGES`—If true, allows you to insert or remove pages from the PDF. The default value is true.

- `PRO_PDFOPT_ALLOW_COMMENTING`—If true, allows you to add or edit comments in the PDF. The default value is true.
- `PRO_PDFOPT_ALLOW_SIGNING`—If true, allows you to add digital signatures to the document. The default value is true.
- `PRO_PDFOPT_ALLOW_PRINTING`—If true, allows you to print the PDF document. The default value is true.
- `PRO_PDFOPT_ALLOW_COPYING`—If true, allows you to copy content from the PDF document. The default value is true.
- `PRO_PDFOPT_ALLOW_ACCESSIBILITY`—If true, enables visually-impaired screen reader devices to extract data independent of the value of the enum `ProPDFRestrictOperationsMode`. The default value is true.
- `PRO_PDFOPT_PENTABLE`—If true, uses the standard `Pro/ENGINEER` pentable to control the line weight, line style, and line color of the exported geometry. The default value is false.
- `PRO_PDFOPT_ADD_VIEWS`—If true, allows you to add view definitions to the U3D model from a file. The default value is true.

The function **ProPDFOptionsStringpropertySet()** sets the value for a string property of the PDF options structure. The types of export options are as follows:

- `PRO_PDFOPT_TITLE`—Specifies a title for the PDF document.
- `PRO_PDFOPT_AUTHOR`—Specifies the name of the person generating the PDF document.
- `PRO_PDFOPT_SUBJECT`—Specifies the subject of the PDF document.
- `PRO_PDFOPT_KEYWORDS`—Specifies relevant keywords in the PDF document.
- `PRO_PDFOPT_PASSWORD_TO_OPEN`—Sets a password to open the PDF document. If the value is not set or NULL, anyone can open the PDF document without a password.
- `PRO_PDFOPT_MASTER_PASSWORD`—Sets a password to restrict or limit the viewer operations that you can perform on the opened PDF document. If the value is set to NULL, you can make any changes to the PDF document regardless of the settings of the modification flags `PRO_PDFOPT_ALLOW_*`.

- **PRO_PDFOPT_SHEET_RANGE**—Specifies the range of sheets in a Pro/ENGINEER drawing that are to be exported to a PDF file. If this property is assigned, then the integer property **PRO_PDFOPT_SHEETS** is set to the value **PRINT_SELECTED_SHEETS**.
- **PRO_PDFOPT_SELECTED_VIEW**—Sets the option **PRO_PDFOPT_VIEW_TO_EXPORT** to the value **PRO_PDF_VIEW_SELECT_BY_NAME**, if the corresponding view is successfully found.

The function **ProPDFOptionsDoublepropertySet()** sets the value for a double property of the PDF options structure. The types of export options are as follows:

- **PRO_PDFOPT_HEIGHT**—Enables you to set the height for a user-defined page size of the exported PDF file. The default value is 0.0. This option is available only if the enum **PRO_PDFOPT_SIZE** is set to **VARIABLE_SIZE_IN_MM_PLOT** or **VARIABLE_SIZE_PLOT**.
- **PRO_PDFOPT_WIDTH**—Enables you to set the width for a user-defined page size of the exported PDF file. The default value is 0.0. This option is available only if the enum **PRO_PDFOPT_SIZE** is set to **VARIABLE_SIZE_IN_MM_PLOT** or **VARIABLE_SIZE_PLOT**.
- **PRO_PDFOPT_TOP_MARGIN**—Enables you to specify the top margin of the view port. The default value is 0.0.
- **PRO_PDFOPT_LEFT_MARGIN**—Enables you to specify the left margin of the view port. The default value is 0.0.
- **PRO_PDFOPT_BACKGROUND_COLOR_RED**—Enables you to specify the default red background color that appears behind the U3D model. You can set any value within the range of 0.0 through 255.0. The default value is 255.0.
- **PRO_PDFOPT_BACKGROUND_COLOR_GREEN**—Enables you to specify the default green background color that appears behind the U3D model. You can set any value within the range of 0.0 through 255.0. The default value is 255.0.
- **PRO_PDFOPT_BACKGROUND_COLOR_BLUE**—Enables you to specify the default blue background color that appears behind the U3D model. You can set any value within the range of 0.0 through 255.0. The default value is 255.0.

The function **ProPDFExport()** exports the file to a PDF document based on the export settings defined in the PDF options structure. Specify the complete name and path, including the extension of the output file.

Use the function **ProPDFWithProfileExport()** to repeatedly export 2D drawings to the PDF format with the same export options. These options are stored in an XML file called a profile. You can have several such profiles. The input arguments to this function are:

- *model*—A drawing model to export. This drawing model must be open and the drawing window must be active.
- *output_file*—The complete path to the output file with extension.
- *profile*—The path to the profile to be used.

Use the function **ProPDFoptionsFree()** to free the memory contained in the PDF options structure.

Importing Parameter Files

Functions Introduced:

- **ProInputFileRead()**

The function **ProInputFileRead()** imports files of several types to create data in Pro/ENGINEER. The file types are declared in *ProUtil.h*. The import formats and their type constants are as listed in the following table:

Import Format	Pro/TOOLKIT Functions	Type Constant
Relations file	ProInputFileRead()	PRO_RELATION_FILE
Program file		PRO_PROGRAM_FILE
Configuration options file		PRO_CONFIG_FILE
Setup file		PRO_DWG_SETUP_FILE
Spool file		PRO_SPOOL_FILE
Cable Parameters file		PRO_CABLE_PARAMS_FILE
Connector Parameters file		PRO_CONNECTOR_PARAMS_FILE
Model Tree Configuration file		PRO_ASSEM_TREE_CFG_FILE

Import Format	Pro/TOOLKIT Functions	Type Constant
Wirelist file		PRO_WIRELIST_FILE
SLD Variant file		SLD_VARIANT_FILE

The option `PRO_RELATION_FILE` reads a text file that contains a list of all the model relations and parameters relations in exactly the same format as the Pro/ENGINEER user enters them.

Use the function **ProInputFileRead()** with the argument `PRO_CONNECTOR_PARAMS` to identify the connectors. To access parameters on connectors and their entry ports use the following arguments:

- *arg1*—Represents the integer pointer to *ProIdTable*. *ProIdTable* is an integer array of component identifiers.
- *arg2*—Represents the integer pointer to the number of component identifiers.

Use the function **ProInputFileRead()** with the argument `PRO_CABLE_PARAMS_FILE` to read cable parameters. You need to set the following arguments:

- *arg1*—Represents a *ProSolid* (part pointer).
- *arg2*—Represents the cable name.

Use the function **ProInputFileRead()** with the argument `PRO_WIRELIST_FILE` to read files in Mentor Graphics LCABLE format. This function does not create wires, but provides parameters from a wire list for use when creating in a harness assembly a wire with the same name as that in the LCABLE file.

Use function **ProInputFileRead()** with argument `PRO_SPOOL_FILE` to create new spools or update existing ones.

Importing 2D Models

Import Format	Pro/TOOLKIT Functions	Type Constant
STEP file	Pro2dImportCreate(), Pro2dImportAppend()	PRO_STEP_FILE
IGES (2D) file		PRO_IGES_FILE
DXF file		PRO_DXF_FILE
DWG file		PRO_DWG_FILE
CGM file		PRO_CGM_FILE
MEDUSA file		PRO_MEDUSA_FILE
SET file	Pro2dImportAppend()	PRO_SET_FILE
IGES (2D) file	ProInputFileRead()	PRO_IGES_SECTION

Functions Introduced:

- **Pro2dImportCreate()**
- **Pro2dImportAppend()**
- **ProInputFileRead()**

The function **Pro2dImportCreate()** imports interface files and creates a new two-dimensional model with the specified name. The created models can be drawings, layouts, diagrams, or drawing formats. Use the argument `import_2d_views` to control whether or not to import two-dimensional views.

The function **Pro2dImportAppend()** appends a two-dimensional model to a specified, existing model.

Use the function **ProInputFileRead()** with the argument `PRO_IGES_SECTION` to import a 2-D IGES section into a sketch.

Importing 3D Models

The functions described in this section are used to import files of different format types into Pro/ENGINEER.

Functions Introduced:

- **ProIntfimportSourceTypeGet()**
- **ProIntfimportModelCreate()**
- **ProIntfimportModelWithProfileCreate()**
- **ProIntfimportLayerFilter()**

Import Format	Pro/TOOLKIT Functions	Type Constant
ACIS file	ProIntfimportModelCreate() ProIntfimportModelWithProfileCreate()	PRO_INTF_IMPORT_ACIS
CADDS file		PRO_INTF_IMPORT_CADDS
CATIA (.model) file		PRO_INTF_IMPORT_CATIA_MODEL
CATIA (.session) file		PRO_INTF_IMPORT_CATIA_SESSION
Pro/Desktop file (.pdt)		PRO_INTF_IMPORT_DESKTOP
DXF file		PRO_INTF_IMPORT_DXF
ICEM file		PRO_INTF_IMPORT_ICEM
IGES file		PRO_INTF_IMPORT_IGES
Neutral file		PRO_INTF_IMPORT_NEUTRAL
Parasolid-based CADDs system file		PRO_INTF_IMPORT_PARASOLID
POLTXT file		PRO_INTF_IMPORT_POLTXT
SET file		PRO_INTF_IMPORT_SET
STEP file		PRO_INTF_IMPORT_STEP
VDA file		PRO_INTF_IMPORT_VDA
CATIA (.CATpart) file		PRO_INTF_IMPORT_CATIA_PART
UG file		PRO_INTF_IMPORT_UG
PRODUCTVIEW (.ol and .ed) files		PRO_INTF_IMPORT_PRODUCTVIEW

Import Format	Pro/TOOLKIT Functions	Type Constant
JT Open format	ProIntfimportModelCreate()	PRO_INTF_IMPORT_JT
CATIA Graphical Representation (CGR) format	ProIntfimportModelWithProfileCreate()	PRO_INTF_IMPORT_CATIA_CGR
SolidWorks Part (.sldprt) file		PRO_INTF_IMPORT_SW_PART
SolidWorks Asembly (.sldasm) file		PRO_INTF_IMPORT_SW_ASSEM
Inventor Part (.ipt) file		PRO_INTF_IMPORT_INVENTOR_PART
Inventor Assembly (.iam) file		PRO_INTF_IMPORT_INVENTOR_ASSEM
STL file	ProIntfimportModelCreate()	PRO_INTF_IMPORT_STL
VRML file		PRO_INTF_IMPORT_VRML

The function **ProIntfimportSourceTypeGet()** is a utility that returns the type of model that can be created from the geometry file. This function is not applicable for all formats. If this function is not valid for a geometric file, you will need to know the type of model you want to create (part, assembly, or drawing).

The function **ProIntfimportModelCreate()** imports objects of other formats and creates a new model or set of models with the specified name. The input arguments of this function are:

- *import_file*—Full path to file to be imported.
- *type*—The type of file to be imported. This could be part, assembly, or drawing (for STEP associative drawings).
- *create_type*—The type of model to create.
- *new_model_name*— The name of the new top level import model.
- *filter_func*—Callback to the function **ProIntfimportLayerFilter()** that determines how to display and map layers from the imported model. If this is NULL, the default layer handling will take place.
- *application_data*—The application data to be passed to the filter function. Can be NULL.

The function **ProIntfimportModelWithProfileCreate()** imports objects of other formats using a profile and creates a new model or set of models with the specified name. This function takes the same input arguments as the function **ProIntfimportModelCreate()**, except for the argument *profile* that specifies the path to the import profile used. An import profile is an XML file with the extension *.dip* (Dex In Profile) and contains the options that control an import operation. It contains all the options for the supported 3D import formats. Refer to the Pro/ENGINEER Online Help for more information on creation and modification of import profiles.

The function **ProIntfimportLayerFilter()** is a callback function that allows your application to determine the status of each of the imported layers. You can modify the layer information using the functions described in the next section.

Modifying the Imported Layers

Layers help you organize model items so that you can perform operations on those items collectively. These operations primarily include ways of showing the items in the model, such as displaying or blanking, selecting, and suppressing. The methods described in this section modify the attributes of the imported layers.

Functions Introduced:

- **ProLayerfilterdataNameGet()**
- **ProLayerfilterdataNameSet()**
- **ProLayerfilterdataCountsGet()**
- **ProLayerfilterdataActionSet()**

Imported layers are identified by their names. The function **ProLayerfilterdataNameGet()** returns the name of the layer while the function **ProLayerfilterdataNameSet()** can be used to set the name of the layer.

The function **ProLayerfilterdataCountsGet()** specifies the following:

- Number of curves on the specified layer
- Number of surfaces on the specified layer
- Number of trimmed surfaces on the specified layer

The function **ProLayerfilterdataActionSet()** sets the display status of the imported layers. You can set the display status of the layers to one of the following:

- Show—Display the specified layer.
- Blank—Make the specified layer blanked.
- Hidden—(Assembly mode only) Make the specified layer hidden.
- Skip—Do not import the entities on this layer.
- Ignore—Import only entities on this layer but not the layer

Example 5: Importing a 3D Model With Layer Filter Options

This example demonstrates how to import from an IGES file while filtering out a certain layer to be renamed and blanked, using the filter function **UserLayerFilter()**. It also demonstrates how to use the function **ProIntfimportSourceTypeGet()** to determine the type of the model.

```
#include <ProToolkit.h>
#include <ProMenu.h>
#include <ProMdl.h>
#include <ProIntfimport.h>

#define MSGFIL L##"msg_uginterface.txt"

/*=====*\
STRUCTURE: UserImportLayerFilterdata
PURPOSE:   Contains user data to be passed to the import layer filter
function.
\*=====*/
typedef struct
{
    ProName find_layer;
} UserImportLayerFilterdata;

/*=====*\
FUNCTION: UserLayerFilter
PURPOSE:   Blanks and renames an unneeded layer
\*=====*/
ProError UserLayerFilter ( ProLayerfilterdata data,
                          ProAppData application_data)
{
    ProName layer_name;
    ProError status;
    UserImportLayerFilterdata* filterdata =
        (UserImportLayerFilterdata*)application_data;

/*-----*\
Check if the current layer is the target
\*-----*/
```



```

    status = ProLayerfilterdataNameGet (data, layer_name);

    if (ProUtilWstrcmp (filterdata->find_layer, layer_name) == 0)
    {
/*-----*\
Blank the found layer
\*-----*/
        status = ProLayerfilterdataActionSet (data,
                                                PRO_LAYER_IMPORT_BLANK);

/*-----*\
Rename the found layer
\*-----*/
        status = ProLayerfilterdataNameSet (data, L"FOUND");
    }

    return PRO_TK_NO_ERROR;
}

/*=====*\
FUNCTION: UserIGESImportNewModelViaFilter()
PURPOSE: Imports an IGES assembly file to an new model while filtering
layers
\*=====*/
int UserIGESImportNewModelViaFilter(ProPath input_file,
                                    ProName new_model_name,
                                    ProName excluded_layer,
                                    ProMdl* model)
{
    ProError status;
    ProMdl created_model;
    ProIntfImportType import_type = PRO_INTF_IMPORT_IGES;
    ProMdlType mdl_type;

/*-----*\
Set up the layer filter data - search for needed layer
\*-----*/
    UserImportLayerFilterdata data;
    ProUtilWstrcpy (data.find_layer, excluded_layer);

    *model = NULL;

/*-----*\
Check for proper import model type
\*-----*/
    status = ProIntfimportSourceTypeGet (input_file, import_type, &mdl_type);

    if (status != PRO_TK_NO_ERROR)
        return PRO_TK_GENERAL_ERROR;

```

```

/*-----*\
Import IGES file
\*-----*/
    status = ProIntfimportModelCreate(input_file,
                                      import_type,
                                      mdl_type,
                                      new_model_name,
                                      UserLayerFilter, &data,
                                      &created_model);

    if (status == PRO_TK_NO_ERROR)
    {
        *model = created_model;
        return PRO_TK_NO_ERROR;
    }
    else
        return status;
}
#undef MSGFIL

```

Interface: Importing Features

This chapter describes how to create import features in Pro/ENGINEER.

Topic	Page
Creating Import Features from Files	27 - 2
Creating Import Features from Arbitrary Geometric Data	27 - 4
Redefining the Import Feature	27 - 16
Import Feature Properties	27 - 17
Extracting Pro/ENGINEER Geometry as Interface Data	27 - 19

Creating Import Features from Files

To create import features in Pro/ENGINEER from external format files use the functions described in this section.

Functions Introduced:

- **ProIntfDataSourceInit()**
- **ProImportfeatCreate()**
- **ProImportfeatWithProfileCreate()**

The function **ProIntfDataSourceInit()** is used to build the interface data source required by the functions **ProImportfeatCreate()** and **ProImportfeatWithProfileCreate()**.

The input arguments of this function are:

- **intf_type**—Specifies the type of file to import. The valid format files from which the user can create the import features are:

Type Constant	Import Format
PRO_INTF_NEUTRAL_FILE	Neutral file
PRO_INTF_IGES	IGES 3D file
PRO_INTF_STEP	STEP file
PRO_INTF_VDA	VDA file
PRO_INTF_SET	SET file
PRO_INTF_PDGS	PDGS file
PRO_INTF_DESKTOP	Pro/DESKTOP file
PRO_INTF_ICEM	ICEM file
PRO_INTF_ACIS(*.sat)	ACIS format file
PRO_INTF_DXF	DXF file
PRO_INTF_CDRS	CDRS file
PRO_INTF_STL	STL file
PRO_INTF_VRML	VRML file
PRO_INTF_PARASOLID	Parasolid-based CADDs system file
PRO_INTF_AI	AI file
PRO_INTF_CATIA_PART	CATIA (.CATpart) file
PRO_INTF_UG	UG file

Type Constant	Import Format
PRO_INTF_PRODUCTVIEW	PRODUCTVIEW (.ol) files
PRO_INTF_CATIA_PRODUCT	CATIA V5 Assembly file
PRO_INTF_CATIA_CGR	CATIA Graphical Representation (CGR) format
PRO_INTF_JT	JT Open Interface
PRO_INTF_INVENTOR_PART	Inventor Part (.ipt) file
PRO_INTF_INVENTOR_ASM	Inventor Assembly (.iam) file

- *p_source*—the name of the file with extension. The specified format file should exist in the current working directory or in a path specified in the *search_path* configuration option.

This function returns the handle to the *ProIntfDataSource* object, which should be passed to the functions **ProImportfeatCreate()** and **ProImportfeatWithProfileCreate()**.

The function **ProImportfeatCreate()** is used to create a new import feature in the Pro/ENGINEER solid model. The input arguments of this function are:

- *p_solid*—Specifies the part in which the user wants to create the import feature.
- *data_source*—Specifies a pointer to the interface data source. Use the function **ProIntfDataSourceInit()** to get the handle to the *ProIntfDataSource* object.
- *p_csys*—Specifies the co-ordinate system of the part with which the user wants to align the import feature. If this is NULL, the function uses the default coordinate system in the Pro/ENGINEER model and the import feature will be aligned with respect to this coordinate system.
- *p_attributes*—Specifies the attributes for the creation of the new import feature. Please see the section Import Feature Attributes for more information.

The function **ProImportfeatCreate()** returns the *ProFeature* handle for the created import feature.

The function **ProImportfeatWithProfileCreate()** is used to create a new import feature in the Pro/ENGINEER solid model. This function takes the same input arguments as the function **ProImportfeatCreate()**, except for the argument *profile* that specifies the path to the import profile used. An import profile is an

XML file with the extension `.dip` (Dex In Profile) and contains the options that control an import operation. It contains all the options for the supported 3D import formats. Refer to the Pro/ENGINEER Online Help for more information on creation and modification of import profiles.

Note: The function **ProImportfeatWithProfileCreate()** cannot create an import feature using an import profile for the STL and VRML formats.

Creating Import Features from Arbitrary Geometric Data

You can create an import feature in a Pro/ENGINEER model by building the required entity data in the Pro/TOOLKIT application.

The advantages of importing features from a Pro/TOOLKIT application are:

- You can create virtually non-parametric user-defined geometry at a desired location. This is sometimes an alternative to parametric feature creation, which can be more complicated.
- Import features are regenerated more quickly than corresponding groups of parametric features.
- You can integrate Pro/ENGINEER with non-Pro/ENGINEER supported geometry file formats.

The following sequence of steps is required to create the import feature from memory:

- Allocate the interface data.
- Add surfaces, edges, quilts, and datums.
- Create the import feature from the interface data.

These steps are described in detail in the following sections.

Allocating ProInterfacedata

Function Introduced:

- **ProIntfDataAlloc()**

Use the function **ProIntfDataAlloc()** to allocate memory for the interface data structure.

Adding Surfaces

Function Introduced:

- **ProSurfacedataAlloc()**

Use the function **ProSurfacedataAlloc()** to allocate memory for the surface data. Once the surface data is initialized, it will be appended to the interface data.

Initializing Surface Data

Functions Introduced:

- **ProSurfacedataInit()**
- **ProPlanedataInit()**
- **ProCylinderdataInit()**
- **ProConedataInit()**
- **ProTorusdataInit()**
- **ProSrfrevdataInit()**
- **ProTabcyldataInit()**
- **ProRulsrfddataInit()**
- **ProSplinesrfddataInit()**
- **ProCylsplsrfddataInit()**
- **ProBsplinesrfddataInit()**
- **ProFilsrfddataInit()**

Use the function **ProSurfacedataInit()** to initialize the surface data structure.

The input arguments of this function are:

- **Surface_type**—Specifies the type of surface to be created. The types of surfaces are:
 - **PRO_SRF_PLANE**—Plane
 - **PRO_SRF_CYL**—Cylinder
 - **PRO_SRF_CONE**—Cone
 - **PRO_SRF_TORUS**—Torus
 - **PRO_SRF_COONS**—Coons Patch
 - **PRO_SRF_SPL**—Spline Surface

- PRO_SRF_FIL—Fillet Surface
- PRO_SRF_RUL—Ruled Surface
- PRO_SRF_REV—General Surface of Revolution
- PRO_SRF_TABCYL—Tabulated Cylinder
- PRO_SRF_B_SPL—B-spline surface
- PRO_SRF_FOREIGN—Foreign Surface
- PRO_SRF_CYL_SPL—Cylindrical Spline Surface

The type of the surface determines the function to be used to initialize the surface data structure.

For example, if the type of surface to be created is PRO_SRF_PLANE, then the function **ProPlanedataInit()** should be used to initialize the surface data structure.

- surf_uv_min—Specifies the minimum uv extents of the surface.
- surf_uv_max—Specifies the maximum uv extents of the surface.
- surf_orient— Specifies the orientation of the surface. By default the value is PRO_SURF_ORIENT_OUT
- p_surf_shape—The data containing the information about the shape of the surface.
- Surface_Id—Specifies a unique identifier of the Surface.

Depending on the shape of the surface, call one of the following functions to create the surface data structure ProSurfeshapedata and assign it to variable p_surf_shape of function

ProSurfacedataInit(). Ensure that the function used to create the ProSurfeshapedata matches with the ProSrftype value used in **ProSurfacedataInit()**.

- **ProPlanedataInit()**
- **ProCylinderdataInit()**
- **ProConedataInit()**
- **ProTorusdataInit()**
- **ProSrfrevdataInit()**
- **ProTabcyldataInit()**
- **ProRulsrfdataInit()**
- **ProSplinesrfdataInit()**

- **ProCylsplrsrfddataInit()**
- **ProBsplinesrfddataInit()**

Note: Set the configuration option `intf_in_keep_high_deg_bspl_srfs` to YES to preserve the B-spline surfaces returned by **ProBsplinesrfddataInit()** in the *ProIntData* data structure. If this configuration option is not set, these surfaces are interpreted as spline surfaces.

- **ProFilsrfddataInit()**

Refer to the ‘Geometry Representations’ appendix for more information on how to use the above functions.

Note: The following return values for the functions **ProBsplinesrfddataInit()**, **ProRulsrfddataInit()**, **ProSrfrevdataInit()**, and **ProTabcyldataInit()** should be treated as warnings:

- PRO_TK_BSPL_UNSUITABLE_DEGREE
- PRO_TK_BSPL_NON_STD_END_KNOTS
- PRO_TK_BSPL_MULTI_INNER_KNOTS

They indicate that the geometry finally imported in Pro/ENGINEER is different from the geometry initially supplied to the above functions. The geometry is not rejected by the functions and is used to generate the *ProSurfacedata* data structure.

Surfacedata Contours

The geometric representation of the surface created above is unbounded, that is the nature of the surface boundaries is determined by its array of contours. Multiple contours can be used for surfaces with internal voids.

Functions Introduced:

- **ProSurfacedataContourArraySet()**
- **ProContourdataAlloc()**
- **ProContourdataInit()**
- **ProContourdataEdgeldArraySet()**

Use the function **ProSurfacedataContourArraySet()** to set an array of contours on the surface.

The input arguments of this function are:

- **p_surf_data**—Specifies the surface data to which the array of contour data is to be set.
- **contour_array**—Specifies an array of contours on the surface. The **ProContourdata** handle can be obtained by using the following functions in sequence:
 - **ProContourdataAlloc()**
 - **ProContourdataInit()**
 - **ProContourdataEdgeIdArraySet()**

Use the function **ProContourdataAlloc()** to allocate memory to the contour data structure.

Use the function **ProContourdataInit()** to initialize the contour data structure. The input argument of this function is:

- **contour_trav**—Specifies the contour traversal. This parameter has the following values:
 - **PRO_CONTOUR_TRAV_INTERNAL**—Internal Contour
 - **PRO_CONTOUR_TRAV_EXTERNAL**—External Contour

The function returns the allocated contour data structure.

Use the function **ProContourdataEdgeIdArraySet()** to set identifiers to an array of edges, that form the boundary of the specified surface.

The input arguments of this function are:

- **p_contour_data**—Specifies the contour data to which the array of edge identifiers have to be set.
- **edge_id_arr**—Specifies the array of edge identifiers. These identifiers must be same as those provided in the **ProEdgedata** structures described below.

For example, if the surface is bounded by 4 edges, then the identifier of each edge should be assigned to each element of an array of integers of size 4.

Appending the Surface Data to the Interface Data

Function Introduced:

- **ProIntfDataSurfaceAppend()**

Use the function **ProIntfDataSurfaceAppend()** to append the surface data into the interface data.

Repeat the sequence for each surface desired in the import feature.

Adding Edges

Functions Introduced:

- **ProEdgedataAlloc()**
- **ProEdgedataInit()**
- **ProCurvedataAlloc()**
- **ProLinedataInit()**
- **ProArcdataInit()**
- **ProEllipsedataInit()**
- **ProSplinedataInit()**
- **ProBsplinedataInit()**

If the import feature to be created requires any edge information, then call the functions list above in sequence, else skip this section.

Use the function **ProEdgedataAlloc()** to allocate memory for the edge data structure. After initialization, this data will be appended to the interface data.

Use the function **ProEdgedataInit()** to initialize the edge data structure. The following are the input arguments:

- *edge_id*—Specifies a unique identifier of the edge.
- *edge_surf_ids*—Specifies the ID of the surfaces on either side of the edge.
- *edge_directions*—Specifies the edge directions on the surface.
- *edge_uv_point_arr*—Specifies an array of UV points on the surfaces. The value can be NULL.
- *p_edge_uv_curve_data*—Specifies the edge UV curves on the surfaces. The value can be NULL.
- *p_edge_curve_data*—Specifies the curve data handle in the form of the *ProCurvedata* structure. This data handle is returned by the functions **ProLinedataInit()**, **ProArcdataInit()**, **ProEllipsedataInit()**, **ProSplinedataInit()**, or **ProBsplinedataInit()**. Use the function **ProCurvedataFree** to free the *ProCurvedata* data handle.

Note: PTC recommends that you split the closed loop edge into two or more continuous edges while specifying the inputs to the function **ProEdgedataInit()**. For example, to create a circular edge, instead of specifying the start angle as 0 and the end angle as 360, split the circular edge into 2 or more edges. The angular measurements of the split edges could be 0 to 30 for the first split and 30 to 360 for the second split. The function **ProEdgedataInit()** must be called for each split.

Use the function **ProCurvedataAlloc()** to allocate memory for the curve data structure. The curve data structure defines the edge profile.

Depending on the type of curve specified for the edge, call one of the following functions to initialize the curve data.

- **ProLinedataInit()**
- **ProArcdataInit()**
- **ProEllipsedataInit()**
- **ProSplinedataInit()**
- **ProBsplinedataInit()**

Use the function **ProLinedataInit()** to initialize the line data structure. Specify the start of the line and end of the line as inputs of this function.

Use the function **ProArcdataInit()** to initialize an arc data structure. The input arguments of this function are:

- `vector1`—Specifies the first vector of the arc coordinate system.
- `vector2`—Specifies the second vector of the arc coordinate system.
- `origin`—Specifies the center of the arc coordinate system
- `start_angle`—Specifies the starting angle (in radians) of the arc.
- `end_angle`—Specifies the end angle (in radians) of the arc.
- `radius`—Specifies the radius of the arc.

Use the function **ProEllipsedataInit()** to initialize an ellipse data structure. The input arguments of this function are:

- `center`—Specifies the center of the ellipse.

- `x_axis`—Specifies the first (x) axis vector of the ellipse.
- `plane_normal`—Specifies the axis vector that is normal to the plane of the ellipse.
- `x_radius`—Specifies the radius of the ellipse in the direction of 'x' axis.
- `y_radius`—Specifies the radius of the ellipse in the direction of 'y' axis. The 'y' axis can be found as a vector product of the `plane_normal` on `x_axis`.
- `start_ang`—Specifies the starting angle (in radians) of the ellipse.
- `end_ang`—Specifies the end angle (in radians) of the ellipse.

Use the function **ProSplinedataInit()** to initialize the spline data structure. The input arguments of this function are:

- `par_arr`—Specifies an array of spline parameters
- `pnt_arr`—Specifies an array of spline interpolant points
- `tan_arr`—Specifies an array of tangent vectors at each point
- `num_points`—Specifies the size for all the arrays

Use the function **ProBsplinedataInit()** to initialize the B-spline data structure. The input arguments of this function are:

- `degree`—Specifies the degree of the basis function.
- `params`—Specifies an array of knots on the parameter line.
- `weights`—In the case of rational B-splines, it specifies an array of the same dimension as the array of `c_pnts`. Else, the value of this argument is NULL.
- `c_pnts`—Specifies an array of knots on control points.
- `num_knots`—Specifies the size of the `params` array.
- `num_c_points`—Specifies the size of the `c_pnts` and the size of `weights` if it is not NULL.

Note:

- Although **ProBsplinedataInit()** returns B-spline curves, these curves are interpreted as spline curves in the *ProIntData* data structure used by the function **ProImportfeatCreate()** while creating the import feature.

- The values `PRO_TK_BSPL_UNSUITABLE_DEGREE` and `PRO_TK_BSPL_NON_STD_END_KNOTS` returned by **ProBsplinedataInit()** should be treated as warnings. These values indicate that the geometry finally imported in Pro/ENGINEER is different from the geometry initially supplied to the function. The geometry is not rejected by **ProBsplinedataInit()** and is used to generate the *ProCurvedata* data structure.

Appending the Edge Data to the Interface Data

Function Introduced:

- **ProIntfDataEdgeAppend()**

Use the function **ProIntfDataEdgeAppend()** to append the edge data into the interface data.

Repeat the sequence for each edge required by the import feature.

Adding Quilts

Functions Introduced:

- **ProQuiltdataAlloc()**
- **ProQuiltdataInit()**
- **ProQuiltdataSurfArraySet()**
- **ProIntfDataQuiltAppend()**

Use the function **ProQuiltdataAlloc()** to allocate memory to the quilt data structure.

Use the function **ProQuiltdataInit()** to assign the user defined identity to the quilt data structure. Specify a unique identity for the quilt as the input argument. The function returns the handle to the quilt data structure.

Use the function **ProQuiltdataSurfArraySet()** to define an array of surfaces as a quilt. The input arguments of this function are:

- `p_quilt_data`—Specifies a handle to the quilt data to which we want to assign the set of surfaces.
- `arr_p_surf`—Specifies an array of surfaces that will be defined as a quilt.

Use the function **ProIntfDataQuiltAppend()** to append the quilt data to the interface data. The input arguments of this function are:

- `p_intfdata`—Specifies a handle to the interface data to which you want to append the quilt data.
- `p_quiltdata`—Handle to the quilt data.

Repeat the sequence for each quilt required in the import feature.

Adding Datums

Functions Introduced:

- **ProDatumdataAlloc()**

Use the function **ProDatumdataAlloc()** to allocate memory to the datum data structure.

Initializing Datums

- **ProDatumdataInit()**
- **ProDatumCsysdataInit()**
- **ProDatumCurvedataInit()**
- **ProDatumPlanedataInit()**

Use the function **ProDatumdataInit()** to initialize the datum data structure. The input arguments of this function are:

- `datum_id`—Specifies a unique identifier of the datum.
- `datum_type`—Specifies the datum type. The types of datums are:
 - `PRO_CSYS`
 - `PRO_CURVE`
 - `PRO_DATUM_PLANE`
- `datum_name`—Specifies the name to be assigned to the datum.
- `p_datum_obj`—The datum object that contains the geometrical information about the datum. Depending on the type of the datum to be created, one of the following functions must be used to create the *ProDatumobject* data structure.
 - **ProDatumCsysdataInit()**
 - **ProDatumCurvedataInit()**
 - **ProDatumPlanedataInit()**

Note: The value `PRO_TK_BSPL_MULTI_INNER_KNOTS` returned by **ProDatumCurvedataInit()** should be treated as a warning. This value indicates that the geometry finally imported in Pro/ENGINEER is different from the geometry initially supplied to the function. The geometry is not rejected by **ProDatumCurvedataInit()** and is used to generate the *ProCurvedata* data structure.

Appending the Datum Data to the Interface Data

Use the function **ProIntfDataDatumAppend()** to append the datum data to the interface data required to create the import feature. The input arguments are:

- `p_intfdata`—Specifies the interface data to which the datum data must be appended.
- `p_datumdata`—Specifies a handle to the datum data obtained from the function **ProDatumdataInit()**.
- Repeat the sequence for each datum member required to be in the import feature.

Creating Features from the Interface Data

Functions Introduced:

- **ProIntfDataSourceInit()**
- **ProImportfeatCreate()**
- **ProImportfeatWithProfileCreate()**
- **ProIntfDataFree()**

Use the function **ProIntfDataSourceInit()** to build the interface data source required by the functions **ProImportfeatCreate()** and **ProImportfeatWithProfileCreate()**. The input arguments of this function are:

- `intf_type`—Specifies the type of the interface. Since the user builds all the data required by the interface, the value should be `PRO_INTF_NEUTRAL`.
- `p_source`—Specifies the handle to the interface data source.

The function returns the handle *ProIntfDataSource*, which must be passed to the functions **ProImportfeatCreate()** and **ProImportfeatWithProfileCreate()**.

Use the function **ProImportfeatCreate()** to create the import feature in the Pro/ENGINEER solid model. The input arguments of this function are:

- **p_solid**—Specifies the part or assembly in which the user wants to create the import feature.
- **data_source**—Specifies a pointer to the interface data source. Use the function **ProIntfDataSourceInit()** to get the handle to the interface data source.
- **p_csys**—Specifies the co-ordinate system of the part with which you want to align the import feature. If this is NULL, the function uses the default coordinate system in the Pro/ENGINEER model and the import feature will be aligned with respect to that coordinate system.
- **p_attributes**—Specifies the attributes for the creation of the new import feature. Refer to the section Import Feature Attributes for more information.

The function **ProImportfeatCreate()** returns the *ProFeature* handle for the created import feature.

The function **ProImportfeatWithProfileCreate()** is used to create a new import feature in the Pro/ENGINEER solid model. This function takes the same input arguments as the function **ProImportfeatCreate()**, except for the argument *profile* that specifies the path to the import profile used. An import profile is an XML file with the extension `.dip` (Dex In Profile) and contains the options that control an import operation. It contains all the options for the supported 3D import formats. Refer to the Pro/ENGINEER Online Help for more information on creation and modification of import profiles.

Note: The function **ProImportfeatWithProfileCreate()** cannot create an import feature using an import profile for the STL and VRML formats.

Use the function **ProIntfDataFree()** to release the memory occupied by the interface data.

Import Feature Attributes

Attributes define the action to be taken when creating the import feature. Following are the defined attributes:

- **attempt_make_solid**—Specifies whether the import feature is to be created as a solid or a surface type. Set the value to 1 to create an import feature of solid type. Set it to 0 to create a surface type of import feature.

Note: If the import feature is an open surface, setting *attempt_make_solid* to 1 does not make the import feature of solid type.

- *cut_or_add*—Specifies whether the solid type of import feature is to be created as a cut or a protrusion. This argument is valid only if *attempt_make_solid* is set to 1. Set the value to 1 to cut the solid import feature from the intersecting solid. Set it to 0 to create it as a protrusion.

Note: When *attempt_make_solid* is set to 0, the value assigned to *cut_or_add* is not considered.

- *join_surfaces*—Specifies whether the import feature is created as a single quilt (joined surface) or separate surfaces (as it was in the original file) if it is of surface type. This argument is valid only if *attempt_make_solid* is set to 0. If the value is set to 1, all surfaces that can be joined are joined to form a single quilt.

Redefining the Import Feature

Use the following functions in sequence to redefine the import feature.

Functions Introduced:

- **ProImportfeatRedefSourceInit()**
- **ProImportfeatRedefine()**

Use the function **ProImportfeatRedefSourceInit()** to initialize the redefine source. Currently Pro/TOOLKIT users may

- Redefine the attributes of any import feature.

Note: When redefining the attributes of the import feature, Pro/ENGINEER will not use the value of the attribute “join_surfaces”, because this attribute is valid only for import feature creation.

- Redefine the geometry of an import feature created from a geometric file. Import features created from memory may not be redefined.

The input arguments are:

- *operation*—Specifies the type of operation to use when redefining the import feature.

- `p_source`—Specifies the handle to the new interface data or the new attributes structure.

The function **ProImportfeatRedefSourceInit()** returns the handle to a structure, that is passed as an input argument to the function **ProImportfeatRedefine()**.

Use the function **ProImportfeatRedefine()** to redefine the import feature. The input arguments are:

- `p_feat_handle`—Specifies the handle for the import feature to be redefined.
- `p_source`—The handle to be used for redefinition from the function **ProImportfeatRedefSourceInit()**.

Note: **ProImportfeatRedefine()** does not support ATB-enabled features. It returns `PRO_TK_BAD_CONTEXT` while accessing such features.

Import Feature Properties

Functions Introduced:

- **ProImportfeatIdArrayCreate()**
- **ProImportfeatIdArrayMapCount()**
- **ProImportfeatIdArrayMapGet()**
- **ProImportfeatIdArrayFree()**
- **ProImportfeatUserIdToItemId()**
- **ProImportfeatItemIdToUserId()**
- **ProImportfeatDataGet()**

Use the function **ProImportfeatIdArrayCreate()** to create an array of mappings between the user defined ids and the ids assigned by Pro/ENGINEER to the entity items in the import feature.

Specify the handle to the feature, for which the user defined ids and ids assigned by Pro/ENGINEER have to be mapped, as the input argument of the function. The function returns an array of mapped ids.

Use the function **ProImportfeatIdArrayMapCount()** to get the number of elements in the array of mappings. Use the function **ProImportfeatIdArrayMapGet()** to get the mapping of a particular element in the array.

Use the function **ProImportfeatIdArrayFree()** to free the array.

Use the function to **ProImportfeatUserIdToItemId()** to obtain the id or ids assigned by Pro/ENGINEER for a user defined id. The function returns multiple ids in an array if the import operation split a particular entity.

For example, if you create a circular edge as a single edge data defined by a single id, Pro/ENGINEER creates the circle by splitting it into two. If you pass the user defined id as an input to the function **ProImportfeatUserIdToItemId()**, the function will return an array of the ids assigned to each half of the circle.

The input arguments of this function are:

- p_feat_handle—Specifies the handle of the import feature.
- user_id—Specifies the identifier of the geometry item.
- item_type—Specifies the type of the geometry item. The types of geometry are:
 - PRO_SURFACE
 - PRO_EDGE
 - PRO_QUILT

Use the function **ProImportfeatItemIdToUserId()** to convert a Pro/ENGINEER item id to an array of user defined ids.

For example, if the edges defined by the user are created as a single edge by Pro/ENGINEER, and you pass a single item id assigned by Pro/ENGINEER to the function

ProImportfeatItemIdToUserId(), it will return an array of user ids.

- p_feat_handle—Specifies the handle of the import feature.
- item_id—Specifies the identifier of the geometry item.
- item_type—Specifies the type of the geometry item. The types of geometry are:
 - PRO_SURFACE
 - PRO_EDGE
 - PRO_QUILT

Use the function **ProImportfeatDataGet()** to retrieve the parameters assigned to the import feature. The output returned by this function will contains the following:

- Information about the interface type of the import feature.

- The filename from which the import feature is created. This is applicable for import features created from a file.
- The coordinate system with respect to which the import feature is aligned.
- The attributes of the import feature.

Extracting Pro/ENGINEER Geometry as Interface Data

Functions Introduced:

- **ProPartToProIntfData()**
- **ProIntfDataAccuracyGet()**
- **ProIntfDataAccuracytypeGet()**
- **ProIntfDataOutlineGet()**
- **ProIntfDataDatumCount()**
- **ProIntfDataDatumGet()**
- **ProIntfDataEdgeCount()**
- **ProIntfDataEdgeGet()**
- **ProIntfDataQuiltCount()**
- **ProIntfDataQuiltGet()**
- **ProIntfDataSurfaceCount()**
- **ProIntfDataSurfaceGet()**

Use the function **ProPartToProIntfData()** to extract a **ProIntfData** structure describing the geometry of a part as if it were an import feature. This provides a single interface to extract all geometric data in order to convert it to another geometric format.

The functions **ProIntfDataAccuracytypeGet()**, **ProIntfDataAccuracyGet()**, and **ProIntfDataOutlineGet()** provide access to properties of the interface data structure.

The other functions allow you to count and access each individual geometric data structure in the interface data.

28

Interface: Customized Plot Driver

This chapter describes the customized plot driver functions supported by Pro/TOOLKIT.

Topic	Page
Using the Plot Driver Functionality	28 - 2

Using the Plot Driver Functionality

Functions Introduced:

- **prointerface_create()**
- **prointerface_object_set()**
- **prointerface_load_function()**
- **prointerface_2d()**
- **user_intf_text()**
- **user_intf_circle()**
- **user_intf_arc()**
- **user_intf_line()**
- **user_intf_polyline()**
- **user_intf_filled_poly()**

These functions enable you to implement your own plot format in Pro/ENGINEER. You do this by providing your own function for plotting each of the two-dimensional primitives Pro/ENGINEER uses (line, circle, arc, text, and so on), and binding them to Pro/ENGINEER so it uses them to plot the contents of the current object. These bound functions are called with arguments that describe the actual primitive to be plotted.

Please note that this functionality has the following restrictions:

1. Shaded models plotting will not be sent to the output. This applies both to plots of models generated from 3D model modes and shaded drawing views.
2. The contents of OLE objects embedded in drawings will not be included in the output.
3. Text contents provided to the implementation may include special Pro/ENGINEER symbols. It is the responsibility of the application to transform this text into something that can be displayed correctly in the output format.
4. Newly created plotter formats registered with this method will not be accessible from the Print dialogs, or from the Pro/TOOLKIT function ProPrintExecute(). In order to export the model with this custom plot driver, you must use prointerface_2d().

Each plot format you define has a unique name. The function **prointerface_create()** simply declares the name you will use; that name is referenced in calls to the other functions in this chapter.

The function **prointerface_object_set()** tells Pro/ENGINEER the types of object for which your plot format can be used. You supply a list of file extensions (“PRT”, “DRW”, and so on) that define the types of object you want to be able to plot.

The function **prointerface_load_function()** binds your function to a specified plot primitive type. The types are defined in the include file *pro_intf.h*, and are shown in the following table.

The table also shows the names for the bound functions under which their arguments are documented. For example, if you are binding a function for the primitive PROINTF_LINE (to plot a line), the arguments it will be called with are described under the function **user_intf_line()**, even though your function might have a different name.

Primitive Type	Bound Function Name
PROINTF_LINE	user_intf_line()
PROINTF_TEXT	user_intf_text()
PROINTF_CIRCLE	user_intf_circle()
PROINTF_ARC	user_intf_arc()
PROINTF_POLYLINE	user_intf_polyline()
PROINTF_FILLED_POLY	user_intf_filled_poly()

The function **prointerface_2d()** is used to invoke the user-defined plot on the current object.

Example 1: Sample Plot Driver Program

The following example code demonstrates how to use the customized plot driver functions.

```
/*----- Pro/Toolkit Includes -----*/
#include <ProToolkit.h>
#include "ProMenu.h"
#include <PTApplsUnicodeUtils.h>
#include <ProMessage.h>
/*----- Pro/Develop Includes -----*/
#include "prodevelop.h"
#include "pro_intf.h"
#include "prodtl_str.h"
```

```

/*----- Application Includes -----*/
#include <TestError.h>
#include <string.h>

/*----- Function Prototypes -----*/
int user_Demo_Plot();
void user_demo_line ();
void user_demo_text ();
void user_demo_circle ();
void user_demo_arc ();
void user_demo_polyline ();
void user_demo_filled_poly ();

/*----- External Data -----*/
extern int UserMenuDeleteAndPop();

/*----- Global Data -----*/
static FILE *demo_plot_file;

/*=====
Function : UserPlotSetup()
Purpose  : Set up the Customized Plot Menu
\=====*/
int UserPlotSetup()
{
    int menu_id, action, status;
    ProMdl model;
    static char type[4], null[1], *type_list[] = {type, null};
/*-----*\
    Declare the plot format "demo".
\-----*/
    prointerface_create ("demo");
/*-----*\
    Make "demo" valid for Drawing mode.
\-----*/
    strcpy (type, "DRW");
    strcpy (null, "");
    prointerface_object_set ("demo", type_list);
/*-----*\
    Bind the functions user_demo*() to the primitives for
    the format "demo".
\-----*/
    prointerface_load_function ("demo", PROINTF_LINE,
        user_demo_line);
    prointerface_load_function ("demo", PROINTF_TEXT,
        user_demo_text);
    prointerface_load_function ("demo", PROINTF_CIRCLE,
        user_demo_circle);
    prointerface_load_function ("demo", PROINTF_ARC,
        user_demo_arc);
    prointerface_load_function ("demo", PROINTF_POLYLINE,

```

```

        user_demo_polyline);
prointerface_load_function ("demo", PROINTF_FILLED_POLY,
        user_demo_filled_poly);

status = ProMenuFileRegister("UGPLOT", "ugplot.mnu", &menu_id);
ERROR_CHECK( "UserPlotSetup", "ProMenuFileRegister", status );

status = ProMenubuttonActionSet("UGPLOT", "-Plot Demo",
        user_Demo_Plot, NULL, 0);
ERROR_CHECK( "UserPlotSetup", "ProMenubuttonActionSet", status );

status = ProMenubuttonActionSet("UGPLOT", "-Done/Return",
        UserMenuDeleteAndPop, NULL, 0);
ERROR_CHECK( "UserPlotSetup", "ProMenubuttonActionSet", status );

status = ProMenubuttonActionSet("UGPLOT", "UGPLOT",
        UserMenuDeleteAndPop, NULL, 0);
ERROR_CHECK( "UserPlotSetup", "ProMenubuttonActionSet", status );

status = ProMenuCreate(PROMENUTYPE_MAIN, "UGPLOT", &menu_id);
ERROR_CHECK( "UserPlotSetup", "ProMenuCreate", status );

status = ProMenuProcess("UGPLOT", &action);
ERROR_CHECK( "UserPlotSetup", "ProMenuProcess", status );

return(PRO_TK_NO_ERROR);
}
/*=====*\
Command function to invoke a plot of type "demo"
\*=====*/
int user_Demo_Plot()
{
    int            status;
    wchar_t        w_fname[PRODEV_NAME_SIZE];
    char           fname[PRODEV_NAME_SIZE];
    ProFileName msgfil;
/*-----*\
Get the output plot file name.
\*-----*/
    ProStringToWstring( msgfil, "msg_ugfund.txt" );
    status = ProMessageDisplay (msgfil, "USER Demo plot filename [QUIT] :
                                ");
    ERROR_CHECK("user_Demo_Plot", "ProMessageDisplay", status);

    status = ProMessageStringRead (PRODEV_NAME_SIZE, w_fname) ;
    ERROR_CHECK("user_Demo_Plot", "ProMessageStringRead", status);
    if (status != PRO_TK_NO_ERROR)
        return(status);

    ProWstringToString (fname, w_fname);
    demo_plot_file = PTApplsUnicodeFopen (fname, "w");

```

```

/*-----*\
Invoke the plot.
/*-----*/
    status = prointerface_2d ("demo");
    ERROR_CHECK("user_Demo_Plot", "prointerface_2d", (status <=0));
/*-----*\
Close and display the plot file.
/*-----*/
    fclose (demo_plot_file);
    status = ProInfoWindowDisplay (w_fname, NULL, NULL);
    ERROR_CHECK("user_Demo_Plot", "ProInfoWindowDisplay", status);
    return (PRO_TK_NO_ERROR);
}
/*=====*\
Bound function for plotting a LINE in the format "demo".
/*=====*/
void user_demo_line (point1, point2, color)
double point1[3],
        point2[3];
int     color;
{
    ProTKFprintf (demo_plot_file, "LINE..\n");
    ProTKFprintf (demo_plot_file, " point1 = %f, %f\n",
        point1[0], point1[1]);
    ProTKFprintf (demo_plot_file, " point2 = %f, %f\n",
        point2[0], point2[1]);
    ProTKFprintf (demo_plot_file, " color = %d\n", color);
}
/*=====*\
Bound function for plotting a TEXT item in the format
"demo".
/*=====*/
void user_demo_text (point, text, size, angle, slant_angle,
                    width_factor, dummy, color)
double point[3];
wchar_t *text;
double size,
        angle,
        slant_angle,
        width_factor;
int     dummy, color;
{
    char str[PRODTL_LINE_LEN];
    ProTKFprintf (demo_plot_file, "TEXT..\n");
    ProTKFprintf (demo_plot_file, " point = %f, %f\n", point[0],
        point[1]);
    ProTKFprintf (demo_plot_file, " text = %s\n", ProWstringToString
        (str, text));
    ProTKFprintf (demo_plot_file, " size = %f\n", size);
    ProTKFprintf (demo_plot_file, " angle = %f\n", angle);
}

```

```

    ProTKFprintf (demo_plot_file,"  slant = %f\n", slant_angle);
    ProTKFprintf (demo_plot_file,"  width = %f\n", width_factor);
    ProTKFprintf (demo_plot_file,"  color = %d\n", color);
}
/*=====*\
Bound function for plotting a CIRCLE in the format "demo"
\*=====*/
void user_demo_circle (center, radius, color)
double  center[3],
        radius;
int     color;
{
    ProTKFprintf (demo_plot_file, "CIRCLE..\n");
    ProTKFprintf (demo_plot_file,"  center = %f, %f\n", center[0],
        center[1]);
    ProTKFprintf (demo_plot_file,"  radius = %f\n", radius);
    ProTKFprintf (demo_plot_file,"  color = %d\n", color);
}
/*=====*\
Bound function for plotting an ARC in the format "demo"
\*=====*/
void user_demo_arc (center, radius, point1, point2, t0, t1,
        color)
double  center[3],
        radius,
        point1[3],
        point2[3],
        t0,
        t1;
int     color;
{
    ProTKFprintf (demo_plot_file, "ARC..\n");
    ProTKFprintf (demo_plot_file,"  center = %f, %f\n", center[0],
        center[1]);
    ProTKFprintf (demo_plot_file,"  radius = %f\n", radius);
    ProTKFprintf (demo_plot_file,"  point1 = %f, %ff\n", point1[0],
        point1[1]);
    ProTKFprintf (demo_plot_file,"  point2 = %f, %ff\n", point2[0],
        point2[1]);
    ProTKFprintf (demo_plot_file,"  t1      = %f\n", t1);
    ProTKFprintf (demo_plot_file,"  color = %d\n", color);
}
/*=====*\
Bound function for plotting a POLYLINE in the format "demo"
\*=====*/
void user_demo_polyline (n_pts, array, color)
int     n_pts;
double  array[][3];
int     color;
{
    int  p;

```

```

    ProTKFprintf (demo_plot_file, "POLYLINE..\n");
    for (p = 0; p < n_pts; p++)
        ProTKFprintf (demo_plot_file, "    Point %d = %f, %f\n", p,
            array[p][0], array[p][1]);
    ProTKFprintf (demo_plot_file, "    color      = %d\n", color);
}
/*=====*\
Bound function for plotting a FILLED POLYGON in the
format "demo"
\*=====*/
void user_demo_filled_poly (n_pts, array, color)
int      n_pts;
double   array[][3];
int      color;
{
    int  p;

    ProTKFprintf (demo_plot_file, "FILLED POLY..\n");
    for (p = 0; p < n_pts; p++)
        ProTKFprintf (demo_plot_file, "    Point %d = %f, %f\n", p,
            array[p][0], array[p][1]);
    ProTKFprintf (demo_plot_file, "    color      = %d\n", color);
}

```

Element Trees: Principles of Feature Creation

This chapter describes the basic principles of programmatic feature creation that are applicable to all types of feature that can be created in the current version of Pro/TOOLKIT. This chapter also describes how to extract the internal description of features of those feature types in the Pro/TOOLKIT database, and how to modify them.

Topic	Page
Overview of Feature Creation	29 - 2
Feature Inquiry	29 - 26
Feature Redefine	29 - 27
XML Representation of Feature Element Trees	29 - 28

Overview of Feature Creation

This section provides references to additional material on feature creation and an overview of creating features.

References to Feature Creation Data

The creation of specific feature types is dealt with in more detail in the following chapters:

- Element Trees: Datum Features
- Element Trees: Sketched Features
- Production Applications: Manufacturing

This chapter defines and describes the following Pro/TOOLKIT objects:

- *ProElement*
- *ProElmpath*
- *ProValue*

For a definition of the *ProFeature* object, see the chapter 'Core: Features'.

Feature Creation

There are many kinds of feature in Pro/ENGINEER and each one can contain a large and varied amount of information. All this information must be complete and consistent before a feature can be used in regeneration and give rise to geometry. This raises several problems for programmatic creation of features through an API.

It is necessary to build all the information needed by a feature into a data structure before passing that whole structure to Pro/ENGINEER. However, the object-oriented style of Pro/TOOLKIT requires that such a data structure is not directly visible to the application. Therefore, Pro/TOOLKIT defines this structure as a workspace object that can be allocated and filled using special functions for that purpose, but that is not part of the Pro/ENGINEER database.

There are three steps in creating a feature in Pro/ENGINEER:

1. Allocate the workspace structure.
2. Fill the workspace structure.

3. Pass the workspace structure to Pro/ENGINEER to create the feature.

Note: Creating sketched features requires a few more steps. For detailed information, see the chapter Element Trees: Sketched Features.

The procedure described above allows a feature of arbitrary complexity to be built up in a sequence of manageable steps, with the maximum of error checking along the way.

Although it is not yet possible to create all feature types using Pro/TOOLKIT, the workspace structure must be capable of defining any feature type so the range of features can be extended without affecting the techniques already in use. For this reason, the workspace structure for feature creation takes the form of a tree of data elements. This has the advantage of being simple for simple features, yet is flexible enough to provide for all possible feature types without introducing new principles.

The root and branch points in the tree are called “elements,” and the complete tree is called the “feature element tree.” Each element is modeled by the object *ProElement*, which is a pointer to an opaque structure.

The feature element tree contains all the information required to define the feature. This includes the following:

- All options and attributes, such as the material side and depth type for an extrusion or slot, placement method for a hole, and so on.
- All references to existing geometry items, such as placement references, “up to” surfaces, sketching planes, and so on.
- References to Sketcher models used for sections in the feature.
- All dimension values.

Note: Because Pro/TOOLKIT is the same toolkit used to build Pro/ENGINEER, improvements to Pro/ENGINEER may require the definition of the element tree to be altered for some features. PTC will make every effort to maintain upward compatibility. However, there may be cases where the old application will *not* run with the new version of Pro/ENGINEER, unless you rewrite the application's code to conform to the new definition of the feature tree.

Note that although the values of dimensions used by the feature are in the element tree, there are no descriptions of, or references to, the dimension objects themselves. The only exception is as follows: for an element tree for a feature already in the Pro/ENGINEER database, you can ask the identifier of the dimension used for a particular element using the function **ProFeatureElemDimensionIdGet()**. This is explained in detail in the section Feature Inquiry. For more general functions that access dimensions, see the chapter ‘Core: Relations’.

Each element in the tree is assigned an element ID, which is really a description of the role it is playing in this feature—the kind of information it is supplying. It is called an element ID because no two elements at the same level in the tree will have the same identifier, unless they belong to an array element, so the element ID also acts as a unique identifier.

The possible element roles are values in an enumerated type called *ProElemId*, declared in *ProElemId.h*. Example values are as follows:

- PRO_E_FEATURE_TYPE
- PRO_E_FEATURE_FORM
- PRO_E_EXT_DEPTH
- PRO_E_THICKNESS
- PRO_E_4AXIS_PLANE

There are four different element types:

- Single-valued
- Multivalued
- Compound
- Array

A single-valued element can contain various types of value. The simplest is an integer used to define, for example, the type of the feature, or one of the option choices, such as the material side for a thin protrusion. The value can be a wide string (for example, the name of the feature), or a double (for example, the depth of a blind extrusion). If the element defines a reference to an existing geometry item in the solid, its value contains an entire *ProSelection* object so it can refer to anything in an entire assembly.

A multivalued element contains several values of one of these types. Multivalued elements occur at the lowest level of the element tree—the “leaves.” An example is the element with the identifier PRO_E_FIXT_COMPONENTS in a Fixture Setup feature in Pro/NC. That element specifies the components in the assembly that belong to the fixture; in general, there can be any number of such components, so the element contains several component identifiers.

A compound element is one that acts as a branch point in the tree. It does not have a value of its own, but acts as a container for elements further down in the hierarchy.

An array element is also a branch point, but one that contains many child elements of the same element ID. An example of this is the PRO_E_DTMPLN_CONSTRAINTS element in a datum plane feature, which contains an array of elements of type PRO_E_DTMPLN_CONSTRAINT (note the singular), each of which is a compound element whose contents describe one of the constraints that determine the position of the datum plane.

The feature element tree enables you to build a complex feature in stages, with only a small set of functions. However, the form of the tree required for a particular feature needs to be clearly defined so you know exactly what elements and values to add, and so Pro/TOOLKIT can check for errors each time you add a new element to the tree.

Pro/TOOLKIT documents the necessary contents of the element tree for each type of feature that can be created programmatically. It does this through two types of description:

- Feature element tree
- Feature element table

The feature element tree defines the structure of the tree, specifying the element ID (or role) for the elements at all levels in the tree, and which elements are optional.

The feature element table defines the following for each of the element IDs in the tree:

- A description of its role in the feature
- The value type it has (that is, whether it is single value or compound; or an array of integer, double, or string)
- The range of values valid for it in this context

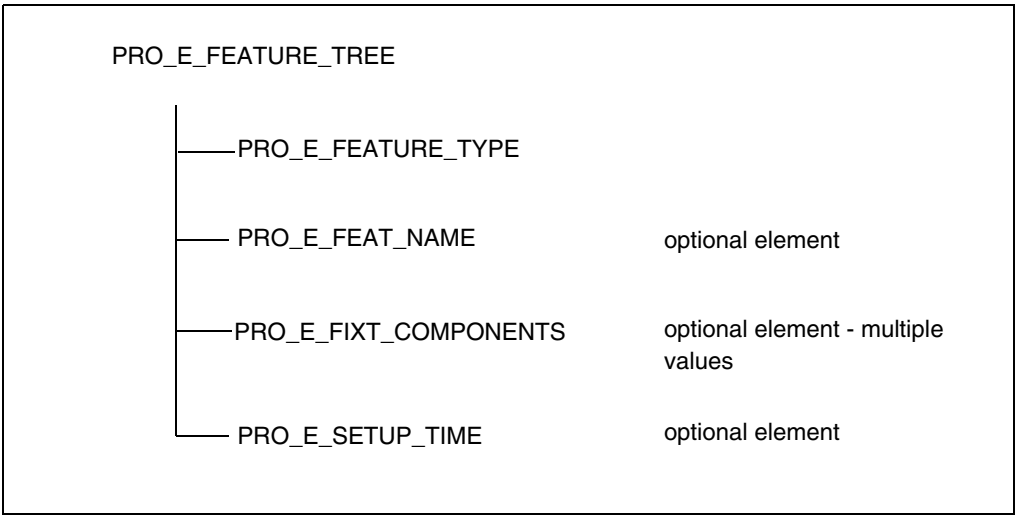
Each type of feature that can be created using Pro/TOOLKIT has its own header file that contains the feature element tree and table, in the form of code comments. The header files for the feature types that can be created in the current version are as follows:

Header File	Feature Type
<i>ProAnalysis.h</i>	External Analysis
<i>ProAsmcomp.h</i>	Assembly component, mechanism connection
<i>ProChamfer.h</i>	Chamfer or corner chamfer
<i>ProDataShareFeat.h</i>	General merge (merge, cutout, inheritance), copy geom., publish geom., shrinkwrap
<i>ProDraft.h</i>	Draft
<i>ProDtmAxis.h</i>	Datum axis
<i>ProDtmCrv.h</i>	Datum curve
<i>ProDtmCsys.h</i>	Datum coordinate system
<i>ProDtmPln.h</i>	Datum plane
<i>ProDtmPnt.h</i>	Datum point
<i>ProExtrude.h</i>	Extruded protrusion, cut, surface, surface trim, thin, sheetmetal wall, or sheetmetal cut
<i>ProFixture.h</i>	Fixture (for Pro/NC)
<i>ProFlatSrf.h</i>	Fill surface or sheetmetal flat wall
<i>ProForeignCurve.h</i>	Foreign datum curve
<i>ProHole.h</i>	Hole
<i>ProMerge.h</i>	Merge
<i>ProMirror.h</i>	Mirror (geometry only)
<i>ProMove.h</i>	Move and copy (geometry only)
<i>ProMfgoper.h</i>	Manufacturing operation
<i>ProNcseq.h</i>	Manufacturing NC sequence
<i>ProProcstep.h</i>	Process step
<i>ProReplace.h</i>	Surface replacement feature
<i>ProRevolve.h</i>	Revolved protrusion, cut, surface, surface trim or thin

Header File	Feature Type
<i>ProRib.h</i>	Rib
<i>ProRound.h</i>	Round
<i>ProShell.h</i>	Shell
<i>ProSmtFlangeWall.h</i>	Sheetmetal flange wall
<i>ProSmtFlatWall.h</i>	Sheetmetal flat wall
<i>ProSolidify.h</i>	Solidify
<i>ProStdSection.h</i>	Standard section
<i>ProSweep.h</i>	Simple swept protrusion, cut
<i>ProThicken.h</i>	Thicken
<i>ProTrim.h</i>	Trim
<i>ProWcell.h</i>	Manufacturing workcell

The feature element tree for a Fixture Setup feature, taken from *ProFixture.h*, looks like this:

Figure 29-1: Fixture Setup Feature Element Tree



The corresponding table of feature elements is as follows:

Element ID	Element Name	Data Type	Valid Values
PRO_E_FEATURE_TYPE	Feature type	INT	PRO_FEAT_FIXSETUP

Element ID	Element Name	Data Type	Valid Values
PRO_E_FEAT_NAME	Feature name	WSTRING	Feature name
PRO_E_FIXT_COMPONENTS	Fixture components	INT ARRAY	Fixture component identifiers
PRO_E_SETUP_TIME	Setup time	DOUBLE	Number of minutes

The first two elements are common to all features. The root of a feature tree is always a compound element with the identifier PRO_E_FEATURE_TREE. The first element within the root always specifies the feature type; it is a single-valued element with the element ID PRO_E_FEAT_TYPE, whose value is one of the integers in the list of feature types in *ProFeatType.h*. In this case, the element table shows that the value must be PRO_FEAT_FIXSETUP.

The next element in a fixture setup gives the name of the feature; its element ID is PRO_E_FEAT_NAME, and it contains a single wide string. The element tree shows that this is optional.

The PRO_E_FIXT_COMPONENTS is a multivalued element, with the value type integer, which contains the identifiers of the assembly components that belong to the fixture.

The last element in a fixture setup is PRO_E_SETUP_TIME, which contains a double.

As you build the elements into the workspace element tree, Pro/TOOLKIT checks the correctness of their types against the structure described by the element tree and table in the corresponding header file. This makes it easy to diagnose errors when you are creating features. The geometrical correctness is checked only when you try to create the feature in the Pro/ENGINEER database.

The following sections of this chapter describe the functions used to build an element tree and create a feature. The sections are as follows:

- Feature Element Values—Introduces the object *ProValue*, used to represent the value of a feature element.
- Feature Element Paths—Introduces the object *ProElemPath*, used to describe the location of an element in an element tree.
- Feature Elements—Introduces the *ProElement* functions used to build and analyze an element tree.

- Feature Element Diagnostics—Introduces the *ProElementDiagnostics* functions used to obtain the diagnostics for a feature element.
- Calling ProFeatureCreate()—Describes the **ProFeatureCreate()** function in detail.
- Example of Complete Feature Creation—Shows how to use functions from the other sections to perform all the steps needed to create a feature.

Feature Element Values

Functions introduced:

- **ProValueAlloc()**
- **ProValueDataGet()**
- **ProValueDataSet()**
- **ProValueFree()**
- **ProWstringArrayToValueArray()**
- **ProValueArrayToWstringArray()**
- **ProValuedataTransformGet()**
- **ProValuedataTransformSet()**

The object *ProValue* is an opaque workspace handle used to contain the value of a feature element. It is the output of the functions **ProElementValueGet()** and **ProElementValuesGet()**, which read the values of a feature element, and is the input to **ProElementValueSet()** and **ProElementValuesSet()**.

You can access the contents of a *ProValue* object by translating it into an object of type *ProValueData*, which is declared as a visible data structure. The declaration is as follows:

```
typedef struct pro_value_data
{
    ProValueDataType    type;
    union
    {
        int             i;      /* integer */
        double          d;      /* double */
        void             *p;     /* pointer or reference */
    }
}
```

```

char          *s;      /* string */
wchar_t       *w;      /* wchar_t */
ProSelection  r;      /* selection */
} v;
} ProValueData;

```

```

typedef enum pro_value_data_type
{
    PRO_VALUE_TYPE_INT           = 1,
    PRO_VALUE_TYPE_DOUBLE,
    PRO_VALUE_TYPE_POINTER,
    PRO_VALUE_TYPE_STRING,
    PRO_VALUE_TYPE_WSTRING,
    PRO_VALUE_TYPE_SELECTION
} ProValueDataType;

```

ProValueData is simply a holder for data values of many different types.

Note: From Pro/ENGINEER Wildfire 2.0 onwards, elements with multiple values, for example, `PRO_E_FIXT_COMPONENTS`, are deprecated. In subsequent releases, these elements will be superseded by reference elements or single-value, type-specific elements. Use the function **ProElementValuetypeGet()** to determine the type of the element.

To access reference elements use the functions **ProElementReferencesGet()** or **ProElementReferencesSet()**. To access single-value, type-specific elements, use the functions **ProElement<type>Get()** or **ProElement<type>Set()**, such as, **ProElementDoubleGet()**, **ProElementIntegerGet()** and so on.

The functions in this section access the contents of a *ProValue* through the *ProValueData* object.

The function **ProValueDataGet()** provides the *ProValueData* object for the specified *ProValue* object.

The function **ProValueAlloc()** allocates a new *ProValue* in memory, as the first step towards setting the value of a feature element.

The function **ProValueDataSet()** sets the value of a *ProValue* object using the contents of a *ProValueData* structure.

The function **ProValueFree()** frees a *ProValue* object in memory.

The function **ProWstringArrayToValueArray()** provides a convenient way to allocate and fill an array of *ProValue* structures that all contain wide string values.

The function **ProValueArrayToWstringArray()** performs the reverse translation, allocating and filling an array of wide strings. In both cases, the output array is an expandable array, so you should release the memory using **ProArrayFree()**.

The transform member of the union *ProValueData* is declared as `double**`. It must be passed a `double[][]` (a *ProMatrix* structure). The utility functions **ProValuedataTransformGet()** and **ProValuedataTransformSet()** specify how to assign the *ProValueData* in order to access the matrix correctly.

Feature Element Paths

Functions introduced:

- **ProElempathAlloc()**
- **ProElempathFree()**
- **ProElempathDataSet()**
- **ProElempathDataGet()**
- **ProElempathCopy()**
- **ProElempathCompare()**
- **ProElempathSizeGet()**

An element path is used to describe the location of an element in an element tree. It is used by some of the *ProElement* functions as a convenient way to refer to elements already in a tree.

The object *ProElempath* is declared as an opaque pointer. It contains a description of the path from the root of the tree down to the element referred to. At most levels in the tree hierarchy, the relevant path member is the element ID of the element (which is unique at that level). When the path steps from an array element to one of its member arrays, the element path instead contains the array index of that element.

To be able to set the value of a *ProElempath*, Pro/TOOLKIT provides a structure called *ProElempathItem* that can describe an element ID, or the index into an array element. An array of *ProElempathItem* structures is therefore a visible equivalent to the opaque contents of *ProElempath*.

The declaration of *ProElempathItem* is as follows:

```
typedef struct path
{
    ProElempathItemtype type;
    union
    {
        int elem_id;
        int elem_index;
    } path_item;
} ProElempathItem;

typedef enum
{
    PRO_ELEM_PATH_ITEM_TYPE_ID,
    PRO_ELEM_PATH_ITEM_TYPE_INDEX
} ProElempathItemtype;
```

The object *ProElempath*, the structure *ProElempathItem*, and all the functions in this section are declared in the header file *ProElempath.h*.

The function **ProElempathAlloc()** allocates a new empty *ProElempath* object, whereas **ProElempathFree()** frees a *ProElempath*.

The function **ProElempathDataSet()** enables you to set the contents of a *ProElempath* by copying from an array of *ProElempathItem* structures.

The function **ProElempathDataGet()** reads the contents of a *ProElempath* into an array of *ProElempathItem* structures. The array is an expandable array that must be allocated by a call to **ProArrayAlloc()** before you call the function.

The function **ProElemPathCopy()** copies the contents of one *ProElemPath* object into another. The output object is allocated by the function.

The function **ProElemPathCompare()** tells you whether two *ProElemPath* objects refer to the same element.

The function **ProElemPathSizeGet()** tells you the length of the element path contained in a *ProElemPath* object.

Feature Elements

Functions introduced:

- **ProElementAlloc()**
- **ProElementFree()**
- **ProElementIdGet()**
- **ProElementIdSet()**
- **ProElemtreeElementGet()**
- **ProElemtreeElementAdd()**
- **ProElemtreeElementRemove()**
- **ProElementIsMultival()**
- **ProElementIsCompound()**
- **ProElementIsArray()**
- **ProElementChildrenGet()**
- **ProElementChildrenSet()**
- **ProElementArraySet()**
- **ProElementArrayGet()**
- **ProElementArrayCount()**
- **ProElemtreeElementVisit()**

The function **ProElementAlloc()** allocates a new *ProElement* object with a specified element ID. The function **ProElementFree()** frees a *ProElement*.

The function **ProElementIdGet()** outputs the element ID of a specified *ProElement*. The function **ProElementIdSet()** enables you to set the element ID of a specified *ProElement*.

The function **ProElemtreeElementGet()** enables you to read a specified element in a tree. The inputs are the root of the tree, specified as a *ProElement* object, and the path to the element, specified by a *ProElemPath*. The output is a *ProElement* object.

The function **ProElemtreeElementAdd()** adds a new element to the specified element tree. The inputs are the *ProElement* for the tree root, the *ProElemPath* to the new element, and the *ProElement* for the new element.

The function **ProElemtreeElementRemove()** removes an element from a specified tree and path. It outputs a *ProElement* for the element removed.

The functions **ProElementIsMultival()**, **ProElementIsCompound()**, and **ProElementIsArray()** tell you the type of a specified element in a tree. See the section Overview of Feature Creation for an explanation of the types.

The function **ProElementChildrenGet()** provides an expandable array of *ProElement* objects for the children of the specified compound element in a tree. The array must be allocated using **ProArrayAlloc()** before you call this function. The function **ProElementChildrenSet()** adds a set of elements, specified by an expandable array of *ProElement* objects, as the children of the specified element in a tree.

The function **ProElementArraySet()** adds an expandable array of *ProElement* objects as the members of a specified array element in an element tree.

The function **ProElementArrayGet()** fills an expandable *ProElement* array with the members of an array element in an element tree. The function **ProElementArrayCount()** tells you how many members are in an array element in the specified element tree.

The function **ProElemtreeElementVisit()** visits the elements that are members of the specified array element in an element tree.

Access to ProElement Data

In earlier releases, Pro/TOOLKIT recommended using functions that access the element value(s) as *ProValue* objects. These functions are maintained for compatibility. PTC recommends using the functions in this section to provide greater flexibility for all new development related to any Pro/ENGINEER feature type. Use of these new functions is required for features first supported in Pro/ENGINEER Wildfire 2.0 and later.

Functions introduced:

- **ProElementValuetypeGet()**
- **ProElementReferenceGet()**
- **ProElementReferenceSet()**
- **ProElementReferencesGet()**
- **ProElementReferencesSet()**
- **ProElementIntegerGet()**
- **ProElementIntegerSet()**
- **ProElementDoubleGet()**
- **ProElementDoubleSet()**
- **ProElementDecimalsGet()**
- **ProElementDecimalsSet()**
- **ProElementwstropsAlloc()**
- **ProElementwstropsExpressionsSet()**
- **ProElementwstropsSignoptionsSet()**
- **ProElementwstropsFree()**
- **ProElementWstringGet()**
- **ProElementWstringSet()**
- **ProElementStringGet()**
- **ProElementStringSet()**
- **ProElementSpecialvalueGet()**
- **ProElementSpecialvalueSet()**
- **ProElementBooleanGet()**
- **ProElementBooleanSet()**
- **ProElementTransformGet()**
- **ProElementTransformSet()**

These functions are the preferred method of accessing element values information over **ProElementValueGet()** or **ProElementValueSet()** and **ProValueDataGet()** or **ProValueDataSet()**.

The function **ProElementValuetypeGet()** returns the nominal value type for the element.

The functions **ProElementReferenceGet()** and **ProElementReferenceSet()** returns and sets a single reference value for the element.

The function **ProElementReferencesGet()** returns an array of reference values for the element. The function outputs a reference array, which is a ProArray. Free this output using **ProReferencearrayFree()**.

The function **ProElementReferencesSet()** sets the multiple reference values for the element.

The function **ProElementIntegerGet()** returns an integer value representation for the element. The function **ProElementIntegerSet()** sets the integer value for the element.

The function **ProElementDoubleGet()** returns a double value representation for the element. The function **ProElementDoubleSet()** sets the double value for the element.

The function **ProElementDecimalsGet()** obtains the number of decimal places to be used for the double value of an element in the feature.

The function **ProElementDecimalsSet()** assigns the number of decimals to be used for the double value of an element in the feature. The double value is used in the feature dimension related to this element.

Note: Use the function **ProElementDecimalsSet()** before using the function **ProElementDoubleSet()** to ensure that the double value is assigned with the correct number of decimal places.

The following functions show how options are constructed and set for **ProElement*Get()** functions.

The function **ProElementwstropsAlloc()** allocates the options used to retrieve wide string values.

The function **ProElementwstropsExpressionsSet()** sets the options to retrieve values as expressions or relations, if they exist, instead of a string representations of the actual value. This function is applicable to nominal double and integer value elements only.

The function **ProElementwstropsSignoptionsSet()** sets the options to retrieve values with special sign formatting (+/-), etc. This function is applicable to nominal double and integer value elements only.

The function **ProElementwstropsFree()** frees the options used to retrieve string values.

The function **ProElementWstringGet()** returns a string value representation for the element. The function allows, optionally, a **ProElementWstrOpts()** structure that dictates the format of the output. Use the function **ProWstringFree()** to free this string.

The function **ProElementWstringSet()** sets the string value for the element.

The function **ProElementStringGet()** returns an ASCII string value representation for the element. The inputs for this function are the element and options for how the string should be obtained. The output is the ASCII string value. Free this string using **ProStringFree()**.

The function **ProElementStringSet()** sets the ASCII string value for the element.

The function **ProElementSpecialvalueGet()** returns the pointer representation for the element and the function **ProElementSpecialvalueSet()** sets the pointer representation for the element.

The function **ProElementBooleanGet()** returns the boolean representation for the element and the function **ProElementBooleanSet()** sets the boolean value for the element.

The function **ProElementTransformGet()** returns the transform representation for the element and the function **ProElementTransformSet()** sets the transform value for the element.

Feature Element Diagnostics

Functions Introduced:

- **ProElemtreeVerify()**
- **ProElementDiagnosticsCollect()**
- **ProElemdiagnosticProarrayFree()**
- **ProElemdiagnosticSeverityGet()**
- **ProElemdiagnosticMessageGet()**
- **ProElemdiagnosticFree()**
- **ProReferenceDiagnosticsCollect()**

The function **ProElemtreeVerify()** verifies the contents of the element tree. It parses the entire element tree and determines, where possible, if any of the elements have potentially incorrect values. You can obtain the results of this verification by using one of the following paths:

- Use the function **ProElementDiagnosticsCollect()** to identify if any diagnostic items were added to certain elements in the tree.
- Use the function **ProReferenceStatusGet()** to see if certain references in the tree show a missing or error status.

The function **ProElemtreeVerify()** is primarily intended for new element trees created in order to create a new feature in the model. For feature element trees extracted using the function **ProFeatureElemtreeExtract()**, the diagnostics information is embedded based upon the owner model state at the time of extraction.

The function **ProElementDiagnosticsCollect()** obtains a *ProArray* of diagnostics for the element. These diagnostics include warnings and errors regarding the value of the element within the context of the feature and remainder of the element tree. Use the function **ProElemdiagnosticProarrayFree()** to free the *ProArray* of diagnostic items.

The function **ProElemdiagnosticSeverityGet()** returns the severity of the diagnostic item for the element.

The function **ProElemdiagnosticMessageGet()** obtains the message describing the diagnostic item for the element. This message is in the user's localized language.

The function **ProElemdiagnosticFree()** frees the diagnostic item for the element.

The function **ProReferenceDiagnosticsCollect()** obtains a *ProArray* of diagnostics for the reference element. These diagnostics include warnings and errors regarding the state of the reference element within the context of the feature.

Calling ProFeatureCreate()

Function introduced:

- **ProFeatureCreate()**

The syntax of **ProFeatureCreate()** is as follows:

```
ProError ProFeatureCreate (
```



```

ProSelection          model,          /* (In)  The part on which the
                                         feature is being
                                         created. If the feature
                                         is created in an
                                         assembly, you must
                                         specify the component
                                         path. */
ProElement            elemtree,       /* (In)  The element tree. */
ProFeatureCreateOptions options[],    /* (In)  An array of user
                                         options. */
int                   num_opts,       /* (In)  The number of options
                                         in the options array. */
ProFeature            *p_feature,     /* (Out) The feature handle. */
ProErrorlist          *p_errors      /* (Out) The list of errors. */
)

```

The first input argument to **ProFeatureCreate()** identifies the solid that is to contain the new feature. This is expressed in the form of a *ProSelection* object.

Note: The *ProSelection* object input to **ProFeatureCreate()**, and all the *ProSelection* objects assigned to elements in the feature element tree, must all refer to the *same* root assembly.

The second input argument is the *ProElement* object that forms the root of the feature element tree.

The next two inputs are an array of creation options and the size of the array. The creation options specify what **ProFeatureCreate()** should do if the element tree is incomplete, or if the geometry cannot be constructed. Each option is one of the following values of the enumerated type *ProFeatureCreateOptions*:

- PRO_FEAT_CR_NO_OPTS—No options were chosen.
- PRO_FEAT_CR_DEFINE_MISS_ELEMS—Prompt the user to complete the feature if any elements are missing.
- PRO_FEAT_CR_INCOMPLETE_FEAT—Create the feature, even if some elements are missing. The feature will appear in the Pro/ENGINEER feature list and model tree, but will not be used in regeneration.
- PRO_FEAT_CR_FIX_MODEL_ON_FAIL—If the feature geometry cannot be constructed, prompt the user to fix the problem.

If no options are needed, you can set the array to NULL, and the size to zero. If you do not specify any options, **ProFeatureCreate()** fails and reports errors if the element tree is incomplete, or if the geometry cannot be constructed.

To check whether a feature is incomplete, use the function **ProFeatureIsIncomplete()**.

The next argument is an output that provides a *ProFeature* object that identifies the newly created feature.

The final argument is an output that reports errors found in the feature element tree. This is designed as an aid to application developers because it is reporting errors that occur only as a result of incorrect application code; it is not designed as a way of reporting errors to the Pro/ENGINEER user. The errors are written to a structure called *ProErrorlist* whose declaration, in *ProItemerr.h*, is as follows:

```
typedef struct
{
    ProItemerror      *error_list;
    int               error_number;
} ProErrorlist;

typedef struct
{
    int               err_item_id;
    ProErritemType    err_item_type;
    ProError          error;
} ProItemerror;

typedef enum ProErritemTypes
{
    PRO_ERRITEM_NONE = -1,
    PRO_ERRITEM_FEATELEM = 1
} ProErritemType;
```

The field *error_list* is an array of all the errors in the feature element tree found by **ProFeatureCreate()**. Each error has a value expressed in terms of the standard Pro/TOOLKIT error type *ProError*, and can refer to an element of a specified identifier, or be a more general error.

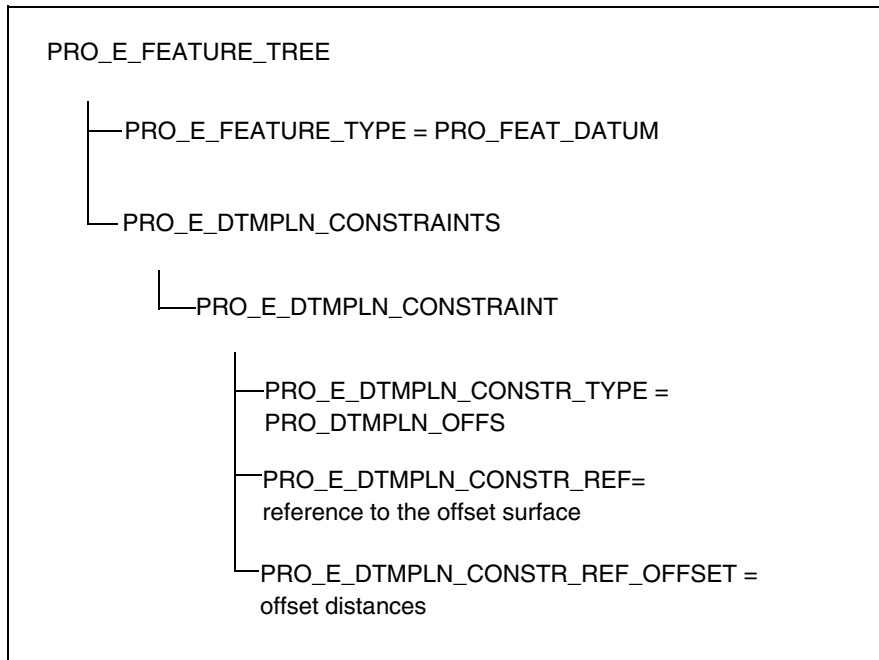
Note:

- There are many useful utilities located in the Pro/TOOLKIT sample code under the Pro/TOOLKIT loadpoint. Utilities such as *ProUtilElementtreePrint()* are particularly useful when building and debugging element trees.
- From Pro/ENGINEER Wildfire 5.0 onward, the function **ProFeatureCreate()** has been deprecated. Instead, use the function **ProFeatureWithoptionsCreate()** with its input argument *flags* set to `PRO_REGEN_NO_FLAGS` for the equivalent behavior. **ProFeatureWithoptionsCreate()** has been described in detail in the “Manipulating Features based on Regeneration Flags” section in the “Core: Features” chapter.

Example of Complete Feature Creation

This section illustrates all the techniques explained so far in this chapter by showing the code required to create a datum plane in a part or an assembly. The datum plane created here is offset from a plane surface, and therefore has the following element tree:

Figure 29-2: Example Element Tree: Offset Datum Plane



(The full element tree for datum plane features is described in the chapter ‘Element Trees: Datum Features’.)

In the code examples that follow, no checks of function return statuses are shown, for clarity. No variable declarations are shown, but the style of the code samples should make these self-explanatory.

The example assumes that the datum plane is being created by a utility function that has two inputs describing the offset surface as a *ProSelection* relative to the current part or assembly, and the offset distance, as follows:

```
int ProDatumPlaneCreate (ProSelection offset_surface,
                        double offset_dist);
```

The first step is to create the element that forms the root of the element tree. This element has the element ID `PRO_E_FEATURE_TREE` but has no value, so it can be created simply by a call to **ProElementAlloc()**:

```
/*-----*\
  Create the root of the element tree.
\*-----*/
```

```
ProElementAlloc (PRO_E_FEATURE_TREE, &elem_tree);
```

The first element inside the root is, as for all features, the feature type. Its ID is PRO_E_FEATURE_TYPE, and it has the single value PRO_FEAT_DATUM. To set the value, you must first create a *ProValue* object of type integer.

```
/*-----*\
    Allocate the feature type element.
\*-----*/
ProElementAlloc (PRO_E_FEATURE_TYPE, &elem_ftype);
/*-----*\
    Set the value of the feature type element.
\*-----*/
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_FEAT_DATUM;
ProValueAlloc (&value);
ProValueDataSet (value, &value_data);
ProElementValueSet (elem_ftype, value);
/*-----*\
    Add the feature type element as a child of the tree root.
\*-----*/
ProElemtreeElementAdd (elem_tree, NULL, elem_ftype);
```

The next element is simply the holder for the datum plane constraints, and this in turn contains a single constraint element (to be used for the offset constraint).

```
/*-----*\
    Add a PRO_E_DTMLPLN_CONSTRAINTS element to the root of the
    tree.
\*-----*/
ProElementAlloc (PRO_E_DTMLPLN_CONSTRAINTS, &elem_consts);
ProElemtreeElementAdd (elem_tree, NULL, elem_consts);
/*-----*\
    Add a PRO_E_DTMLPLN_CONSTRAINT element to the constraints
    element.
\*-----*/
ProElementAlloc (PRO_E_DTMLPLN_CONSTRAINT, &elem_offset);
ProElemtreeElementAdd (elem_consts, NULL, elem_offset);
```

Inside the single constraint element, add an element of type PRO_E_DTMLPLN_CONSTR_TYPE that specifies the constraint type to be PRO_DTMLPLN_OFFS.

```
/*-----*\
    Allocate the constraint type element.
\*-----*/
ProElementAlloc (PRO_E_DTMLPLN_CONSTR_TYPE,
                &elem_const_type);
/*-----*\
```

```

    Set its value to be PRO_DTMLN_OFFS.
/*-----*/
    value_data.type = PRO_VALUE_TYPE_INT;
    value_data.v.i = PRO_DTMLN_OFFS;
    ProValueAlloc (&value);
    ProValueDataSet (value, &value_data);
    ProElementValueSet (elem_const_type, value);
/*-----*/
    Add it as a member of the constraint element.
/*-----*/
    ProElemtreeElementAdd (elem_offset, NULL, elem_const_type);

```

Also in the constraint element, you need an element to identify the reference plane surface, PRO_E_DTMLN_CONSTR_REF, with value type *ProSelection*.

```

/*-----*/
    Allocate the offset reference element.
/*-----*/
    ProElementAlloc (PRO_E_DTMLN_CONSTR_REF, &elem_offset_ref);
/*-----*/
    Set its value to be the ProSelection for the offset
    reference surface.
/*-----*/
    value_data.type = PRO_VALUE_TYPE_SELECTION;
    value_data.v.r = offset_surface;
    ProValueAlloc (&value);
    ProValueDataSet (value, &value_data);
    ProElementValueSet (elem_offset_ref, value);
/*-----*/
    Add it to the constraint element.
/*-----*/
    ProElemtreeElementAdd (elem_offset, NULL, elem_offset_ref);

```

Finally, you need an element of type PRO_E_DTMLN_CONSTR_REF_OFFSET to contain the double value of the offset distance.

```

/*-----*/
    Allocate the offset distance element.
/*-----*/
    ProElementAlloc (PRO_E_DTMLN_CONSTR_REF_OFFSET,
                    &elem_offset_dist);
/*-----*/
    Set its value to be the offset distance.
/*-----*/
    value_data.type = PRO_VALUE_TYPE_DOUBLE;
    value_data.v.d = offset_dist;
    ProValueAlloc (&value);
    ProValueDataSet (value, &value_data);
    ProElementValueSet (elem_offset_dist, value);

```

```
/*-----*\
    Add it to the constraint element.
\*-----*/
ProElemtreeElementAdd (elem_offset, NULL, elem_offset_dist);
```

The element tree is complete.

The next step is to set up a *ProSelection* object that refers to the solid in which you will create the datum plane.

You have the information about the context, in the form of the *ProSelection* object, for the offset surface that was an input to the function you are writing. The component path you need is the same one used to specify that surface. The solid to contain the new feature is the one that owns the offset surface. Therefore, you can build the *ProSelection* object for it as follows:

```
/*-----*\
    Get the component path for the offset surface.
\*-----*/
ProSelectionAsmcomppathGet (offset_surface, &comppath);
/*-----*\
    Get the model item for the offset surface.
\*-----*/
ProSelectionModelitemGet (offset_surface, &surf_modelitem);
/*-----*\
    Make a ProModelitem that refers to the owner of the offset
    surface.
\*-----*/
ProMdlToModelitem (surf_modelitem.owner, &model_modelitem);
/*-----*\
    Make a ProSelection for the solid that will contain the
    new feature.
\*-----*/
ProSelectionAlloc (&comppath, &model_modelitem, &featsel);
```

If the offset surface belongs to a part in a current assembly, and your function is required to add the datum plane not to the part but to the assembly, the code above would be replaced by this:

```
/*-----*\
    Get the component path for the offset surface.
\*-----*/
ProSelectionAsmcomppathGet (offset_surface, &comppath);
/*-----*\
    Make a ProModelitem that refers to the root of the assembly.
\*-----*/
ProMdlToModelitem (comppath.owner, &model_modelitem);
/*-----*\
    Make a ProSelection for the root of the assembly.
\*-----*/
```

```
ProSelectionAlloc (NULL, &model_modelitem, &featsel);
```

Finally, call **ProFeatureCreate()**.

```
/*-----*\
   Create the datum plane feature.
\*-----*/
ProFeatureCreate (featsel, elem_tree, NULL, 0, &feature,
                  &errors);
```

Feature Inquiry

Functions introduced:

- **ProFeatureElemtreeCreate()**
- **ProFeatureElemtreeExtract()**
- **ProFeatureElemtreeFree()**
- **ProFeatureElemValueGet()**
- **ProFeatureElemValuesGet()**
- **ProFeatureElemDimensionIdGet()**
- **ProFeatureElemIsVisible()**
- **ProFeatureElemIsIncomplete()**
- **ProFeatureElemIsIncomplete()**

This section describes how to extract the element tree from a feature and analyze it. To find out how to inquire about the feature as a whole and its role in the owning solid, see the section 'Feature Inquiry' in the chapter 'Core: Features'.

The function **ProFeatureElemtreeCreate()** creates a copy of the feature element tree that describes the contents of a specified feature in the Pro/ENGINEER database. It is applicable only to those feature types that can be created using **ProFeatureCreate()** (as described in the 'Overview of Feature Creation'). The tree can then be analyzed using the read-access functions, such as **ProElement*Get()**, **ProElement*Visit()**, and **ProElementArrayCount()**, described in the sections 'Feature Elements' and 'Feature Element Paths'.

Note: The function **ProFeatureElemtreeCreate()** has been deprecated as it does not provide options to resolve the paths of external references of the feature in case of assemblies. Use **ProFeatureElemtreeExtract()** instead.

The function **ProFeatureElemtreeExtract()** creates a copy of the feature element tree of a specified feature. It also provides options to resolve the paths of external references of the feature in case of assemblies.

Use the function **ProFeatureElemtreeFree()** to free a copy of the feature element tree extracted using **ProFeatureElemtreeCreate()** or **ProFeatureElemtreeExtract()**. The function **ProElementFree()** does not free all of the feature-specific runtime data associated with the element tree, and thus can result in a memory leak for certain features.

Instead of copying the entire element tree to analyze it, you can extract information about particular elements directly from the feature. The remaining functions in this section serve that purpose.

The function **ProFeatureElemValueGet()** provides the value of a single-valued element specified by the *ProFeature* object and a *ProElemPath*. The function **ProFeatureElemValuesGet()** provides the values of a multivalued element in a feature.

The function **ProFeatureElemDimensionIdGet()** gives you the integer identifier of the dimension in the Pro/ENGINEER database used to define the value of the specified single-valued element.

The function **ProFeatureElemIsVisible()** distinguishes elements added to the tree by Pro/ENGINEER for internal reasons only, and are neither defined as needed for creation of that type of feature, nor otherwise documented.

The function **ProFeatureIsIncomplete()** identifies features in the Pro/ENGINEER database whose element trees are still incomplete. Such a feature can arise by using the option **PRO_FEAT_CR_INCOMPLETE_FEAT** when calling **ProFeatureCreate()**, and does not give rise to geometry until completed. If a feature is incomplete, you can find out which element in its tree is at fault using the function **ProFeatureElemIsIncomplete()**. Its input is a *ProFeature* and a *ProElemPath*.

Feature Redefine

Function introduced:

- **ProFeatureRedefine()**

The function **ProFeatureRedefine()** enables you to redefine a feature. It is applicable only to those feature types that can be created using **ProFeatureCreate()**.

To Redefine a Feature

1. Call **ProFeatureElemtreeExtract()** to get a copy of the element tree.
2. Analyze and modify the tree using functions **ProFeatureElem*()**, **ProElement*()**, and **ProElem*path*()**.
3. Call **ProFeatureRedefine()** to replace the old element tree with the new one.

The function **ProFeatureRedefine()** has arguments for the create options and for the resulting element errors, like those for **ProFeatureCreate()**.

XML Representation of Feature Element Trees

Pro/TOOLKIT offers the capability of representing most feature element trees in the XML format. The XML representation of element trees simplifies the procedure of using Pro/TOOLKIT access to Pro/ENGINEER features. The Pro/TOOLKIT API's that access the XML for element trees are capable of converting the XML content to and from an element tree structure (ProElement.)

In Pro/ENGINEER Wildfire 2.0, PTC recommends the use of XML representation for the core Pro/ENGINEER features and for the new sheetmetal wall features. Manufacturing features support is not available in this release.

Introduction to Feature Element Trees in XML

In Pro/TOOLKIT, the access to Pro/ENGINEER features is available using a generic data-structure called element tree. You can use the element tree APIs to query, modify, and redefine the Pro/ENGINEER features. To simplify the access to Pro/ENGINEER feature using Pro/TOOLKIT, the concept of XML representation of the element trees has been introduced in Pro/ENGINEER Wildfire 2.0. The XML representation shows the details of the element tree in text format. The XML files are easy to edit and you can recycle them back to create or to redefine features. All the Pro/TOOLKIT supported features have support for their respective XML representations.

The function **ProElemtreeWrite()** writes the XML version of the element tree in a text file and the function **ProElemtreeFromXMLCreate()** is used to convert the XML version from a text file to the Pro/TOOLKIT native element trees.

Functions Introduced:

- **ProElemtreeWrite()**
- **ProElemtreeFromXMLCreate()**
- **ProXmlerrorlistProarrayFree()**

The function **ProElemtreeWrite()** writes the feature element tree in XML format to the specified file. This function uses the Pro/TOOLKIT native element tree as an input. The type of output required should be specified. Currently only XML format for the output is supported. The only output argument is the name of the output file included with the location and the extension.

The function **ProElemtreeFromXMLCreate()** reads in the XML file and converts into a corresponding Pro/TOOLKIT native element tree. This function takes the name with location of the output XML file as an input and returns the generated element tree as the output.

The created element tree can be used as an input to **ProFeatureCreate** or **ProFeatureRedefine**.

In case of error in execution of this function, an error object is populated and returned. This error object is a ProArray of errors. It contains the information like type of the error, path of the XML file, line number, column number, and the error message. The memory for the error object is allocated by the function is required to be freed in the Pro/TOOLKIT application using **ProXmlerrorlistProarrayFree()**. The function has three levels of error on the basis of the severity warning, general error and fatal error. Upon encountering warning and general errors, the function continues parsing the input XML till the end and collects all the errors in the error object. For a fatal error, the function stops the parsing and returns immediately.

Validation Using XML Schema

XML schemas are used for validation of element tree which is used for feature creation. Element tree in the form of XML representations are validated and passed to **ProFeatureCreate()** and **ProFeatureRedefine()**.

To use the Schema validation, the following config option must be set

```
enable_protk_xml_schema yes
```

The XML Schema defines the structure and data type for XML. The element name is compared with the element specified in schema for validation. In the case of compound elements, validation is done for name, order and number of the sub-elements. The Schema validates the accuracy of data to be passed as it supports the data type of the element. The Schema also checks whether the element is optional or compulsory.

The location of the Schema is

```
<install_dir>/proe/xmlelem/schema/
```

ProTKFeature.xsd contains the representation of different feature element.

ProTKObjects.xsd contains the representation of basic and Pro/TOOLKIT objects.

The schema validation of element tree takes place if the config option enable_protk_xml_schema is set to "yes" before passing XML file to **ProElemtreeFromXMLCreate()**.

If the validation fails, the corresponding error is populated into the error object which can be printed out and can be used to correct the input XML file. The error object contains the line number, column number and the error message as debug information.

The following is an example of Schema for Solidify Feature:

```
<xsd:sequence>
  <xsd:element name="PRO_E_FEATURE_TYPE"
type="StringData"
minOccurs="1" maxOccurs="1"/>
  <xsd:element name="PRO_E_FEATURE_FORM" type="StringData"
minOccurs="1" maxOccurs="1"/>
  <xsd:element name="PRO_E_PATCH_QUILT" type="Selection"
minOccurs="1" maxOccurs="1"/>
  <xsd:element name="PRO_E_PATCH_TYPE" type="IntegerData"
minOccurs="0" maxOccurs="1"/>
  <xsd:element name="PRO_E_PATCH_MATERIAL_SIDE" type="IntegerData"
minOccurs="0" maxOccurs="1"/>
  <xsd:element name="PRO_E_STD_FEATURE_NAME" type="StringData"
minOccurs="0" maxOccurs="1"/>
</xsd:sequence>
<!-- End Solidify -->
```

Following code demonstrate how to use the error object to print the information

```
status = ProElemtreeFromXMLCreate ( input_file, &elemtree, &errors );
if ( ( status == PRO_TK_GENERAL_ERROR ) || ( status ==
PRO_TK_BAD_CONTEXT ) )
{
    printf ("ProElemtreeFromXMLCreate FAILURE ::  %d \n", status );
    status = ProArraySizeGet ( errors, &size_errors );
    printf("Size of the Error Object is %d \n", size_errors);
    printf("Content of the Error Object are \n");
    for ( error_count = 0; error_count < size_errors; error_count++ )
    {
        printf("Error No  = %d\n", error_count);
        printf("\t+ Error Type  = %d\n", errors[error_count].error_type );
        if ( status == PRO_TK_GENERAL_ERROR )
        {
            printf("\t+ Error Path  = %s\n",
                ProWstringToString ( temp_string,
errors[error_count].error_source ));
            printf("\t+ Line  No  = %d\n", errors[error_count].line_number);
            printf("\t+ Column  No  = %d\n",
errors[error_count].column_number);
        }
        else if ( status == PRO_TK_BAD_CONTEXT )
        {
            printf("\t+ Element Sr No  = %d\n",
errors[error_count].line_number);
            printf("\t+ Element ID    = %d\n",
errors[error_count].column_number);
        }
        ProWstringLengthGet ( errors[error_count].message, &string_length );
        temp_message_string = (char *) calloc ( 1, (string_length+1) *
sizeof (char) );
        printf("\t+ Error Message  = %s\n",
            ProWstringToString ( temp_message_string,
errors[error_count].message ));
        free ( temp_message_string );
    }
}
```

The list of features supported for XML schema validation are as follows:

Feature Type	Feature Name
PRO_FEAT_DATUM_AXIS	Datum Axis Feature
PRO_FEAT_DATUM	Datum Plane Feature

Feature Type	Feature Name
PRO_FEAT_CSYS	Datum Coordinate System Feature
PRO_FEAT_CURVE	Datum Curve Feature
PRO_FEAT_DATUM_POINT	Datum Point Feature
PRO_FEAT_CHAMFER	Chamfer Feature
PRO_FEAT_ROUND	Round Feature
PRO_FEAT_DRAFT	Draft Feature
PRO_FEAT_HOLE	Hole Feature
PRO_FEAT_WALL	Sheetmetal Flat or Flange Wall Feature
PRO_FEAT_COMPONENT	Assembly Component Feature
PRO_FEAT_DATUM_QUILT	Surface Merge Feature
PRO_FEAT_PATCH	Solidify Feature of the type Patch
PRO_FEAT_GEN_MERGE	General Merge Feature
PRO_FEAT_SRF_MDL	Mirror or Move Feature
PRO_FEAT_PROTRUSION	Solidify Feature of the type Protrusion
PRO_FEAT_SHELL	Shell Feature
PRO_FEAT_FIRST_FEAT	First Feature
PRO_FEAT_CUT	Solidify Feature of the type Cut
PRO_FEAT_SLOT	Slot Feature
PRO_FEAT_GEOM_COPY	Copy Geometry Feature
PRO_FEAT_RIB	Rib Feature
PRO_FEAT_REPLACE_SURF	Replace Surface Feature

XML Representations for Common Elements

The following section gives details about the XML representation of the common elements. The elements can represent a primitive type e.g. an integer or some complex Pro/TOOLKIT object e.g.

ProReference or **ProCollection**. The purpose of this representation is to enable externalize these objects in a model specific text format. The examples of the XML representations are also provided for easier understanding.

Single Valued Element

A single valued element contains an element of type Integer, Double, String, or Boolean. Its value will be shown as the XML tags. For example:

```
<PRO_E_REV_ANGLE_FROM_VAL type="double">0.00</PRO_E_REV_ANGLE_FROM_VAL>
```

Empty or Optional Element

An empty element in XML does not have any data value, but just the opening and closing XML tags. Empty elements are ignored by ProElemtreeFromXMLCreate(). Unused optional elements in the feature's element tree appear as empty in the XML output from ProElemtreeWrite().

```
<PRO_E_REV_ANGLE_FROM_REF type="selection" />
OR
```

```
<PRO_E_REV_ANGLE_FROM_REF type="selection"></PRO_E_REV_ANGLE_FROM_REF
type>
```

XML Representation for ProSelection or ProReference

This represents the contents of the **ProSelection** or **ProReference** object. If the reference is currently not active or fully available, the reference is included. Consult the function ProReferenceStatusGet() for more information. The reference XML includes details like UV parameters and 3D point for specific elements that require them. The assembly component path will be available for references that include the assembly context.

```
<PRO_E_DPOINT_PLA_CONSTR_REF type="selection">
<PRO_XML_REFERENCE type="reference">
<PRO_XML_REFERENCE_STATUS type=
"int">PRO_REF_NOT_FOUND</PRO_XML_REFERENCE_STATUS>
<PRO_XML_REFERENCE_OWNER type="owner">COMP2.prt</PRO_XML_REFERENCE_OWNER>
<PRO_XML_REFERENCE_ID type="id">46</PRO_XML_REFERENCE_ID>
<PRO_XML_REFERENCE_TYPE type="prototype">PRO_EDGE</PRO_XML_REFERENCE_TYPE>
<PRO_XML_ASMCOMP_PATH comppath="compound">
<PRO_XML_ASMCOMP_PATH_OWNER
type="model">ASM_PNT.asm</PRO_XML_ASMCOMP_PATH_OWNER>
<PRO_XML_ASMCOMP_PATH_ARRAY type="array">
<PRO_XML_ASMCOMP_PATH_ITEM index="1">40</PRO_XML_ASMCOMP_PATH_ITEM>
</PRO_XML_ASMCOMP_PATH_ARRAY>
</PRO_XML_ASMCOMP_PATH>
<PRO_XML_UV_PARAM uv_parameter="array">
<PRO_XML_DOUBLE_VALUE type="u">0.000000</PRO_XML_DOUBLE_VALUE>
<PRO_XML_DOUBLE_VALUE type="v">0.000000</PRO_XML_DOUBLE_VALUE>
</PRO_XML_UV_PARAM>
</PRO_XML_REFERENCE>
```

```
</PRO_E_DPOINT_PLA_CONSTR_REF>
```

XML Representation for ProCollection

This represents the contents of the **ProCollection** object. Collection type, collection instructions, instruction types, references, reference types are some of the contents. These are classified into surface and curve collections. Surface collection represents a set of surfaces in the model governed by rules like seed and boundary. Curve collection represents a set of edges or curves in the model governed by rules like one-by-one, surface-loop etc. The instruction type for the particular collection identifies the rule.

Curve Collection

```
<PRO_E_STD_CURVE_COLLECTION_APPL type="collection">
  <PRO_XML_COLLECTION type="curve">
    <PRO_XML_COLLECTION_INSTRUCTIONS type="array">
      <PRO_XML_COLLECTION_INSTRUCTION type="compound">
        <PRO_XML_COLLECTION_INSTRUCTION_TYPE
type="int">105</PRO_XML_COLLECTION_INSTRUCTION_TYPE>
        <PRO_XML_CRV_COLL_REFS type="array">
          <PRO_XML_CRV_COLL_REF type="selection">
            <PRO_XML_REFERENCE type="reference">
<PRO_XML_REFERENCE_OWNER
type="owner">CRV_COLLECTION_PART.prt</PRO_XML_REFERENCE_OWNER>
<PRO_XML_REFERENCE_ID type="id">703</PRO_XML_REFERENCE_ID>
<PRO_XML_REFERENCE_TYPE type="prototype">PRO_QUILT</PRO_XML_REFERENCE_TYPE>
              </PRO_XML_REFERENCE>
            </PRO_XML_CRV_COLL_REF>
          <PRO_XML_CRV_COLL_REF type="selection">
            <PRO_XML_REFERENCE type="reference">
<PRO_XML_REFERENCE_OWNER
type="owner">CRV_COLLECTION_PART.prt</PRO_XML_REFERENCE_OWNER>
<PRO_XML_REFERENCE_ID type="id">679</PRO_XML_REFERENCE_ID>
<PRO_XML_REFERENCE_TYPE type="prototype">PRO_EDGE</PRO_XML_REFERENCE_TYPE>
              </PRO_XML_REFERENCE>
            </PRO_XML_CRV_COLL_REF>
          </PRO_XML_CRV_COLL_REFS>
        </PRO_XML_COLLECTION_INSTRUCTION>
      </PRO_XML_COLLECTION_INSTRUCTIONS>
    </PRO_XML_COLLECTION>
  </PRO_E_STD_CURVE_COLLECTION_APPL>
```

Surface Collection

```
<PRO_E_STD_SURF_COLLECTION_APPL type="collection">
  <PRO_XML_COLLECTION type="surface">
    <PRO_XML_COLLECTION_INSTRUCTIONS type="array">
      <PRO_XML_COLLECTION_INSTRUCTION type="compound">
```



```

        <PRO_XML_COLLECTION_INSTRUCTION_TYPE
type="int">2</PRO_XML_COLLECTION_INSTRUCTION_TYPE>
        <PRO_XML_SRFCOLL_INCLUDE
type="boolean">1</PRO_XML_SRFCOLL_INCLUDE>
        <PRO_XML_SRFCOLL_REFS type="array">
            <PRO_XML_SRFCOLL_REF type="compound">
                <PRO_XML_SRFCOLL_REFITEM_TYPE
type="int">3</PRO_XML_SRFCOLL_REFITEM_TYPE>
                    <PRO_XML_SRFCOLL_REFITEM type="selection">
                        <PRO_XML_REFERENCE type="reference">
                            <PRO_XML_REFERENCE_OWNER
type="owner">
                                SRF_COLLECTION_PART.prt</PRO_XML_REFERENCE_OWNER>
                                    <PRO_XML_REFERENCE_ID
type="id">17</PRO_XML_REFERENCE_ID>
                                        <PRO_XML_REFERENCE_TYPE
type="prototype">PRO_SURFACE</PRO_XML_REFERENCE_TYPE>
                                            </PRO_XML_REFERENCE>
                                                </PRO_XML_SRFCOLL_REFITEM>
                                                    </PRO_XML_SRFCOLL_REF>
                                                        <PRO_XML_SRFCOLL_REF type="compound">
                                                            <PRO_XML_SRFCOLL_REFITEM_TYPE
type="int">4</PRO_XML_SRFCOLL_REFITEM_TYPE>
                                                                <PRO_XML_SRFCOLL_REFITEM
type="selection">
                                                                    <PRO_XML_REFERENCE
type="reference">
                                                                        <PRO_XML_REFERENCE_OWNER
type="owner">
                                    SRF_COLLECTION_PART.prt</PRO_XML_REFERENCE_OWNER>
                                        <PRO_XML_REFERENCE_ID
type="id">26</PRO_XML_REFERENCE_ID>
                                            <PRO_XML_REFERENCE_TYPE
type="prototype">PRO_SURFACE</PRO_XML_REFERENCE_TYPE>
                                                </PRO_XML_REFERENCE>
                                                    </PRO_XML_SRFCOLL_REFITEM>
                                                        </PRO_XML_SRFCOLL_REF>
                                                            </PRO_XML_SRFCOLL_REFS>
                                                                </PRO_XML_COLLECTION_INSTRUCTION>
                                                                    </PRO_XML_COLLECTION_INSTRUCTIONS>
                                                                        </PRO_XML_COLLECTION>
                                                                            </PRO_E_STD_SURF_COLLECTION_APPL>

```

Pointer Element

Most pointer elements are written with their value appearing as “**”. Elements with this value type are not supported by **ProElemtreeFromXMLCreate()** (this element is not added to the created element tree.) The most common pointer element of this type is the Pro/ENGINEER sketch element PRO_E_SKETCHER.

```
<PRO_E_SKETCHER type="pointer">**</PRO_E_SKETCHER>
```

Compound Element

A compound element represents a collection of different types of sub-elements. Following is an example in XML format.

```
<PRO_E_REV_ANGLE type="compound">
  <PRO_E_REV_ANGLE_FROM type="compound">
    <PRO_E_REV_ANGLE_FROM_TYPE
type="int">262144</PRO_E_REV_ANGLE_FROM_TYPE>
    <PRO_E_REV_ANGLE_FROM_VAL
type="double">0.00</PRO_E_REV_ANGLE_FROM_VAL>
    <PRO_E_REV_ANGLE_FROM_REF type="selection" />
    <PRO_E_REV_ANGLE_FROM_LIMIT
type="int">0</PRO_E_REV_ANGLE_FROM_LIMIT>
  </PRO_E_REV_ANGLE_FROM>
  <PRO_E_REV_ANGLE_TO type="compound">
    <PRO_E_REV_ANGLE_TO_TYPE
type="int">4194304</PRO_E_REV_ANGLE_TO_TYPE>
    <PRO_E_REV_ANGLE_TO_VAL
type="double">120.00</PRO_E_REV_ANGLE_TO_VAL>
    <PRO_E_REV_ANGLE_TO_REF type="selection" />
    <PRO_E_REV_ANGLE_TO_LIMIT
type="int">0</PRO_E_REV_ANGLE_TO_LIMIT>
  </PRO_E_REV_ANGLE_TO>
</PRO_E_REV_ANGLE>
```

Array Element

An array element represents a collection of elements of the same type. Following is an example of an array of PRO_E_RNDCH_RADIUS elements in XML format.

```
<PRO_E_RNDCH_RADII type="array">
  <PRO_E_RNDCH_RADIUS type="compound">
    <PRO_E_STD_POINT_COLLECTION_APPL type="selection">
      <PRO_XML_REFERENCE type="reference">
        <PRO_XML_REFERENCE_OWNER
type="owner">K01_X_COLLECTION_PART.prt</PRO_XML_REFERENCE_OWNER>
        <PRO_XML_REFERENCE_ID type="id">1268</PRO_XML_REFERENCE_ID>
        <PRO_XML_REFERENCE_TYPE
type="prototype">PRO_CRV_END</PRO_XML_REFERENCE_TYPE>
      </PRO_XML_REFERENCE>
    </PRO_E_STD_POINT_COLLECTION_APPL>
    <PRO_E_RNDCH_LEG1 type="compound">
      <PRO_E_RNDCH_LEG_TYPE type="int">1</PRO_E_RNDCH_LEG_TYPE>
      <PRO_E_RNDCH_LEG_VALUE type="double">36.00</PRO_E_RNDCH_LEG_VALUE>
      <PRO_E_RNDCH_REFERENCE_EDGE type="selection" />
      <PRO_E_RNDCH_REFERENCE_POINT type="selection" />
    </PRO_E_RNDCH_LEG1>
  </PRO_E_RNDCH_RADIUS>
</PRO_E_RNDCH_RADII>
```

```
<PRO_E_RNDCH_LEG2 type="compound">
<PRO_E_RNDCH_LEG_TYPE type="int">0</PRO_E_RNDCH_LEG_TYPE>
<PRO_E_RNDCH_LEG_VALUE type="double">0.00</PRO_E_RNDCH_LEG_VALUE>
<PRO_E_RNDCH_REFERENCE_EDGE type="selection" />
<PRO_E_RNDCH_REFERENCE_POINT type="selection" />
</PRO_E_RNDCH_LEG2>
</PRO_E_RNDCH_RADIUS>
<PRO_E_RNDCH_RADIUS type="compound">
<PRO_E_STD_POINT_COLLECTION_APPL type="selection">
<PRO_XML_REFERENCE type="reference">
<PRO_XML_REFERENCE_OWNER
type="owner">K01_X_COLLECTION_PART.prt</PRO_XML_REFERENCE_OWNER>
<PRO_XML_REFERENCE_ID type="id">1268</PRO_XML_REFERENCE_ID>
<PRO_XML_REFERENCE_TYPE
type="prototype">PRO_CRV_START</PRO_XML_REFERENCE_TYPE>
</PRO_XML_REFERENCE>
</PRO_E_STD_POINT_COLLECTION_APPL>
<PRO_E_RNDCH_LEG1 type="compound">
<PRO_E_RNDCH_LEG_TYPE type="int">1</PRO_E_RNDCH_LEG_TYPE>
<PRO_E_RNDCH_LEG_VALUE type="double">55.39</PRO_E_RNDCH_LEG_VALUE>
<PRO_E_RNDCH_REFERENCE_EDGE type="selection" />
<PRO_E_RNDCH_REFERENCE_POINT type="selection" />
</PRO_E_RNDCH_LEG1>
<PRO_E_RNDCH_LEG2 type="compound">
<PRO_E_RNDCH_LEG_TYPE type="int">0</PRO_E_RNDCH_LEG_TYPE>
<PRO_E_RNDCH_LEG_VALUE type="double">0.00</PRO_E_RNDCH_LEG_VALUE>
<PRO_E_RNDCH_REFERENCE_EDGE type="selection" />
<PRO_E_RNDCH_REFERENCE_POINT type="selection" />
</PRO_E_RNDCH_LEG2>
</PRO_E_RNDCH_RADIUS>
</PRO_E_RNDCH_RADII>
```

Multivalued Element

The multi-valued element represents an array of same types of elements. The difference between the array and multi-valued elements lies in the Pro/TOOLKIT handling of these elements. In the XML format both the representations are similar.

```
<PRO_E_SHELL_SRF type="multivalue">
  <PRO_E_SHELL_SRF_MULT1 type="selection">
    <PRO_XML_REFERENCE type="reference">
      <PRO_XML_REFERENCE_OWNER
type="owner">X_SHELL_REDEF.prt</PRO_XML_REFERENCE_OWNER>
      <PRO_XML_REFERENCE_ID type="id">76</PRO_XML_REFERENCE_ID>
      <PRO_XML_REFERENCE_TYPE
type="prototype">PRO_SURFACE</PRO_XML_REFERENCE_TYPE>
    </PRO_XML_REFERENCE>
  </PRO_E_SHELL_SRF_MULT1>
  <PRO_E_SHELL_SRF_MULT1 type="selection">
    <PRO_XML_REFERENCE type="reference">
```

```

<PRO_XML_REFERENCE_OWNER
type="owner">X_SHELL_REDEF.prt</PRO_XML_REFERENCE_OWNER>
<PRO_XML_REFERENCE_ID type="id">72</PRO_XML_REFERENCE_ID>
<PRO_XML_REFERENCE_TYPE
type="prototype">PRO_SURFACE</PRO_XML_REFERENCE_TYPE>
</PRO_XML_REFERENCE>
</PRO_E_SHELL_SRF_MULTI>
<PRO_E_SHELL_SRF_MULTI type="selection">
<PRO_XML_REFERENCE type="reference">
<PRO_XML_REFERENCE_OWNER
type="owner">X_SHELL_REDEF.prt</PRO_XML_REFERENCE_OWNER>
<PRO_XML_REFERENCE_ID type="id">68</PRO_XML_REFERENCE_ID>
<PRO_XML_REFERENCE_TYPE
type="prototype">PRO_SURFACE</PRO_XML_REFERENCE_TYPE>
</PRO_XML_REFERENCE>
</PRO_E_SHELL_SRF_MULTI>
</PRO_E_SHELL_SRF>

```

Tips for Recycling XML Output of Element Trees

The easiest way to create an XML file for any feature element tree is to call **ProFeatureElemtreeExtract()** followed by **ProElemtreeWrite()**. The pt_userguide application includes this capability. While running this application, select a feature. Click the right mouse button and choose Export XML. The output XML produced can be used and reused with modifications by passing it to **ProElemtreeFromXMLCreate()** to create an element tree. The resulting element tree can be used for new feature creation or redefinition of an existing feature. The following tips would help easier reuse and understanding failures.

- Pro/E models can have only a single feature with a specific name - before using the XML for creating a new feature, modify the value of the element `PRO_E_STD_FEATURE_NAME` appropriately. Since, feature name is an optional element, the XML tag for this element can be deleted from the XML file to promote reuse.
- The function **ProElemtreeWrite()** writes all of the elements in the input Pro/TOOLKIT native element tree. Optional elements with empty data within XML tags from the XML file can be removed before reusing the XML file. It is also permitted to keep the optional elements as they are in the XML, but these elements are ignored by **ProElemtreeFromXMLCreate()**.

- **ProElemtreeFromXMLCreate()** converts the XML file to an element tree without validating that the data is appropriate for that feature type (for example, ensuring that the correct type is selected for references.) The error is detected when the element tree is used by **ProFeatureCreate()** or **ProFeatureRedefine()**.
- **ProElemtreeFromXMLCreate()** supports some enumerated symbols or the actual integer values. Use the output XML file as a guide to see which enumerated type names are accepted.
- The tag names used in the XML representation are derived from the list of **ProElemId** from *ProElemId.h*. These names must be used exactly as found in the *ProElemId.h* file.
- XML files created by one version of Pro/ENGINEER are not guaranteed to be usable by other Pro/ENGINEER versions, because the structure of the element tree also change between versions.

A Sample Element Tree in XML for a Shell Feature

```
<?xml version="1.0" encoding="UTF-8"?>
<!--Sample XML File for shell feature -->
<PRO_E_FEATURE_TREE AppName="Pro/ENGINEER" AppVersion="2003440"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="ProTKFeature.xsd" type="compound">
  <PRO_E_FEATURE_TYPE type="int">PRO_FEAT_SHELL</PRO_E_FEATURE_TYPE>
    <PRO_E_STD_FEATURE_NAME type="wstring">SHELL</PRO_E_STD_FEATURE_NAME>
      <PRO_E_SHELL_SRF type="multivalued">
        <PRO_E_SHELL_SRF_MULTI type="selection">
          <PRO_XML_REFERENCE type="reference">
            <PRO_XML_REFERENCE_OWNER
type="owner">X_SHELL.prt</PRO_XML_REFERENCE_OWNER>
            <PRO_XML_REFERENCE_ID type="id">76</PRO_XML_REFERENCE_ID>
            <PRO_XML_REFERENCE_TYPE
type="prototype">PRO_SURFACE</PRO_XML_REFERENCE_TYPE>
          </PRO_XML_REFERENCE>
        </PRO_E_SHELL_SRF_MULTI>
      </PRO_E_SHELL_SRF>
      <PRO_E_SHELL_THICK type="double">4.02</PRO_E_SHELL_THICK>
      <PRO_E_SHELL_FLIP type="int">1</PRO_E_SHELL_FLIP>
      <PRO_E_ST_SHELL_LOCL_LIST type="array">
        <PRO_E_ST_SHELL_LOCL_CMPD type="compound">
          <PRO_E_ST_SHELL_SPEC_SRF type="selection">
            <PRO_XML_REFERENCE type="reference">
              <PRO_XML_REFERENCE_OWNER
type="owner">X_SHELL.prt</PRO_XML_REFERENCE_OWNER>
              <PRO_XML_REFERENCE_ID type="id">53</PRO_XML_REFERENCE_ID>
              <PRO_XML_REFERENCE_TYPE
type="prototype">PRO_SURFACE</PRO_XML_REFERENCE_TYPE>
```

```

</PRO_XML_REFERENCE>
</PRO_E_ST_SHELL_SPEC_SRF>
<PRO_E_ST_SHELL_SPEC_THCK type="double">10.00</PRO_E_ST_SHELL_SPEC_THCK>
</PRO_E_ST_SHELL_LOCL_CMPD>
<PRO_E_ST_SHELL_LOCL_CMPD type="compound">
<PRO_E_ST_SHELL_SPEC_SRF type="selection">
<PRO_XML_REFERENCE type="reference">
<PRO_XML_REFERENCE_OWNER
type="owner">X_SHELL.prt</PRO_XML_REFERENCE_OWNER>
<PRO_XML_REFERENCE_ID type="id">66</PRO_XML_REFERENCE_ID>
<PRO_XML_REFERENCE_TYPE
type="prototype">PRO_SURFACE</PRO_XML_REFERENCE_TYPE>
</PRO_XML_REFERENCE>
</PRO_E_ST_SHELL_SPEC_SRF>
<PRO_E_ST_SHELL_SPEC_THCK type="double">9.00</PRO_E_ST_SHELL_SPEC_THCK>
</PRO_E_ST_SHELL_LOCL_CMPD>
<PRO_E_ST_SHELL_LOCL_CMPD type="compound">
<PRO_E_ST_SHELL_SPEC_SRF type="selection">
<PRO_XML_REFERENCE type="reference">
<PRO_XML_REFERENCE_OWNER
type="owner">X_SHELL.prt</PRO_XML_REFERENCE_OWNER>
<PRO_XML_REFERENCE_ID type="id">74</PRO_XML_REFERENCE_ID>
<PRO_XML_REFERENCE_TYPE
type="prototype">PRO_SURFACE</PRO_XML_REFERENCE_TYPE>
</PRO_XML_REFERENCE>
</PRO_E_ST_SHELL_SPEC_SRF>
<PRO_E_ST_SHELL_SPEC_THCK type="double">15.00</PRO_E_ST_SHELL_SPEC_THCK>
</PRO_E_ST_SHELL_LOCL_CMPD>
</PRO_E_ST_SHELL_LOCL_LIST>
</PRO_E_FEATURE_TREE>

```

The following example shows how to use **ProElemtreeWrite()** to export a feature element tree to XML.

```

#include <ProToolkit.h>
#include <ProMenuBar.h>
#include <ProMdl.h>
#include <ProSelection.h>
#include <ProFeature.h>
#include <ProWstring.h>

static
wchar_t MSGFIL[] =
{'u','t','i','l','i','t','i','e','s','.','t','x','t','\0'};

/*=====*\
FUNCTION: UserFeatXMLWrite
PURPOSE: Write a feature's element tree to an XML file.
\*=====*/
int UserFeatXMLWrite (ProFeature* feat)
{
    ProElement elemtree;

```

```

    wchar_t wFilename [PRO_FILE_NAME_SIZE];

/*-----*\
Prompt for the filename
\*-----*/
    ProMessageDisplay ( MSGFIL, "USER Enter the XML file name:");

    if (ProMessageStringRead (PRO_FILE_NAME_SIZE, wFilename)
        != PRO_TK_NO_ERROR)
        return (0);

/*-----*\
Extract the element tree and convert it to XML
\*-----*/
    status = ProFeatureElemtreeExtract (feat,NULL,PRO_FEAT_EXTRACT_NO_OPTS,
                                        &elemtree);

    if ( status == PRO_TK_NO_ERROR )
    {
        status = ProElemtreeWrite (elemtree, PRO_ELEMTREE_XML, wFilename);

        ProElementFree (&elemtree);
    }

    return status;
}

```


Element Trees: References

This chapter describes functions that provide access to reference objects as an alternative for accessing the information as selections.

Topic	Page
Overview of Reference Objects	30 - 2
Reading References	30 - 2
Modifying References	30 - 5

Overview of Reference Objects

Reference objects are an alternative method for representing a geometric reference in Pro/ENGINEER. Geometric references are usually represented using the *ProSelection* structure. Since *ProSelection* is designed as the result of an interactively selected item, it lacks some capabilities to provide complete meaning as a geometric reference.

The opaque handle *ProReference* provides complete functionality for geometric referencing including functions to access multiple or missing references.

Reading References

Functions Introduced:

- **ProReferenceStatusGet()**
- **ProReferenceIsLocalcopy()**
- **ProReferenceTypeGet()**
- **ProReferenceIdGet()**
- **ProReferenceOwnerGet()**
- **ProReferenceOwnernameGet()**
- **ProReferenceOriginaltypeGet()**
- **ProReferenceOriginalidGet()**
- **ProReferenceOriginalownerGet()**
- **ProReferenceOriginalownernameGet()**
- **ProReferenceAsmcomppathGet()**
- **ProReferenceParamsGet()**
- **ProReferencePointGet()**
- **ProReferenceToSelection()**
- **ProSelectionToReference()**
- **ProReferencearrayToSelections()**
- **ProSelectionarrayFree()**
- **ProSelectionarrayToReferences()**

- **ProReferenceFree()**
- **ProReferencearrayFree()**

The function **ProReferenceStatusGet()** identifies the status of the reference. Typically references are of status `PRO_REF_ACTIVE`, indicating that the reference is available and its geometry is usable for construction of features. Other reference statuses include:

- `PRO_REF_MISSING` – The reference is to geometry that is inactive in the model. When reading the element tree of a feature that modifies the geometry it references (for example, a round of edge removes the edge) the references will have this status.
- `PRO_REF_NOT_FOUND` – Similar to `PRO_REF_MISSING`. The function indicates a reference that is critical to the feature that uses the function.
- `PRO_REF_FROZEN` – The reference is to geometry in a component frozen due to other missing references.
- `PRO_REF_FROZEN_PLACE`, `PRO_REF_SUPPRESSED`, `PRO_REF_EXCLUDED` – Not returned by **ProReferenceStatusGet()**.
- `PRO_REF_INVALID` – The reference is invalid in the context of the feature and the element in which it is used (for example, a non-linear edge used as direction reference for drafts or translations.)
- `PRO_REF_ALTERNATE` – The reference is using an alternate reference. The original reference information is available.

Note: Reference status is highly dependent on the model state at the time the reference was obtained. References obtained from a feature via **ProFeatureElemtreeExtract()** may become “Missing” or “Not found” due to changes in the geometry applied later in the feature list. To obtain the reference status for a given reference as it is seen by a feature, use **ProInsertModeActivate()** to revert the model to the state just after that feature is created.

The function **ProReferenceIsLocalcopy()** identifies if the reference is a local copy of the external reference. If the reference is a local copy, the original reference information is available as well.

The function **ProReferenceTypeGet()** gets the type of handle that is referenced.

The function **ProReferenceIdGet()** gets the item identity of the reference handle.

The function **ProReferenceOwnerGet()** gets the `ProMdl` handle of the owner model for the reference. The output of these three functions provides access to the basic **ProModelitem** referenced by the **ProReference** handle. If the reference status is not `PRO_REF_ACTIVE`, some or all of this information may not be accessible (for example, the reference owner model is not accessible if the reference is to a geometry item in an unretrieved component). The function **ProReferenceOwnernameGet()** gets the reference owner name of the referenced geometry item, which may be available even if the model itself has not been retrieved.

The functions **ProReferenceOriginaltypeGet()**, **ProReferenceOriginalidGet()**, **ProReferenceOriginalownerGet()**, **ProReferenceOriginalownernameGet()** get the original properties of a geometric reference handle. These could be different from the actively used type if the reference has been backed up, copied locally, or replaced with an alternate.

The function **ProReferenceAsmcomppathGet()** gets the component path of a reference handle.

The function **ProReferenceParamsGet()** gets the u-v parameters of a reference handle.

The function **ProReferencePointGet()** gets the selected point of a reference handle.

The function **ProReferenceToSelection()** gets and allocates a **ProSelection** containing a representation for this reference. The output of this function is the resulting **ProSelection** handle. This selection is independently allocated and should be freed using **ProSelectionFree()**.

The function **ProSelectionToReference()** gets and allocates a **ProReference** containing a representation for this selection. The output of this function is the resulting **ProReference** handle. This reference is independently allocated and should be freed using **ProReferenceFree()**.

The function **ProReferencearrayToSelections()** converts a reference *ProArray* to a selection *ProArray*. The input arguments for this function are

- references – The *ProArray* of reference handles.

- `skip_unusable` – `PRO_B_TRUE` to skip the processing of missing references that cannot be valid selections.
`PRO_B_FALSE` to process all references.

The output for this function is a selection that is a *ProArray* of selection handles. You can free this array using the function **ProSelectionarrayFree()**.

The function **ProSelectionarrayToReferences()** converts a selection *ProArray* to a reference *ProArray*. *ProArray* of selection handles is given as the input and the *ProArray* of reference handles is the output. Free this array using **ProReferencearrayFree()**.

The function **ProReferenceFree()** frees a reference handle.

The function **ProReferencearrayFree()** frees a reference *ProArray*. This function also free each *ProReference* handle using **ProReferenceFree()**.

Modifying References

Functions Introduced:

- **ProReferenceAlloc()**
- **ProReferenceSet()**
- **ProReferenceParamsSet()**
- **ProReferencePointSet()**

The function **ProReferenceAlloc()** allocates a reference handle.

The function **ProReferenceSet()** sets the referenced model item and optionally component path for the reference handle.

The function **ProReferenceParamsSet()** sets the u-v parameters of a reference handle.

The function **ProReferencePointSet()** sets the selected point of a reference handle.

Element Trees: Datum Features

This chapter describes how to use the include files *ProDtmPln.h*, *ProDtmPnt.h*, *ProDtmAxis.h*, and *ProDtmCsys.h* to create datum features programmatically. The chapter on ‘Element Trees: Principles of Feature Creation’ provides necessary background for creating datum features; we recommend you read that material first.

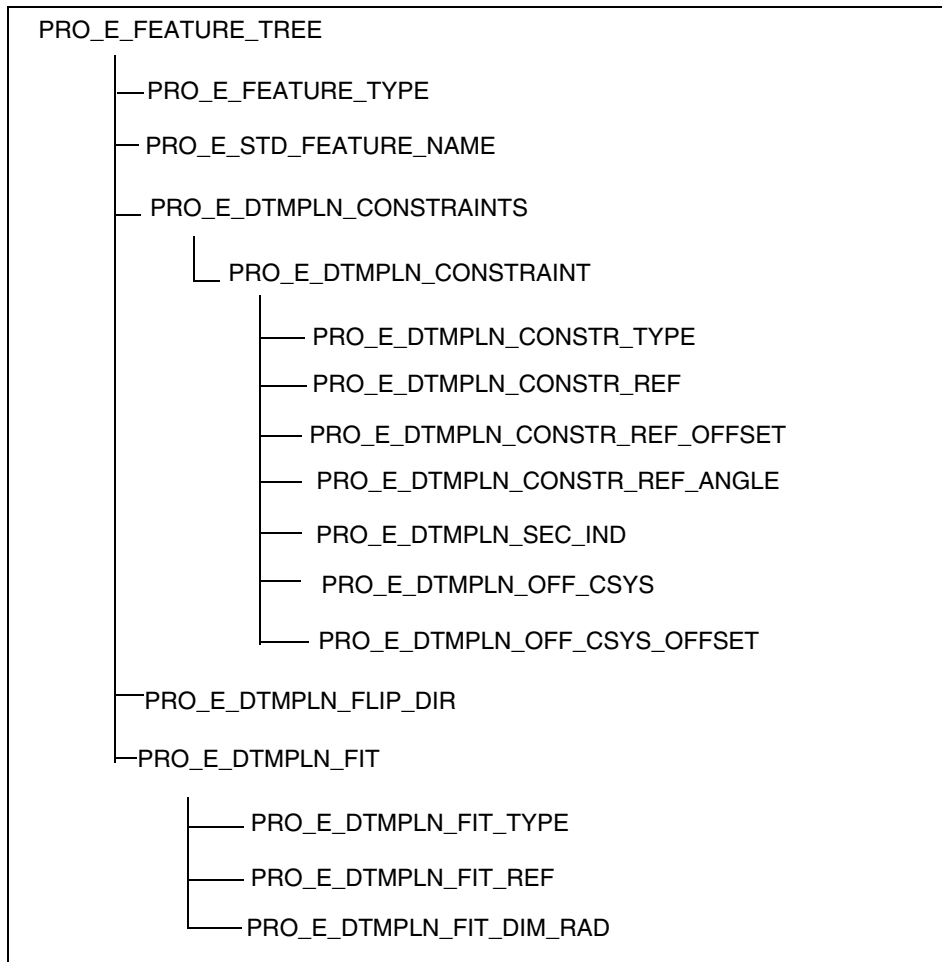
Topic	Page
Datum Plane Features	31 - 2
Datum Point Features	31 - 10
Datum Axis Features	31 - 56
Datum Coordinate System Features	31 - 68

Datum Plane Features

The element tree for a datum plane feature is documented in the header file *ProDtmPln.h*, and has a simple structure. Apart from the usual elements for the tree root and feature type, a datum plane contains the positioning constraints, an optional flip direction, and an optional fit type.

The constraints element `PRO_E_DTMPLN_CONSTRAINTS` is an array element that contains a `PRO_E_DTMPLN_CONSTRAINT` element for each constraint. Many elements forming the constraint element `PRO_E_DTMPLN_CONSTRAINT` are used only for certain constraint types, so any given datum plane may contain fewer elements than are shown in the tree. Similarly, all the elements forming the constraint element `PRO_E_DTMPLN_FIT` are not always essential.

The following figure shows the element tree for datum planes.



Element Trees:
Datum Features

Many elements forming the constraint element `PRO_E_DT MPLN_CONSTRAINT` are used only for the following constraint types:

- `PRO_E_DT MPLN_CONSTR_REF_OFFSET`—Used if the constraint type is “offset.”
- `PRO_E_DT MPLN_CONSTR_REF_ANGLE`—Used if the constraint type is “angle.”
- `PRO_E_DT MPLN_CONSTR_SEC_IND`—Used if the constraint type is “section.”
- `PRO_E_DT MPLN_OFF_CSYS`—Used if the constraint type is “offset” and the reference is “Csys.”

- PRO_E_DTMPLN_OFF_CSYS_OFFSET—Used if the constraint type is “offset” and the reference is “Csys.”

Similarly, elements of the optional element PRO_E_DTMPLN_FIT are used for the following fit types:

- PRO_E_DTMPLN_FIT_REF—Used if the fit type is not “default” or “fit.”
- PRO_E_DTMPLN_FIT_DIM_RAD—Used if the fit type is “fit radius.”

The following table describes the tree elements in detail.

Element ID	Element Name	Data Type	Valid Value
PRO_E_FEATURE_TYPE	Feature type	PRO_VALUE_TYPE_INT	PRO_FEAT_DATUM
PRO_E_STD_FEATURE_NAME	Feature Name	PRO_VALUE_TYPE_WSTRING	
PRO_E_DTMPLN_CONSTRAINTS	Constraints	Array	
PRO_E_DTMPLN_CONSTRAINT	Constraints	Compound	-
PRO_E_DTMPLN_CONSTR_TYPE	Type	PRO_VALUE_TYPE_INT	See ProDtmplnConstrType
PRO_E_DTMPLN_CONSTR_REF	References	PRO_VALUE_TYPE_SELECTION	See Constraint Reference Types
PRO_E_DTMPLN_CONSTR_REF_OFFSET	Offset	PRO_VALUE_TYPE_DOUBLE	Any
PRO_E_DTMPLN_CONSTR_REF_ANGLE	Angle	PRO_VALUE_TYPE_DOUBLE	(-360.0, 360.0)
PRO_E_DTMPLN_SEC_IND	Section index	PRO_VALUE_TYPE_INT	[0, sec num - 1]
PRO_E_DTMPLN_OFF_CSYS	Offset coordinate system	PRO_VALUE_TYPE_INT	See ProDtmplnOffCsysAxis

Element ID	Element Name	Data Type	Valid Value
PRO_E_DTMPLN_OFF_CSYS_OFFSET	Offset coordinate system value	PRO_VALUE_TYPE_DOUBLE	Any
PRO_E_DTMPLN_FLIP_DIR	Flip direction	PRO_VALUE_TYPE_INT	ProDtmplnFlipDir
PRO_E_DTMPLN_FIT	Fit	Compound	
PRO_E_DTMPLN_FIT_TYPE	Fit type	PRO_VALUE_TYPE_INT	ProDtmplnFitType
PRO_E_DTMPLN_FIT_REF	Reference	PRO_VALUE_TYPE_SELECTION	See Fit Reference Types
PRO_E_DTMPLN_FIT_DTM_RAD	Datum radius	PRO_VALUE_TYPE_DOUBLE	≥ 0.0

The following table does not describe the entire list of combinations of geometrical constraints that can be applied, or the rules for what geometry items they can refer to. These are partially documented in Note 1 of the elements table in *ProDtmPln.h*, which includes the following information:

Constraint Reference Types

Constraint Type	Valid Reference Types
PRO_DTMPLN_THRU	PRO_AXIS, PRO_EDGE, PRO_CURVE, Channel, PRO_POINT, PRO_EDGE_START, PRO_EDGE_END, PRO_CRV_START, PRO_CRV_END, PRO_SURFACE (Plane, Cylinder)
PRO_DTMPLN_NORM	PRO_AXIS, PRO_EDGE, PRO_CURVE, Channel PRO_SURFACE (plane)
PRO_DTMPLN_PRL	PRO_SURFACE (plane)
PRO_DTMPLN_OFFS	PRO_SURFACE (plane), PRO_CSYS
PRO_DTMPLN_ANG	PRO_SURFACE (plane)
PRO_DTMPLN_TANG	PRO_SURFACE (cylinder)
PRO_DTMPLN_SEC	PRO_FEATURE (blend)

Constraint Type	Valid Reference Types
PRO_DTMPLN_DEF_X	No reference needed
PRO_DTMPLN_DEF_Y	No reference needed
PRO_DTMPLN_DEF_Z	No reference needed
PRO_DTMPLN_THRU_CSYS_XY	PRO_CSYS
PRO_DTMPLN_THRU_CSYS_YZ	PRO_CSYS
PRO_DTMPLN_THRU_CSYS_ZX	PRO_CSYS

The following table describes the corresponding rules for the fit options in detail.

Fit Reference Types

Fit Type	Valid Reference Types
PRO_DTMPLN_FIT_DEFAULT	—
PRO_DTMPLN_FIT_PART	PRO_PART
PRO_DTMPLN_FIT_FEATURE	PRO_FEATURE
PRO_DTMPLN_FIT_SURFACE	PRO_SURFACE
PRO_DTMPLN_FIT_EDGE	PRO_EDGE
PRO_DTMPLN_FIT_AXIS	PRO_AXIS
PRO_DTMPLN_FIT_RADIUS	—

Example 1: Creating a Datum Plane

This example shows how to create a datum plane that is offset from the specified plane. The user selects the reference plane and supplies the offset.

```

/*=====*\
FUNCTION: UserDatumRefs()
PURPOSE:  Get the reference surface and offset for datum plane
          creation.
\*=====*/
int UserDatumRefs()
{
    static wchar_t    msgfil[80];
    ProError          status;
    ProSelection       *ref_surf;
    int                sel_num, level_flag;
    double             offset_value;

```

```

wchar_t          w_level[PRO_TYPE_SIZE];
char             level[PRO_TYPE_SIZE];
ProStringToWstring (msgfil, "msg_ugdatum.txt");

/*-----*\
  Prompt the user for the reference surface.
/*-----*\
status = ProMessageDisplay (msgfil, "USER Select a reference
                           surface");
status = ProSelect ("surface", 1, NULL, NULL, NULL, NULL,
                   &ref_surf, &sel_num);
/*-----*\
  Prompt the user for the offset distance
/*-----*\
status = ProMessageDisplay (msgfil, "USER Enter the offset value:");
if ((status = ProMessageDoubleRead(NULL, &offset_value))
    != PRO_TK_NO_ERROR)
    return (status);
/*-----*\
  Prompt user for level at which the datum is created (part or assembly).
/*-----*\
status = ProMessageDisplay (msgfil, "USER Create datum plane at
                           part or assembly level (prt or asm):");
status = ProMessageStringRead (PRO_TYPE_SIZE, w_level);

ProWstringToString (level, w_level);
if (!strcmp (level, "prt"))
    level_flag = PART;
else if (!strcmp (level, "asm"))
    level_flag = ASSEMBLY;
else
    return (status);
status = ProDatumPlaneCreate (*ref_surf, offset_value, level_flag);
return (status);
}
/*=====*\
FUNCTION: ProDatumPlaneCreate()
PURPOSE:  Create a datum plane as an assembly feature at an
          offset distance from a surface on a part.
/*=====*\
int ProDatumPlaneCreate (
    ProSelection offset_surface,
    double        offset_dist,
    int           dtm_level)
{
    ProError          status;
    ProElement        elem_tree, elem_ftype, elem_consts, elem_offset;
    ProElement        elem_const_type, elem_offset_ref, elem_offset_dist;
    ProValueData      value_data;
    ProValue          value;
    ProModelitem      surf_modelitem, model_modelitem;

```

```

ProSelection      mdl_sel;
ProFeature        feature;
ProErrorlist      errors;
ProAsmcomppath    p_cmp_path;
/*-----*\
Allocate the root element of the tree.
\*-----*/
status = ProElementAlloc (PRO_E_FEATURE_TREE, &elem_tree);
/*-----*\
Allocate the feature type element. Add it to the element tree.
\*-----*/
status = ProElementAlloc (PRO_E_FEATURE_TYPE, &elem_ftype);
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_FEAT_DATUM;
ProValueAlloc (&value);
ProValueDataSet (value, &value_data);
ProElementValueSet (elem_ftype, value);

ProElemtreeElementAdd (elem_tree, NULL, elem_ftype);
/*-----*\
Allocate the constraints element. Add it to the element tree.
\*-----*/
status = ProElementAlloc (PRO_E_DTMLN_CONSTRAINTS, &elem_consts);
ProElemtreeElementAdd (elem_tree, NULL, elem_consts);
/*-----*\
Allocate the constraint element. Add it under constraints element.
\*-----*/
status = ProElementAlloc (PRO_E_DTMLN_CONSTRAINT, &elem_offset);
status = ProElemtreeElementAdd (elem_consts, NULL, elem_offset);
/*-----*\
Allocate the constraint type element. Add it under the
constraint element.
\*-----*/
status = ProElementAlloc (PRO_E_DTMLN CONSTR_TYPE,
&elem_const_type);
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_DTMLN_OFFS;
ProValueAlloc (&value);
ProValueDataSet (value, &value_data);
ProElementValueSet (elem_const_type, value);

status = ProElemtreeElementAdd (elem_offset, NULL,
elem_const_type);
/*-----*\
Allocate the constraint reference element. Add it under the
constraint element.
\*-----*/
status = ProElementAlloc (PRO_E_DTMLN CONSTR_REF,
&elem_offset_ref);
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = offset_surface;

```

```

ProValueAlloc (&value);
ProValueDataSet (value, &value_data);
ProElementValueSet (elem_offset_ref, value);

status = ProElemtreeElementAdd (elem_offset, NULL, elem_offset_ref);
/*-----*\
Allocate the reference offset value element. Add it under the
constraint element.
\*-----*/
status = ProElementAlloc (PRO_E_DTMLN_CONSTR_REF_OFFSET,
                          &elem_offset_dist);
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = offset_dist;
ProValueAlloc (&value);
ProValueDataSet (value, &value_data);
ProElementValueSet (elem_offset_dist, value);

status = ProElemtreeElementAdd (elem_offset, NULL, elem_offset_dist);
/*-----*\
Get the assembly component path to the part that contains the
offset surface.
\*-----*/
status = ProSelectionAsmcomppathGet (offset_surface, &p_cmp_path);

switch (dtm_level)
{
    case ASSEMBLY:
        /*-----*\
        Get a ProModelitem handle to the root assembly
        \*-----*/
        status = ProMdlToModelitem (p_cmp_path.owner,
                                    &model_modelitem);
        /*-----*\
        Allocate a ProSection object for the root assembly.
        \*-----*/
        status = ProSelectionAlloc (NULL, &model_modelitem, &mdl_sel);
        break;

    case PART:
        /*-----*\
        Get a ProModelitem handle to the selected surface.
        \*-----*/
        status = ProSelectionModelitemGet (offset_surface,
                                          &surf_modelitem);
        /*-----*\
        Get a ProModelitem to the owner of the selected surface.
        \*-----*/
        status = ProMdlToModelitem (surf_modelitem.owner,
                                    &model_modelitem);
        /*-----*\
        Allocate a ProSection object for the part to which the

```

```

    selected surface belongs.
/*-----*/
    status = ProSelectionAlloc (&p_cmp_path, &model_modelitem,
                                &mdl_sel);

    break;

    default:
    return (0);
}
/*-----*/
    Create the datum plane.
/*-----*/
    status = ProFeatureCreate (mdl_sel, elem_tree, NULL, 0, &feature,
                                &errors);

    status = ProElementFree (&elem_tree);
    status = ProSelectionFree (&mdl_sel);

    return (status);
}

```

Datum Point Features

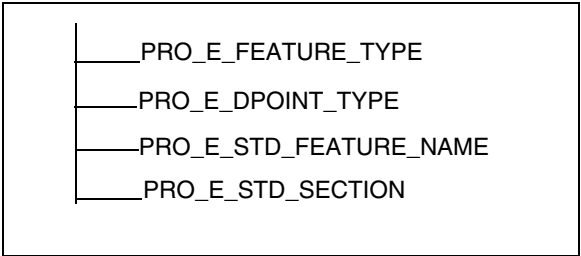
The element tree for a datum point feature is documented in the header file *ProDtmPnt.h*. Apart from the usual elements for the tree root and feature type, a datum point contains the datum point type. The types of datum points available are:

- Sketched Datum Point
- Field Datum Point
- Offset Csys Datum Point
- General Datum Point

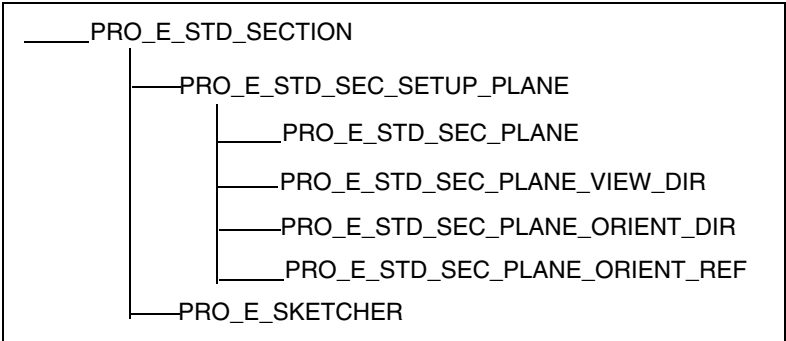
Sketched Datum Point

A sketched datum point is created by sketching the point in the sketcher mode after specifying the plane on which the user wants to create a point.

The following figure shows the element tree for a sketched datum point.



Define the following sub elements of PRO_E_STD_SECTION to complete the sketched datum point feature.



See the chapter ‘Element Trees: Sketched Features’ for techniques that must be used to create a sketched feature, like a sketched datum point.

Feature Elements

The following table describes the elements of the element tree for sketched datum points.

Element ID	Element Name	Data Type	Valid Value
PRO_E_FEATURE_TYPE	Feature Type	PRO_VALUE_TYPE_INT	PRO_FEAT_DATUM_POINT
PRO_E_DPOINT_TYPE	Datum Point Type	PRO_VALUE_TYPE_INT	PRO_DPOINT_TYPE_SKETCHED
PRO_E_STD_FEATURE_NAME	Feature Name	PRO_VALUE_TYPE_WSTRING	

Element ID	Element Name	Data Type	Valid Value
PRO_E_STD_SECTION	Section	Compound	See <i>ProStdSection.h</i>

Example 2: Creating a Sketched Datum Point

This example shows how to create a Sketched Datum Point. The user is prompted to select the sketching planes, orientation planes, and then the reference edges for the sketch. The user is also required to enter the X and Y offsets to be applied to the sketch from the projected edges.

```

/*=====*\
Creating a Sketched Datum Point
\*=====*/
/*----- Pro/Toolkit Includes -----*/
#include "ProToolkit.h"
#include "ProFeature.h"
#include "ProElemId.h"
#include "ProExtrude.h"
#include "ProModFeat.h"
#include "ProStdSection.h"
#include "ProElement.h"
#include "ProElempath.h"
#include "ProFeatType.h"
#include "ProFeatForm.h"
#include "ProSelection.h"
#include "ProSection.h"

#include "ProDtmPnt.h"

#define C_LOG(a) ProTKPrintf ("Status for %s is = %d\n", a, status ); \
ERROR_CHECK( a, "UgSktExtrusionCreate.c", status );

#define C_PRINT(a) ProTKPrintf ( "%s\n", a);

static ProFileName message_file;

/*----- Function Prototypes -----*/
ProError ProDemoSketchedPointCreate();
ProError UserSectionPointBuild ( ProSection section, ProSelection
*sketch_refs);

/*----- External Data -----*/
ProError ProDemoSectCreate();

/*----- Global Data -----*/

/*=====*\
FUNCTION : ProDemoSketchedPointCreate

```

PURPOSE : Demonstrates the creation of the extruded protrusion
base feature.

```
\*=====*/
ProError ProDemoSketchedPointCreate()
{
    ProErrorlist          errors;
    ProMdl                model;
    ProModelitem           model_item;
    ProSelection           model_sel;
    ProFeature             feature;
    ProFeatureCreateOptions opts[1];
    ProElempath            path;
    ProElempathItem        path_items[2];
    ProSection             section;
    ProAsmcomppath         comp_path;
    ProAsmcomppath         *p_comp_path = NULL;
    ProValue               value;

    ProBoolean            alloc;

    ProElement sketch_element;
    ProElement created_elemtree;

    ProElement pro_e_feature_tree;
    ProElement pro_e_feature_type;
    ProElement pro_e_dpoint_type;
    ProElement pro_e_std_feature_name;
    ProElement pro_e_std_section;
    ProElement pro_e_std_sec_method;
    ProElement pro_e_std_sec_setup_plane;
    ProElement pro_e_std_sec_plane;
    ProElement pro_e_std_sec_plane_view_dir;
    ProElement pro_e_std_sec_plane_orient_dir;
    ProElement pro_e_std_sec_plane_orient_ref;

    ProSelection *sketch_refs;

    ProName wide_string;
    ProError status;
    ProValueData value_data;
    ProSelection * p_select;
    int n_select;
    ProBoolean is_interactive = PRO_B_TRUE;

    ProStringToWstring ( message_file, "utilities.txt" );

    /*-----*\
        Populating root element PRO_E_FEATURE_TREE
    \*-----*/
    C_PRINT( " *** Processing Element PRO_E_FEATURE_TREE *** " );
    status = ProElementAlloc ( PRO_E_FEATURE_TREE, &pro_e_feature_tree );
```

Element Trees:
Datum Features

```

C_LOG( " ProElementAlloc " );

/*-----*\
    Populating element PRO_E_FEATURE_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TYPE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TYPE, &pro_e_feature_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_FEAT_DATUM_POINT; /* 931 */
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_feature_type, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_feature_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DPOINT_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_TYPE *** " );
status = ProElementAlloc ( PRO_E_DPOINT_TYPE, &pro_e_dpoint_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_DPOINT_TYPE_SKETCHED; /* 22 ProDPointType */
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_dpoint_type, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_dpoint_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating optional element PRO_E_STD_FEATURE_NAME
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_FEATURE_NAME *** " );
status = ProElementAlloc ( PRO_E_STD_FEATURE_NAME,
                          &pro_e_std_feature_name );
C_LOG( " ProElementAlloc " );
ProStringToWstring ( wide_string, "MY_PNT1" );
value_data.type = PRO_VALUE_TYPE_WSTRING;
value_data.v.w = wide_string;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );

```

```

C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_feature_name, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_std_feature_name );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating compound element PRO_E_STD_SECTION
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SECTION *** " );
status = ProElementAlloc ( PRO_E_STD_SECTION, &pro_e_std_section );
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_std_section );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Not required to populate    PRO_E_STD_SEC_METHOD
\*-----*/
/*
C_PRINT( " *** Processing Element PRO_E_STD_SEC_METHOD *** " );
status = ProElementAlloc ( PRO_E_STD_SEC_METHOD,
                          &pro_e_std_sec_method );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_SEC_SKETCH; ** 25 ProSecMethod **
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_sec_method, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_std_section, NULL,
                                pro_e_std_sec_method );
C_LOG( " ProElemtreeElementAdd" );
*/

sketch_refs = ( ProSelection *) calloc ( 2, sizeof ( ProSelection ));

/*-----*\
    Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SEC_SETUP_PLANE *** " );
status = ProElementAlloc ( PRO_E_STD_SEC_SETUP_PLANE,
                          &pro_e_std_sec_setup_plane );
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_std_section, NULL,
                                pro_e_std_sec_setup_plane );
C_LOG( " ProElemtreeElementAdd" );

```

```

/*-----*\
    Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    -> PRO_E_STD_SEC_PLANE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE *** " );
status = ProMessageDisplay ( message_file, "Select Surface for sketch
                                placement");
C_LOG( " ProMessageDisplay" );
ProTKPrintf ( "Please select datum,surface,sldface,qltface_ID_5 type of
                                Modelitem\n");
status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL, NULL,
                                NULL, NULL, &p_select, &n_select );
C_LOG( " ProSelect" );
if ( n_select <= 0 ) return -1;
else
{
    status = ProSelectionCopy( p_select[0], &sketch_refs[0]);
    C_LOG ( " ProSelectionCopy " );
}
status = ProElementAlloc ( PRO_E_STD_SEC_PLANE, &pro_e_std_sec_plane );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_sec_plane, value );
C_LOG( " ProElementValueSet" );
status = ProElementtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
                                pro_e_std_sec_plane );
C_LOG( " ProElementtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    -> PRO_E_STD_SEC_PLANE_VIEW_DIR
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_VIEW_DIR *** " );
status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_VIEW_DIR,
                                &pro_e_std_sec_plane_view_dir );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_SEC_VIEW_DIR_SIDE_ONE; /* 1 */
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_sec_plane_view_dir, value );

```

```

C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
                                pro_e_std_sec_plane_view_dir );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    -> PRO_E_STD_SEC_PLANE_ORIENT_DIR
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_ORIENT_DIR *** "
);
status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_ORIENT_DIR,
                           &pro_e_std_sec_plane_orient_dir );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_SEC_ORIENT_DIR_UP; /* 1 */
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_sec_plane_orient_dir, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
                                pro_e_std_sec_plane_orient_dir );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    -> PRO_E_STD_SEC_PLANE_ORIENT_REF
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_ORIENT_REF *** "
);
status = ProMessageDisplay ( message_file, "Select Surface for sketch
                                orientation");
C_LOG( " ProMessageDisplay" );
ProTKPrintf ( "Please select datum,surface,sldface,qltface_ID_5 type of
                Modelitem\n");
status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL,
                    NULL, NULL, NULL, &p_select, &n_select );
C_LOG( " ProSelect" );
if ( n_select <= 0 ) return -1;
else
{
    status = ProSelectionCopy( p_select[0], &sketch_refs[1]);
    C_LOG ( " ProSelectionCopy ");
}

status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_ORIENT_REF,
                           &pro_e_std_sec_plane_orient_ref );

```

```

C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_sec_plane_orient_ref, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
                                pro_e_std_sec_plane_orient_ref );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Creating incomplete feature in the current model.
\*-----*/
status = ProMdlCurrentGet (&model);
if ( status != PRO_TK_NO_ERROR ) return ( status );
status = ProMdlToModelitem( model, &model_item );
status = ProSelectionAlloc (p_comp_path, &model_item,
    &model_sel);

opts[0] = PRO_FEAT_CR_INCOMPLETE_FEAT;
status = ProFeatureCreate (model_sel, pro_e_feature_tree, opts, 1,
    &feature, &errors);
C_LOG ( " ProFeatureCreate");

/* Using the element tree from created feature */
status = ProFeatureElemtreeExtract (&feature, NULL,
    PRO_FEAT_EXTRACT_NO_OPTS,
    &created_elemtree);
C_LOG ( "ProFeatureElemtreeExtract()");

/*-----*\
    Getting the initialized section element from the database.
\*-----*/
/* path to PRO_E_SKETCHER element */
path_items[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[0].path_item.elem_id = PRO_E_STD_SECTION;
path_items[1].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[1].path_item.elem_id = PRO_E_SKETCHER;
status = ProElempathAlloc (&path);
status = ProElempathDataSet (path, path_items, 2);

status = ProElemtreeElementGet ( created_elemtree, path,
    &sketch_element);
C_LOG ( " ProElemtreeElementGet PRO_E_SKETCHER " );

status = ProElementValueGet ( sketch_element, &value);
C_LOG ( " ProElementValueGet PRO_E_SKETCHER " );

```



```

status = ProValueDataGet (value, &value_data);
C_LOG (" ProValueDataGet PRO_E_SKETCHER ");
section = (ProSection)value_data.v.p;

/*-----*\
    Create the 3-D Point section and get the section handle
\*-----*/

status = UserSectionPointBuild ( ( ProSection )(value_data.v.p),
                                sketch_refs );
C_LOG (" UserSectionPointBuild");

/*-----*\
    Redefining the feature to make it complete.
\*-----*/
opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMS;
status = ProSelectionAsmcomppathGet (model_sel, &comp_path);
C_LOG (" ProSelectionAsmcomppathGet");

status = ProFeatureRedefine (&comp_path, &feature, created_elemtree,
                             opts, 1, &errors);
C_LOG (" ProFeatureRedefine");

/*-----*\
    Free up the allocated memory.
\*-----*/
status = ProFeatureElemtreeFree (&feature, created_elemtree );
C_LOG (" ProFeatureElemtreeFree");

status = ProElementFree (&pro_e_feature_tree );
C_LOG (" ProElementFree");

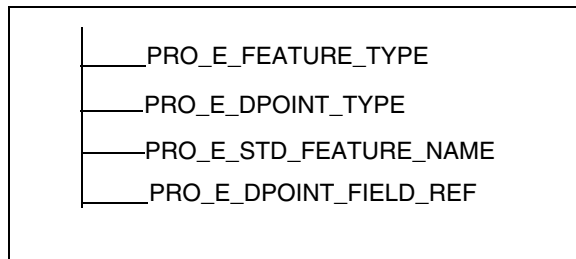
return (status);
}

```

Field Datum Point

A field datum point is created by selecting any point on a surface, edge, curve, or quilt. The point is located depending on the UV parameters. The location of the field point depends on the UV values of the point on the surface, edge, curve, or quilt.

The following figure shows the element tree for the field datum point.



Feature Elements

The following table describes the elements of the element tree for the field datum points.

Element ID	Element Name	Data Type	Valid Value
PRO_E_FEATUR E_TYPE	Feature Type	PRO_VALUE_TYP E_INT	PRO_FEAT_DAT UM_POINT
PRO_E_DPOINT_ TYPE	Datum Point Type	PRO_VALUE_TYP E_INT	PRO_DPOINT_T YPE_FIELD
PRO_E_STD_FE ATURE_NAME	Feature Name	PRO_VALUE_TYP E_WSTRING	
PRO_E_DPOINT_ FIELD_REF	Placement reference	PRO_VALUE_TYP E_SELECTION	Surface, Edge, Curve, or Quilt. Note: UV is used to specify exact location.

Example 3: Creating a Field Datum Point

This example shows how to create a Field Datum Point on an edge or surface. The user is prompted to select a point on a curve or a surface.

```

/*=====*\
Creating a Field Datum Point Feature
\*=====*/
/*----- Pro/Toolkit Includes -----*/
#include "ProToolkit.h"
#include "ProFeature.h"
#include "ProElemId.h"
#include "ProExtrude.h"
#include "ProModFeat.h"
#include "ProStdSection.h"
#include "ProElement.h"
#include "ProElempath.h"

```

```
#include "ProFeatType.h"
#include "ProFeatForm.h"
#include "ProSelection.h"
#include "ProSection.h"

#include "ProDtmPnt.h"

#define C_LOG(a) ProTKPrintf ("Status for %s is = %d\n", a, status ); \
ERROR_CHECK( a, "UgSktExtrusionCreate.c", status );

#define C_PRINT(a) ProTKPrintf ( "%s\n", a);

static ProFileName message_file;

/*----- Function Prototypes -----*/
ProError ProDemoFieldPointCreate();

/*----- External Data -----*/

/*----- Global Data -----*/

/*=====*\
FUNCTION : ProDemoFieldPointCreate
PURPOSE  : Demonstrates the creation of the Field Datum Point
\*=====*/
ProError ProDemoFieldPointCreate()
{
    ProErrorlist          errors;
    ProMdl                model;
    ProModelitem          model_item;
    ProSelection          model_sel;
    ProFeature            feature;
    ProFeatureCreateOptions opts[1];
    ProAsmcompopath       *p_comp_path = NULL;
    ProValue              value;
    char                  name[PRO_NAME_SIZE];
    ProError              status;

    ProElement pro_e_feature_tree;
    ProElement pro_e_feature_type;
    ProElement pro_e_dpoint_type;
    ProElement pro_e_std_feature_name;
    ProElement pro_e_dpoint_field_ref;

    ProName wide_string;
    ProValueData value_data;
    ProSelection * p_select;
    int n_select;
    ProBoolean is_interactive = PRO_B_TRUE;

    ProStringToWstring ( message_file, "utilities.txt" );
```

```

/*-----*\
    Populating root element PRO_E_FEATURE_TREE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TREE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TREE, &pro_e_feature_tree );
C_LOG( " ProElementAlloc " );

/*-----*\
    Populating element PRO_E_FEATURE_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TYPE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TYPE, &pro_e_feature_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_FEAT_DATUM_POINT; /* 931 */
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_feature_type, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_feature_type );
C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Populating element PRO_E_DPOINT_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_TYPE *** " );
status = ProElementAlloc ( PRO_E_DPOINT_TYPE, &pro_e_dpoint_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_DPOINT_TYPE_FIELD; /* 21 */
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_dpoint_type, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_dpoint_type );
C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Populating element PRO_E_DPOINT_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_FEATURE_NAME *** " );
status = ProElementAlloc ( PRO_E_STD_FEATURE_NAME,
                          &pro_e_std_feature_name );
C_LOG( " ProElementAlloc " );

```

```

ProStringToWstring ( wide_string, "MY_FPNT0" );
value_data.type = PRO_VALUE_TYPE_WSTRING;
value_data.v.w = wide_string;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_feature_name, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_std_feature_name );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DPOINT_FIELD_REF
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_FIELD_REF *** " );
status = ProMessageDisplay ( message_file, "Select a reference
                                Surface");
C_LOG( " ProMessageDisplay" );
ProTKPrintf ( "Please select datum,surface,sldface,qltface_ID_5 type of
                                Modelitem\n");
status = ProSelect ( "curve,surface", 1, NULL, NULL, NULL, NULL,
                    &p_select, &n_select );
C_LOG( " ProSelect" );
if ( n_select <= 0 ) return -1;
status = ProElementAlloc ( PRO_E_DPOINT_FIELD_REF,
                           &pro_e_dpoint_field_ref );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_dpoint_field_ref, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_dpoint_field_ref );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Create the feature in the current model.
\*-----*/
status = ProMdlCurrentGet (&model);
if ( status != PRO_TK_NO_ERROR ) return ( status );
status = ProMdlToModelitem( model, &model_item );
status = ProSelectionAlloc ( p_comp_path, &model_item, &model_sel);

opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMS;
status = ProFeatureCreate (model_sel, pro_e_feature_tree, opts, 1,

```

```

                                &feature, &errors);
C_LOG (" ProFeatureCreate");

status = ProElementFree (&pro_e_feature_tree );
C_LOG (" ProElementFree");

return (status);
}

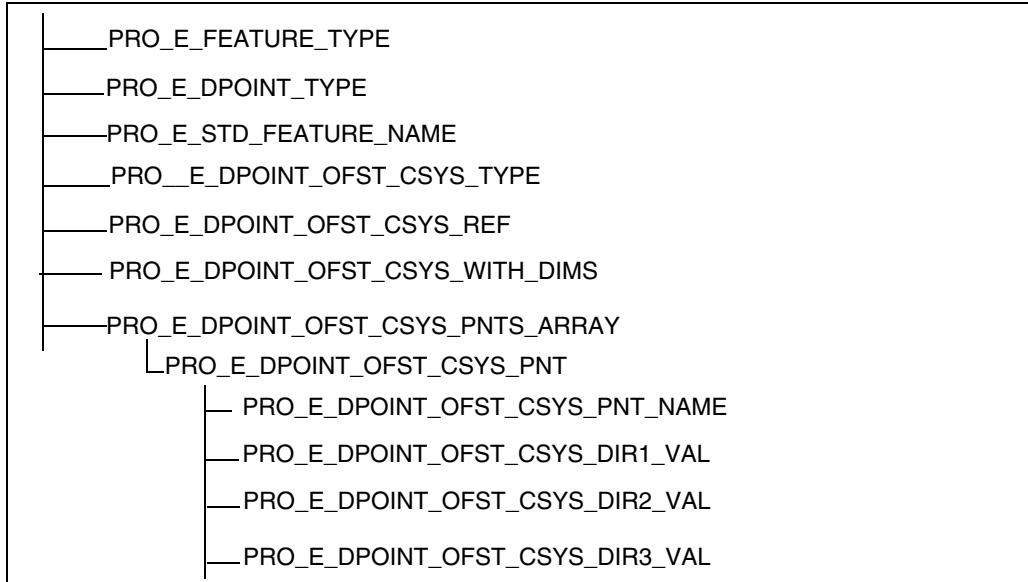
```

Offset Csys Datum Point

An Offset Csys Datum point is created using the coordinate system and values along the coordinate axes. Three types of coordinate systems can be used:

- Cartesian—Requires values along X, Y, Z axis.
- Cylindrical—Requires values along R, *theta*, Z axis.
- Spherical—Requires values along r, *phi*, *theta* axis.

The following figure shows the element tree for Offset Csys Datum Point.



Element Trees:
Datum Features

Feature Elements

The following table describes the elements in the element tree for datum points.

Element ID	Element Name	Data Type	Valid Value
PRO_E_FEATURE_TYPE	Feature Type	PRO_VALUE_TYPE_INT	PRO_FEAT_DATUM_POINT
PRO_E_DPOINT_TYPE	Datum Point Type	PRO_VALUE_TYPE_INT	PRO_DPOINT_TYPE_OFFSET_CSYS
PRO_E_STD_FEATURE_NAME	Feature Name	PRO_VALUE_TYPE_WSTRING	
PRO_E_DPOINT_OFST_CSYS_TYPE	Reference Csys Type	PRO_VALUE_TYPE_INT	See ProDtmpntOffCsysType
PRO_E_DPOINT_OFST_CSYS_REF	Reference Csys	PRO_VALUE_TYPE_SELECTION	Csys

Element ID	Element Name	Data Type	Valid Value
PRO_E_DPOINT_OF ST_CSYS_WITH_DIMS	Parametric or Explicit with or without dimensions	PRO_VALUE_TYPE _INT	PRO_B_TRUE or PRO_B_FALSE
PRO_E_DPOINT_OF ST_CSYS_PNTS_ARRAY Points List	Array		
PRO_E_DPOINT_OF ST_CSYS_PNT	One Point	Compound	
PRO_E_DPOINT_OF ST_CSYS_PNT_NAME	Point Name	PRO_VALUE_TYPE _WSTRING	
PRO_E_DPOINT_OF ST_CSYS_DIR1_VAL	X, R, or <i>pi</i>	PRO_VALUE_TYPE _DOUBLE	Depends on PRO_E_DPOINT_OFST_CSYS _TYPE
PRO_E_DPOINT_OF ST_CSYS_DIR2_VAL	Y, <i>theta</i> , or <i>phi</i>	PRO_VALUE_TYPE _DOUBLE	Depends on PRO_E_DPOINT_OFST_CSYS _TYPE
PRO_E_DPOINT_OF ST_CSYS_DIR3_VAL	Z, <i>Z</i> , or <i>theta</i>	PRO_VALUE_TYPE _DOUBLE	Depends on PRO_E_DPOINT_OFST_CSYS _TYPE

Example 4: Creating an Offset Csys Datum Point

This example shows how to create an Offset Datum Point at an offset from a specified coordinate system. The user is prompted to select the coordinate system.

```

/*=====*\
Creating an Offset Datum Point Feature
\*=====*/
/*----- Pro/Toolkit Includes -----*/
#include "ProToolkit.h"
#include "ProFeature.h"
#include "ProElemId.h"
#include "ProExtrude.h"
#include "ProModFeat.h"
#include "ProStdSection.h"
#include "ProElement.h"
#include "ProElmpath.h"

```



```

#include "ProFeatType.h"
#include "ProFeatForm.h"
#include "ProSelection.h"
#include "ProSection.h"

#include "ProDtmPnt.h"

#define C_LOG(a) ProTKPrintf ("Status for %s is = %d\n", a, status ); \
ERROR_CHECK( a, "UgSktExtrusionCreate.c", status );

#define C_PRINT(a) ProTKPrintf ( "%s\n", a);

/*----- Function Prototypes -----*/
ProError ProDemoOffsetPointCreate();

/*----- External Data -----*/

/*----- Global Data -----*/

/*=====*\
FUNCTION : ProDemoOffsetPointCreate
PURPOSE  : Demonstrates the creation of the extruded protrusion
           base feature.
\*=====*/
ProError ProDemoOffsetPointCreate()
{
    ProErrorlist          errors;
    ProMdl                model;
    ProModelitem          model_item;
    ProSelection          model_sel;
    ProFeature            feature;
    ProFeatureCreateOptions opts[1];
    ProAsmcomppath        *p_comp_path = NULL;
    ProValue              value;
    char                  name[PRO_NAME_SIZE];
    ProError              status;

    ProElement pro_e_feature_tree;
    ProElement pro_e_feature_type;
    ProElement pro_e_dpoint_type;
    ProElement pro_e_dpoint_ofst_csys_type;
    ProElement pro_e_dpoint_ofst_csys_ref;
    ProElement pro_e_dpoint_ofst_csys_with_dims;
    ProElement pro_e_dpoint_ofst_csys_pnts_array;
    ProElement pro_e_dpoint_ofst_csys_pnt;
    ProElement pro_e_dpoint_ofst_csys_pnt_name;
    ProElement pro_e_dpoint_ofst_csys_dir1_val;
    ProElement pro_e_dpoint_ofst_csys_dir2_val;
    ProElement pro_e_dpoint_ofst_csys_dir3_val;

    ProName wide_string;

```

```

ProValueData value_data;
ProSelection * p_select;
int n_select;
ProBoolean is_interactive = PRO_B_TRUE;

ProStringToWstring ( message_file, "utilities.txt" );

/*-----*\
    Populating root element PRO_E_FEATURE_TREE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TREE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TREE, &pro_e_feature_tree );
C_LOG( " ProElementAlloc " );

/*-----*\
    Populating element PRO_E_FEATURE_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TYPE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TYPE, &pro_e_feature_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_FEAT_DATUM_POINT; /* 931 */
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_feature_type, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_feature_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DPOINT_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_TYPE *** " );
status = ProElementAlloc ( PRO_E_DPOINT_TYPE, &pro_e_dpoint_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_DPOINT_TYPE_OFFSET_CSYS; /* 6 */
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_dpoint_type, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_dpoint_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\

```

```

    Populating element PRO_E_DPOINT_OFST_CSYS_TYPE
/*-----*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_OFST_CSYS_TYPE *** " );
status = ProElementAlloc ( PRO_E_DPOINT_OFST_CSYS_TYPE,
                          &pro_e_dpoint_ofst_csys_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_DTMPNT_OFFCSYS_CARTESIAN; /* 0 */
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_dpoint_ofst_csys_type, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_dpoint_ofst_csys_type );
C_LOG( " ProElemtreeElementAdd " );

/*-----*/
    Populating element PRO_E_DPOINT_OFST_CSYS_REF
/*-----*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_OFST_CSYS_REF *** " );
status = ProMessageDisplay ( message_file, "Select a reference CSYS");
C_LOG( " ProMessageDisplay " );
ProTKPrintf ( "Please select csys_ID_25 type of Modelitem\n");
status = ProSelect ( "csys", 1, NULL, NULL, NULL, NULL, &p_select,
                   &n_select );
C_LOG( " ProSelect " );
if ( n_select <= 0 ) return -1;
status = ProElementAlloc ( PRO_E_DPOINT_OFST_CSYS_REF,
                          &pro_e_dpoint_ofst_csys_ref );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_dpoint_ofst_csys_ref, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_dpoint_ofst_csys_ref );
C_LOG( " ProElemtreeElementAdd " );

/*-----*/
    Populating element PRO_E_DPOINT_OFST_CSYS_WITH_DIMS
/*-----*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_OFST_CSYS_WITH_DIMS *** "
);
status = ProElementAlloc ( PRO_E_DPOINT_OFST_CSYS_WITH_DIMS,
                          &pro_e_dpoint_ofst_csys_with_dims );

```

```

C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_B_TRUE;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_dpoint_ofst_csys_with_dims, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_dpoint_ofst_csys_with_dims );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating array element PRO_E_DPOINT_OFST_CSYS_PNTS_ARRAY
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_OFST_CSYS_PNTS_ARRAY ***
" );
status = ProElementAlloc ( PRO_E_DPOINT_OFST_CSYS_PNTS_ARRAY,
                          &pro_e_dpoint_ofst_csys_pnts_array );
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_dpoint_ofst_csys_pnts_array );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DPOINT_OFST_CSYS_PNTS_ARRAY
-> PRO_E_DPOINT_OFST_CSYS_PNT
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_OFST_CSYS_PNT *** " );
status = ProElementAlloc ( PRO_E_DPOINT_OFST_CSYS_PNT,
                          &pro_e_dpoint_ofst_csys_pnt );
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_dpoint_ofst_csys_pnts_array,
                                NULL, pro_e_dpoint_ofst_csys_pnt );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DPOINT_OFST_CSYS_PNTS_ARRAY
-> PRO_E_DPOINT_OFST_CSYS_PNT
-> PRO_E_DPOINT_OFST_CSYS_PNT_NAME
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_OFST_CSYS_PNT_NAME *** "
);
status = ProElementAlloc ( PRO_E_DPOINT_OFST_CSYS_PNT_NAME,
                          &pro_e_dpoint_ofst_csys_pnt_name );
C_LOG( " ProElementAlloc " );
ProStringToWstring ( wide_string, "PNT2" );
value_data.type = PRO_VALUE_TYPE_WSTRING;
value_data.v.w = wide_string;
status = ProValueAlloc ( &value );

```

```

C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_dpoint_ofst_csys_pnt_name, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_dpoint_ofst_csys_pnt, NULL,
                                pro_e_dpoint_ofst_csys_pnt_name );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DPOINT_OFST_CSYS_PNTS_ARRAY
-> PRO_E_DPOINT_OFST_CSYS_PNT
-> PRO_E_DPOINT_OFST_CSYS_DIR1_VAL
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_OFST_CSYS_DIR1_VAL *** "
);
status = ProElementAlloc ( PRO_E_DPOINT_OFST_CSYS_DIR1_VAL,
                           &pro_e_dpoint_ofst_csys_dir1_val );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 100.000000;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_dpoint_ofst_csys_dir1_val, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_dpoint_ofst_csys_pnt, NULL,
                                pro_e_dpoint_ofst_csys_dir1_val );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DPOINT_OFST_CSYS_PNTS_ARRAY
-> PRO_E_DPOINT_OFST_CSYS_PNT
-> PRO_E_DPOINT_OFST_CSYS_DIR2_VAL
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_OFST_CSYS_DIR2_VAL *** "
);
status = ProElementAlloc ( PRO_E_DPOINT_OFST_CSYS_DIR2_VAL,
                           &pro_e_dpoint_ofst_csys_dir2_val );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 200.000000;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_dpoint_ofst_csys_dir2_val, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_dpoint_ofst_csys_pnt, NULL,
                                pro_e_dpoint_ofst_csys_dir2_val );

```

```

C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DPOINT_OFST_CSYS_PNTS_ARRAY
-> PRO_E_DPOINT_OFST_CSYS_PNT
-> PRO_E_DPOINT_OFST_CSYS_DIR3_VAL
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_OFST_CSYS_DIR3_VAL *** "
);
status = ProElementAlloc ( PRO_E_DPOINT_OFST_CSYS_DIR3_VAL,
                          &pro_e_dpoint_ofst_csys_dir3_val );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 300.000000;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_dpoint_ofst_csys_dir3_val, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_dpoint_ofst_csys_pnt, NULL,
                                pro_e_dpoint_ofst_csys_dir3_val );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Creating the feature in the current model.
\*-----*/
status = ProMdlCurrentGet (&model);
if ( status != PRO_TK_NO_ERROR ) return ( status );
status = ProMdlToModelitem( model, &model_item );
status = ProSelectionAlloc (p_comp_path, &model_item, &model_sel);

opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMS;
status = ProFeatureCreate (model_sel, pro_e_feature_tree, opts, 1,
&feature, &errors);
C_LOG ( " ProFeatureCreate");

status = ProElementFree (&pro_e_feature_tree );
C_LOG ( " ProElementFree");

return (status);
}

```

General Datum Point

A general datum point is created and constrained based on the selection context. The supported types are:

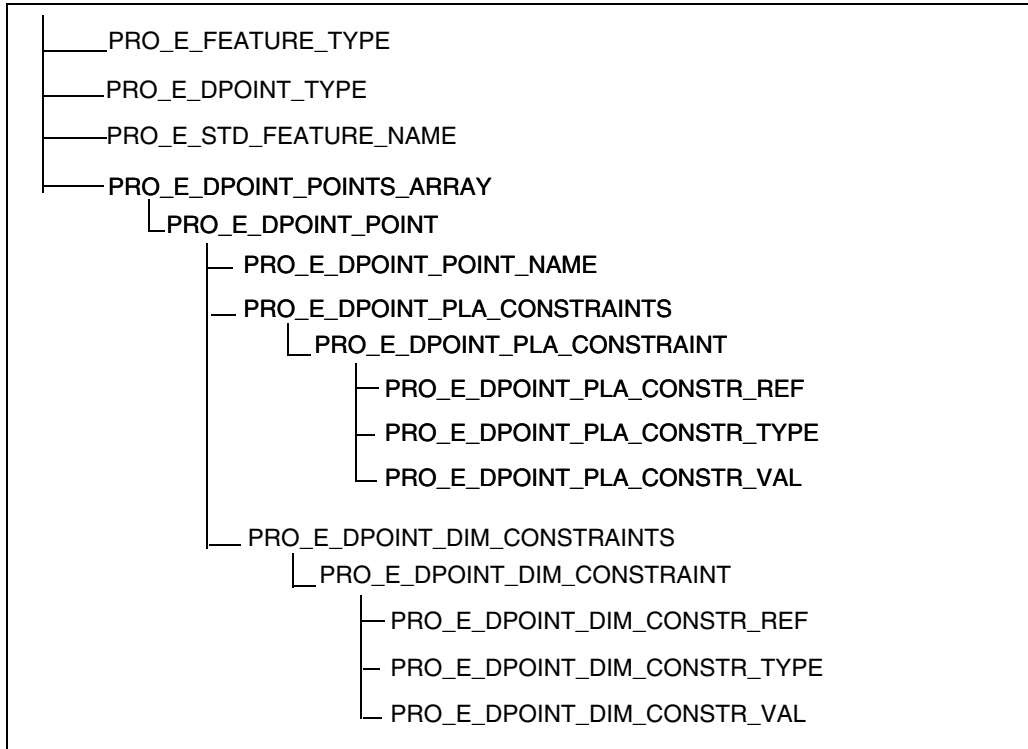
- Point on Vertex

- Offset point
- Point at intersection of three surfaces
- On or Offset surface
- Point at intersection of curve and surface
- Center of curve or surface
- Point at intersection of two curves
- Point on curve

When there are multiple intersections, the point location of general datum point depends on the following:

- Point at intersection of edge and edge—t value of point on second edge
- Point at intersection of edge and plane—t value of point on edge
- Point at intersection of curve and plane—t value of point on curve
- Point at intersection of two curves—t value of point on second curve
- Point at intersection of curve and surface—t value of point on curve
- Point at intersection of curve and axis—t value of point on curve

The following figure shows the element tree of a general datum point.



Feature Elements

The following table describes the elements of the element tree for datum points.

Element ID	Element Name	Data Type	Valid Value
PRO_E_FEATURE_TYPE	Feature Type	PRO_VALUE_TYPE_INT	PRO_FEAT_DATUM_POINT
PRO_E_DPOINT_TYPE	Datum Point Type	PRO_VALUE_TYPE_INT	PRO_DPOINT_TYPE_GENERAL
PRO_E_STD_FEATURE_NAME	Feature Name	PRO_VALUE_TYPE_WSTRING	
PRO_E_DPOINT_POINTS_ARRAY	Points List	Array	Not applicable

Element ID	Element Name	Data Type	Valid Value
PRO_E_DPOINT_POINT	One Point	Compound	Not applicable
PRO_E_DPOINT_POINT_NAME	Point Name	PRO_VALUE_TYPE_WSTRING	
PRO_E_DPOINT_PLA_CONSTRAINTS	Placement Constraints	Array	Not applicable
PRO_E_DPOINT_PLA_CONSTRAINT	One Placement Constraint	Compound	Not applicable
PRO_E_DPOINT_PLA_CONSTR_REF	Placement Reference	PRO_VALUE_TYPE_SELECTION	Depends on the context. See PRO_E_DPOINT_PLA_CONSTR_REF .
PRO_E_DPOINT_PLA_CONSTR_TYPE	Constraint Type	PRO_VALUE_TYPE_INT	See ProDtmpntConstrType .
PRO_E_DPOINT_PLA_CONSTR_VAL	Value	PRO_VALUE_TYPE_DOUBLE	
PRO_E_DPOINT_DIM_CONSTRAINTS	Dimension Constraints	Array	Not applicable
PRO_E_DPOINT_DIM_CONSTRAINT	One Dimension Constraint	Compound	Not applicable
PRO_E_DPOINT_DIM_CONSTR_REF	Dimension Reference	PRO_VALUE_TYPE_SELECTION	See Placement Constraint References.
PRO_E_DPOINT_DIM_CONSTR_TYPE	Constraint Type	PRO_VALUE_TYPE_INT	Depends on the context. See Constraint Type.
PRO_E_DPOINT_DIM_CONSTR_VAL	Value	PRO_VALUE_TYPE_DOUBLE	See ProDtmpntConstrType

Placement Constraint References

Valid values for the PRO_E_DPOINT_PLA_CONSTR_REF placement reference are as follows:

- Curve—SEL_3D_CURVE, SEL_3D_CABLE, SEL_IGES_WF
- Edge—SEL_3D_EDG
- Axis—SEL_3D_AXIS
- Vertex—SEL_3D_VERT or SEL_CURVE_END
- CSYS—SEL_3D_CSYS
- Surface—SEL_3D_SRF, SEL_3D_SRF_LIST
- Datum Pnt—SEL_3D_PNT

Placement Constraint Type

Valid values for PRO_E_DPOINT_PLA_CONSTR_TYPE are as follows:

- PRO_DTMPNT_CONSTR_TYPE_ON
- PRO_DTMPNT_CONSTR_TYPE_OFFSET
- PRO_DTMPNT_CONSTR_TYPE_CENTER
- PRO_DTMPNT_CONSTR_TYPE_PARALLEL
- PRO_DTMPNT_CONSTR_TYPE_NORMAL
- PRO_DTMPNT_CONSTR_TYPE_CARTESIAN
- PRO_DTMPNT_CONSTR_TYPE_CYLINDRICAL
- PRO_DTMPNT_CONSTR_TYPE_SPHERICAL

The last three are used when defining a point offset to a coordinate system.

Constraint References

Valid values for the PRO_E_DPOINT_DIM_CONSTR_REF dimension references are as following:

- Curve—SEL_3D_CURVE, SEL_3D_CABLE, SEL_CRV_PNT, SEL_IGES_WF
- Edge—SEL_3D_EDG, SEL_EDG_PNT
- Axis—SEL_3D_AXIS
- Coordinate system—SEL_3D_CSYS
- Vertex—SEL_3D_VERT or SEL_CURVE_END

- Surface—SEL_3D_SRF, SEL_SRF_PNT, SEL_3D_SRF_LIST
- Coordinate system axis—SEL_3D_CSYS_AXIS
- Datum Point—SEL_3D_PNT

Constraint Type

Valid values for PRO_E_DPOINT_DIM_CONSTR_TYPE are as follows:

- PRO_DTMPNT_CONSTR_TYPE_OFFSET
- PRO_DTMPNT_CONSTR_TYPE_LENGTH
- PRO_DTMPNT_CONSTR_TYPE_RATIO
- PRO_DTMPNT_CONSTR_TYPE_LENGTH_END
- PRO_DTMPNT_CONSTR_TYPE_RATIO
- PRO_DTMPNT_CONSTR_TYPE_RATIO_END
- PRO_DTMPNT_CONSTR_TYPE_ALONG_X
- PRO_DTMPNT_CONSTR_TYPE_ALONG_Y
- PRO_DTMPNT_CONSTR_TYPE_ALONG_Z

Example 5: Creating General Datum Point

This example shows how to create a General Datum Point formed at the intersection of three selected surfaces. The user is prompted to select the three surfaces.

```
/*=====*\
Creating a General Datum Point Feature
\*=====*/

/*----- Pro/Toolkit Includes -----*/
#include "ProToolkit.h"
#include "ProFeature.h"
#include "ProElemId.h"
#include "ProExtrude.h"
#include "ProModFeat.h"
#include "ProStdSection.h"
#include "ProElement.h"
#include "ProElmpath.h"
#include "ProFeatType.h"
#include "ProFeatForm.h"
#include "ProSelection.h"
#include "ProSection.h"

#include "ProDtmPnt.h"
```

```

#define C_LOG(a) ProTKPrintf ("Status for %s is = %d\n", a, status ); \
ERROR_CHECK( a, "UgSktExtrusionCreate.c", status );

#define C_PRINT(a) ProTKPrintf ( "%s\n", a);

static ProFileName message_file;

/*----- Function Prototypes -----*/
ProError ProDemoGeneralPointCreate();

/*----- External Data -----*/

/*----- Global Data -----*/

/*=====*\
FUNCTION : ProDemoGeneralPointCreate
PURPOSE  : Demonstrates the creation of the General Datum Point
\*=====*/
ProError ProDemoGeneralPointCreate()
{
    ProErrorlist          errors;
    ProMdl                model;
    ProModelitem          model_item;
    ProSelection          model_sel;
    ProFeature            feature;
    ProFeatureCreateOptions opts[1];
    ProAsmcomppath        *p_comp_path = NULL;
    ProValue              value;
    char                  name[PRO_NAME_SIZE];
    ProError              status;

    ProElement pro_e_feature_tree;
    ProElement pro_e_feature_type;
    ProElement pro_e_dpoint_type;
    ProElement pro_e_std_feature_name;
    ProElement pro_e_dpoint_points_array;
    ProElement pro_e_dpoint_point;
    ProElement pro_e_dpoint_point_name;
    ProElement pro_e_dpoint_pla_constraints;
    ProElement pro_e_dpoint_pla_constraint;
    ProElement pro_e_dpoint_pla_constr_ref;
    ProElement pro_e_dpoint_pla_constr_type;
    ProElement pro_e_dpoint_pla_constr_val;

    ProName wide_string;
    ProValueData value_data;
    ProSelection * p_select;
    int n_select;
    ProBoolean is_interactive = PRO_B_TRUE;

```

```

ProStringToWstring ( message_file, "utilities.txt" );

/*-----*\
    Populating root element PRO_E_FEATURE_TREE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TREE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TREE, &pro_e_feature_tree );
C_LOG( " ProElementAlloc " );

/*-----*\
    Populating element PRO_E_FEATURE_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TYPE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TYPE, &pro_e_feature_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_FEAT_DATUM_POINT; /* 931 */
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_feature_type, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_feature_type );
C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Populating element PRO_E_DPOINT_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_TYPE *** " );
status = ProElementAlloc ( PRO_E_DPOINT_TYPE, &pro_e_dpoint_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_DPOINT_TYPE_GENERAL; /* -1 ProDPointType */
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_dpoint_type, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_dpoint_type );
C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Populating element PRO_E_STD_FEATURE_NAME
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_FEATURE_NAME *** " );
status = ProElementAlloc ( PRO_E_STD_FEATURE_NAME,
                           &pro_e_std_feature_name );

```

```

C_LOG( " ProElementAlloc " );
ProStringToWstring ( wide_string, "MY_PNT0" );
value_data.type = PRO_VALUE_TYPE_WSTRING;
value_data.v.w = wide_string;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_feature_name, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_std_feature_name );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DPOINT_POINTS_ARRAY
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_POINTS_ARRAY *** " );
status = ProElementAlloc ( PRO_E_DPOINT_POINTS_ARRAY,
                          &pro_e_dpoint_points_array );
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_dpoint_points_array );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DPOINT_POINTS_ARRAY
    -> PRO_E_DPOINT_POINT
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_POINT *** " );
status = ProElementAlloc ( PRO_E_DPOINT_POINT, &pro_e_dpoint_point );
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_dpoint_points_array, NULL,
                                pro_e_dpoint_point );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DPOINT_POINTS_ARRAY
    -> PRO_E_DPOINT_POINT
    -> PRO_E_DPOINT_POINT_NAME
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_POINT_NAME *** " );
status = ProElementAlloc ( PRO_E_DPOINT_POINT_NAME,
                          &pro_e_dpoint_point_name );
C_LOG( " ProElementAlloc " );
ProStringToWstring ( wide_string, "PNT0" );
value_data.type = PRO_VALUE_TYPE_WSTRING;
value_data.v.w = wide_string;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );

```

```

C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_dpoint_point_name, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_dpoint_point, NULL,
                                pro_e_dpoint_point_name );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DPOINT_POINTS_ARRAY
    -> PRO_E_DPOINT_POINT
    -> PRO_E_DPOINT_PLA_CONSTRAINTS
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_PLA_CONSTRAINTS *** " );
status = ProElementAlloc ( PRO_E_DPOINT_PLA_CONSTRAINTS,
                          &pro_e_dpoint_pla_constraints );
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_dpoint_point, NULL,
                                pro_e_dpoint_pla_constraints );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DPOINT_POINTS_ARRAY
    -> PRO_E_DPOINT_POINT
    -> PRO_E_DPOINT_PLA_CONSTRAINTS
    -> PRO_E_DPOINT_PLA_CONSTRAINT
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_PLA_CONSTRAINT *** " );
status = ProElementAlloc ( PRO_E_DPOINT_PLA_CONSTRAINT,
                          &pro_e_dpoint_pla_constraint );
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_dpoint_pla_constraints, NULL,
                                pro_e_dpoint_pla_constraint );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DPOINT_POINTS_ARRAY
    -> PRO_E_DPOINT_POINT
    -> PRO_E_DPOINT_PLA_CONSTRAINTS
    -> PRO_E_DPOINT_PLA_CONSTRAINT
    -> PRO_E_DPOINT_PLA_CONSTR_REF
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_PLA_CONSTR_REF *** " );
status = ProMessageDisplay ( message_file, "Select a reference
                                Surface");
C_LOG( " ProMessageDisplay" );
ProTKPrintf ( "Please select datum,surface,sldface,qltface_ID_5 type of
                Modelitem\n");
status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL, NULL,
                   NULL, NULL, &p_select, &n_select );
C_LOG( " ProSelect" );
if ( n_select <= 0 ) return -1;

```

```

status = ProElementAlloc ( PRO_E_DPOINT_PLA_CONSTR_REF,
                          &pro_e_dpoint_pla_constr_ref );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_dpoint_pla_constr_ref, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_dpoint_pla_constraint, NULL,
                                pro_e_dpoint_pla_constr_ref );
C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Populating element PRO_E_DPOINT_POINTS_ARRAY
    -> PRO_E_DPOINT_POINT
        -> PRO_E_DPOINT_PLA_CONSTRAINTS
            -> PRO_E_DPOINT_PLA_CONSTRAINT
                -> PRO_E_DPOINT_PLA_CONSTR_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_PLA_CONSTR_TYPE *** " );
status = ProElementAlloc ( PRO_E_DPOINT_PLA_CONSTR_TYPE,
                          &pro_e_dpoint_pla_constr_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_DTMPNT_CONSTR_TYPE_ON; /* 0 ProDtmpntConstrType */
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_dpoint_pla_constr_type, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_dpoint_pla_constraint, NULL,
                                pro_e_dpoint_pla_constr_type );
C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Populating element PRO_E_DPOINT_POINTS_ARRAY
    -> PRO_E_DPOINT_POINT
        -> PRO_E_DPOINT_PLA_CONSTRAINTS
            -> PRO_E_DPOINT_PLA_CONSTRAINT
                -> PRO_E_DPOINT_PLA_CONSTR_VAL
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_PLA_CONSTR_VAL *** " );
status = ProElementAlloc ( PRO_E_DPOINT_PLA_CONSTR_VAL,
                          &pro_e_dpoint_pla_constr_val );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 0.000000;

```



```

status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_dpoint_pla_constr_val, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_dpoint_pla_constraint, NULL,
                                pro_e_dpoint_pla_constr_val );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DPOINT_POINTS_ARRAY
    -> PRO_E_DPOINT_POINT
    -> PRO_E_DPOINT_PLA_CONSTRAINTS
    -> PRO_E_DPOINT_PLA_CONSTRAINT
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_PLA_CONSTRAINT *** " );
status = ProElementAlloc ( PRO_E_DPOINT_PLA_CONSTRAINT,
                          &pro_e_dpoint_pla_constraint );
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_dpoint_pla_constraints, NULL,
                                pro_e_dpoint_pla_constraint );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DPOINT_POINTS_ARRAY
    -> PRO_E_DPOINT_POINT
    -> PRO_E_DPOINT_PLA_CONSTRAINTS
    -> PRO_E_DPOINT_PLA_CONSTRAINT
    -> PRO_E_DPOINT_PLA_CONSTR_REF
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_PLA_CONSTR_REF *** " );
status = ProMessageDisplay ( message_file, "Select a reference
                                Surface");
C_LOG( " ProMessageDisplay" );
ProTKPrintf ( "Please select datum,surface,sldface,qltface_ID_5 type of
                                Modelitem\n");
status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL, NULL,
                    NULL, NULL, &p_select, &n_select );
C_LOG( " ProSelect" );
if ( n_select <= 0 ) return -1;
status = ProElementAlloc ( PRO_E_DPOINT_PLA_CONSTR_REF,
                          &pro_e_dpoint_pla_constr_ref );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_dpoint_pla_constr_ref, value );

```

```

C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_dpoint_pla_constraint, NULL,
                                pro_e_dpoint_pla_constr_ref );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DPOINT_POINTS_ARRAY
    -> PRO_E_DPOINT_POINT
        -> PRO_E_DPOINT_PLA_CONSTRAINTS
            -> PRO_E_DPOINT_PLA_CONSTRAINT
                -> PRO_E_DPOINT_PLA_CONSTR_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_PLA_CONSTR_TYPE *** " );
status = ProElementAlloc ( PRO_E_DPOINT_PLA_CONSTR_TYPE,
                           &pro_e_dpoint_pla_constr_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_DTMPNT_CONSTR_TYPE_ON; /* 0 ProDtmpntConstrType */
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_dpoint_pla_constr_type, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_dpoint_pla_constraint, NULL,
                                pro_e_dpoint_pla_constr_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DPOINT_POINTS_ARRAY
    -> PRO_E_DPOINT_POINT
        -> PRO_E_DPOINT_PLA_CONSTRAINTS
            -> PRO_E_DPOINT_PLA_CONSTRAINT
                -> PRO_E_DPOINT_PLA_CONSTR_VAL
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_PLA_CONSTR_VAL *** " );
status = ProElementAlloc ( PRO_E_DPOINT_PLA_CONSTR_VAL,
                           &pro_e_dpoint_pla_constr_val );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 0.000000;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_dpoint_pla_constr_val, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_dpoint_pla_constraint, NULL,
                                pro_e_dpoint_pla_constr_val );
C_LOG( " ProElemtreeElementAdd" );

```

```

/*-----*\
    Populating element PRO_E_DPOINT_POINTS_ARRAY
    -> PRO_E_DPOINT_POINT
    -> PRO_E_DPOINT_PLA_CONSTRAINTS
    -> PRO_E_DPOINT_PLA_CONSTRAINT
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_PLA_CONSTRAINT *** " );
status = ProElementAlloc ( PRO_E_DPOINT_PLA_CONSTRAINT,
                          &pro_e_dpoint_pla_constraint );
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_dpoint_pla_constraints, NULL,
                                pro_e_dpoint_pla_constraint );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DPOINT_POINTS_ARRAY
    -> PRO_E_DPOINT_POINT
    -> PRO_E_DPOINT_PLA_CONSTRAINTS
    -> PRO_E_DPOINT_PLA_CONSTRAINT
    -> PRO_E_DPOINT_PLA_CONSTR_REF
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_PLA_CONSTR_REF *** " );
status = ProMessageDisplay ( message_file, "Select a reference
                                Surface");
C_LOG( " ProMessageDisplay" );
ProTKPrintf ( "Please select datum,surface,sldface,qltface_ID_5 type of
                                Modelitem\n");
status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL, NULL,
                    NULL, NULL, &p_select, &n_select );
C_LOG( " ProSelect" );
if ( n_select <= 0 ) return -1;
status = ProElementAlloc ( PRO_E_DPOINT_PLA_CONSTR_REF,
                          &pro_e_dpoint_pla_constr_ref );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_dpoint_pla_constr_ref, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_dpoint_pla_constraint, NULL,
                                pro_e_dpoint_pla_constr_ref );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DPOINT_POINTS_ARRAY
    -> PRO_E_DPOINT_POINT
    -> PRO_E_DPOINT_PLA_CONSTRAINTS
    -> PRO_E_DPOINT_PLA_CONSTRAINT

```

```

        -> PRO_E_DPOINT_PLA_CONSTR_TYPE
/*-----*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_PLA_CONSTR_TYPE *** " );
status = ProElementAlloc ( PRO_E_DPOINT_PLA_CONSTR_TYPE,
                          &pro_e_dpoint_pla_constr_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_DTMPNT_CONSTR_TYPE_ON; /* 0 ProDtmpntConstrType */
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_dpoint_pla_constr_type, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_dpoint_pla_constraint, NULL,
                                pro_e_dpoint_pla_constr_type );
C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Populating element PRO_E_DPOINT_POINTS_ARRAY
        -> PRO_E_DPOINT_POINT
            -> PRO_E_DPOINT_PLA_CONSTRAINTS
                -> PRO_E_DPOINT_PLA_CONSTRAINT
                    -> PRO_E_DPOINT_PLA_CONSTR_VAL
/*-----*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_PLA_CONSTR_VAL *** " );
status = ProElementAlloc ( PRO_E_DPOINT_PLA_CONSTR_VAL,
                          &pro_e_dpoint_pla_constr_val );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 0.000000;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_dpoint_pla_constr_val, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_dpoint_pla_constraint, NULL,
                                pro_e_dpoint_pla_constr_val );
C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Create the feature in the current model.
/*-----*/
status = ProMdlCurrentGet (&model);
if ( status != PRO_TK_NO_ERROR ) return ( status );
status = ProMdlToModelitem( model, &model_item );
status = ProSelectionAlloc ( p_comp_path, &model_item, &model_sel);

opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMS;
status = ProFeatureCreate (model_sel, pro_e_feature_tree, opts, 1,

```

```

&feature, &errors);
C_LOG (" ProFeatureCreate");

status = ProElementFree (&pro_e_feature_tree );
C_LOG (" ProElementFree");

return (status);
}

```

Examples

Point on a Vertex

To create a datum point on the vertex, the following constraints are required.

Placement Constraint Element	Reference Element	Valid Value
PRO_E_DPOINT_PLA_CONSTRAINT (Constraint 1)	PRO_E_DPOINT_PLA_CONSTR_REF	Vertex
	PRO_E_DPOINT_PLA_CONSTR_TYPE	PRO_DTMPNT_CONSTR_TYPE_ON
	PRO_E_DPOINT_PLA_CONSTR_VAL	Not applicable

Offset Point

To create one or more datum points at an offset, the following constraints are required.

Placement Constraint Element	Reference Element	Valid Value
PRO_E_DPOINT_PLA_CONSTRAINT (Constraint 1)	PRO_E_DPOINT_PLA_CONSTR_REF	Vertex, Csys, or DPnt
	PRO_E_DPOINT_PLA_CONSTR_TYPE	PRO_DTMPNT_CONSTR_TYPE_OFFSET
	PRO_E_DPOINT_PLA_CONSTR_VAL	Offset value.

The following tables provide valid values for Constraint 2. You can create a point at an offset using values from one of the following tables for Constraint 2.

Placement Constraint Element	Reference Element	Valid Value
PRO_E_DPOINT_PLA_ CONSTRAINT (Constraint 2)	PRO_E_DPOINT_PLA_CO NSTR_REF	Curve, Edge or Axis
	PRO_E_DPOINT_PLA_CO NSTR_TYPE	PRO_DTMPNT_CONSTR _TYPE_PARALLEL
	PRO_E_DPOINT_PLA_CO NSTR_VAL	Not applicable

OR

Placement Constraint Element	Reference Element	Valid Value
PRO_E_DPOINT_PLA_ CONSTRAINT (Constraint 2)	PRO_E_DPOINT_PLA_CO NSTR_REF	Surface
	PRO_E_DPOINT_PLA_CO NSTR_TYPE	PRO_DTMPNT_CONSTR _TYPE_NORMAL
	PRO_E_DPOINT_PLA_CO NSTR_VAL	Not applicable

OR

Placement Constraint Element	Reference Element	Valid Value
PRO_E_DPOINT_PLA_ CONSTRAINT (Constraint 2)	PRO_E_DPOINT_PLA_CO NSTR_REF	Csys Axis
	PRO_E_DPOINT_PLA_CO NSTR_TYPE	PRO_DTMPNT_CONSTR _TYPE_PARALLEL
	PRO_E_DPOINT_PLA_CO NSTR_VAL	Not applicable

OR

Placement Constraint Element	Reference Element	Valid Value
PRO_E_DPOINT_PLA_ CONSTRAINT (Constraint 2)	PRO_E_DPOINT_PLA_CO NSTR_REF	Csys
	PRO_E_DPOINT_PLA_CO NSTR_TYPE	PRO_DTMPNT_CONSTR _TYPE_CARTESIAN or PRO_DTMPNT_CONSTR _TYPE_CYLINDRICAL or PRO_DTMPNT_CONSTR _TYPE_SPHERICAL
	PRO_E_DPOINT_PLA_CO NSTR_VAL	Not applicable

Element Trees:
Datum Features

The following table provides valid values for dimension constraints.

Dimension Constraint Element	Reference Element	Valid Value
PRO_E_DPOINT_DIM_ CONSTRAINT (Constraint 3)	PRO_E_DPOINT_DIM_CO NSTR_REF	Not applicable
	PRO_E_DPOINT_DIM_CO NSTR_TYPE	PRO_DTMPNT_CONSTR _TYPE_ALONG_X
	PRO_E_DPOINT_DIM_CO NSTR_VAL	Offset Value

Dimension Constraint Element	Reference Element	Valid Value
PRO_E_DPOINT_DIM_ CONSTRAINT (Constraint 4)	PRO_E_DPOINT_DIM_CO NSTR_REF	Not applicable
	PRO_E_DPOINT_DIM_CO NSTR_TYPE	PRO_DTMPNT_CONSTR _TYPE_ALONG_Y
	PRO_E_DPOINT_DIM_CO NSTR_VAL	Offset Value

Dimension Constraint Element	Reference Element	Valid Value
PRO_E_DPOINT_DIM _CONSTRAINT (Constraint 5)	PRO_E_DPOINT_DIM_CO NSTR_REF	Not applicable
	PRO_E_DPOINT_DIM_CO NSTR_TYPE	PRO_DTMPNT_CONSTR _TYPE_ALONG_Z
	PRO_E_DPOINT_DIM_CO NSTR_VAL	Offset Value

As per Offset types the values are as follows—

	CARTESIAN	CYLINDRICAL	SPHERICAL
PRO_DTMPNT_CONS TR_TYPE_ALONG_X	X	R	RHO
PRO_DTMPNT_CONS TR_TYPE_ALONG_Y	Y	THETA	PHI
PRO_DTMPNT_CONS TR_TYPE_ALONG_Z	Z	Z	THETA

Point at Intersection of Three Surfaces

To create a datum point at the intersection of three surfaces, use the following constraints. Each surface can be a part surface, surface feature, or datum plane.

Placement Constraint Element	Reference Element	Valid Value
PRO_E_DPOINT_PLA_ CONSTRAINT (Constraint 1)	PRO_E_DPOINT_PLA_CO NSTR_REF	Surface
	PRO_E_DPOINT_PLA_CO NSTR_TYPE	PRO_DTMPNT_CONSTR _TYPE_ON
	PRO_E_DPOINT_PLA_CO NSTR_VAL	Not applicable
PRO_E_DPOINT_PLA_ CONSTRAINT (Constraint 2)	PRO_E_DPOINT_PLA_CO NSTR_REF	Surface
	PRO_E_DPOINT_PLA_CO NSTR_TYPE	PRO_DTMPNT_CONSTR _TYPE_ON
	PRO_E_DPOINT_PLA_CO NSTR_VAL	Not applicable

Placement Constraint Element	Reference Element	Valid Value
PRO_E_DPOINT_PLA_ CONSTRAINT (Constraint 3)	PRO_E_DPOINT_PLA_CO NSTR_REF	Surface
	PRO_E_DPOINT_PLA_CO NSTR_TYPE	PRO_DTMPNT_CONSTR _TYPE_ON
	PRO_E_DPOINT_PLA_CO NSTR_VAL	Not applicable

Point On a Surface or Offset from a Surface

The following constraints are required to create a point on a surface or at an offset distance from a surface:

Placement Constraint Element	Reference Element	Valid Value
PRO_E_DPOINT_PLA_ CONSTRAINT (Constraint 1)	PRO_E_DPOINT_PLA_CO NSTR_REF	Surface
	PRO_E_DPOINT_PLA_CO NSTR_TYPE	PRO_DTMPNT_CONSTR _TYPE_ON or PRO_DTMPNT_CONSTR _TYPE_OFFSET
	PRO_E_DPOINT_PLA_CO NSTR_VAL	Not applicable
PRO_E_DPOINT_PLA_ CONSTRAINT (Constraint 2)	PRO_E_DPOINT_PLA_CO NSTR_REF	Edge or Surface
	PRO_E_DPOINT_PLA_CO NSTR_TYPE	PRO_DTMPNT_CONSTR _TYPE_OFFSET
	PRO_E_DPOINT_PLA_CO NSTR_VAL	Offset value
PRO_E_DPOINT_PLA_ CONSTRAINT (Constraint 3)	PRO_E_DPOINT_PLA_CO NSTR_REF	Edge or Surface
	PRO_E_DPOINT_PLA_CO NSTR_TYPE	PRO_DTMPNT_CONSTR _TYPE_OFFSET
	PRO_E_DPOINT_PLA_CO NSTR_VAL	Offset value

Point at Intersection of a Curve and a Surface

To create a datum point at the intersection of a curve and a surface, use the following constraints. The curve can be a part edge, surface feature edge, datum curve, axis, or an imported datum curve. The surface can be a part surface, surface feature, or datum plane.

Placement Constraint Element	Reference Element	Valid Value
PRO_E_DPOINT_PLA_ CONSTRAINT (Constraint 1)	PRO_E_DPOINT_PLA_CO NSTR_REF	Curve, axis, edge, or surface
	PRO_E_DPOINT_PLA_CO NSTR_TYPE	PRO_DTMPNT_CONSTR _TYPE_ON
	PRO_E_DPOINT_PLA_CO NSTR_VAL	Not applicable
PRO_E_DPOINT_PLA_ CONSTRAINT (Constraint 2)	PRO_E_DPOINT_PLA_CO NSTR_REF	- If value of constraint 1 is Curve, Axis, or Edge, the value of constraint 2 is surface. - If value of constraint 1 is surface, the value of constraint 2 is Curve, Axis, or Edge.
	PRO_E_DPOINT_PLA_CO NSTR_TYPE	PRO_DTMPNT_CONSTR _TYPE_ON
	PRO_E_DPOINT_PLA_CO NSTR_VAL	Not applicable

Note: If more than one intersections exist, the point is created at the intersection nearest to the curve reference parameter value.

Point At Center of Curve or Surface

To create a datum point at the center of an arc or circle entity, use the following constraints.

Placement Constraint Element	Reference Element	Valid Value
PRO_E_DPOINT_PLA_CONSTRAINT (Constraint 1)	PRO_E_DPOINT_PLA_CONSTR_REF	Curve, edge, or surface (Sphere)
	PRO_E_DPOINT_PLA_CONSTR_TYPE	PRO_DTMPNT_CONSTR_TYPE_CENTER
	PRO_E_DPOINT_PLA_CONSTR_VAL	Not applicable

Point at Intersection of Two Curves

To create a point at intersection of two curves, use the following constraints.

Placement Constraint Element	Reference Element	Valid Value
PRO_E_DPOINT_PLA_CONSTRAINT (Constraint 1)	PRO_E_DPOINT_PLA_CONSTR_REF	Curve, edge, or axis
	PRO_E_DPOINT_PLA_CONSTR_TYPE	PRO_DTMPNT_CONSTR_TYPE_ON
	PRO_E_DPOINT_PLA_CONSTR_VAL	Not applicable
PRO_E_DPOINT_PLA_CONSTRAINT (Constraint 2)	PRO_E_DPOINT_PLA_CONSTR_REF	Curve, edge, or axis
	PRO_E_DPOINT_PLA_CONSTR_TYPE	PRO_DTMPNT_CONSTR_TYPE_ON
	PRO_E_DPOINT_PLA_CONSTR_VAL	Not applicable

Note: If more than one intersections exist, the point is created at the intersection nearest to the second reference parameter value.

Point On Curve

To create a datum point on a curve, the following constraints are required.

Placement Constraint Element	Reference Element	Valid Value
PRO_E_DPOINT_PLA_ CONSTRAINT (Constraint 1)	PRO_E_DPOINT_PLA_CO NSTR_REF	Curve, edge, or axis (It is valid with offset plane)
	PRO_E_DPOINT_PLA_CO NSTR_TYPE	PRO_DTMPNT_CONSTR _TYPE_ON
	PRO_E_DPOINT_PLA_CO NSTR_VAL	Not applicable

The following tables provide valid values for constraint 2. You can create a point on curve using values from one of the following tables for constraint 2.

Use the following values for constraint 2 if the length of curve from the start point or the end point is used to locate the point.

Placement Constraint Element	Reference Element	Valid Value
PRO_E_DPOINT_PLA_ CONSTRAINT (Constraint 2)	PRO_E_DPOINT_PLA_CO NSTR_REF	Curve (Use the same curve as used in Constraint 1)
	PRO_E_DPOINT_PLA_CO NSTR_TYPE	PRO_DTMPNT_CONSTR _TYPE_LENGTH or PRO_DTMPNT_CONSTR _TYPE_LENGTH_END
	PRO_E_DPOINT_PLA_CO NSTR_VAL	Length value (from curve start point or end point)

Use the following values for constraint 2 if the ratio of distance from the start point or the end point is used to locate the point.

Placement Constraint Element	Reference Element	Valid Value
PRO_E_DPOINT_PLA_CONSTRAINT (Constraint 2)	PRO_E_DPOINT_PLA_CONSTR_REF	Curve (Use the same curve as used in contrarian 1)
	PRO_E_DPOINT_PLA_CONSTR_TYPE	PRO_DTMPNT_CONSTR_TYPE_RATIO or PRO_DTMPNT_CONSTR_TYPE_RATIO_END
	PRO_E_DPOINT_PLA_CONSTR_VAL	Ratio value (from curve start or end)

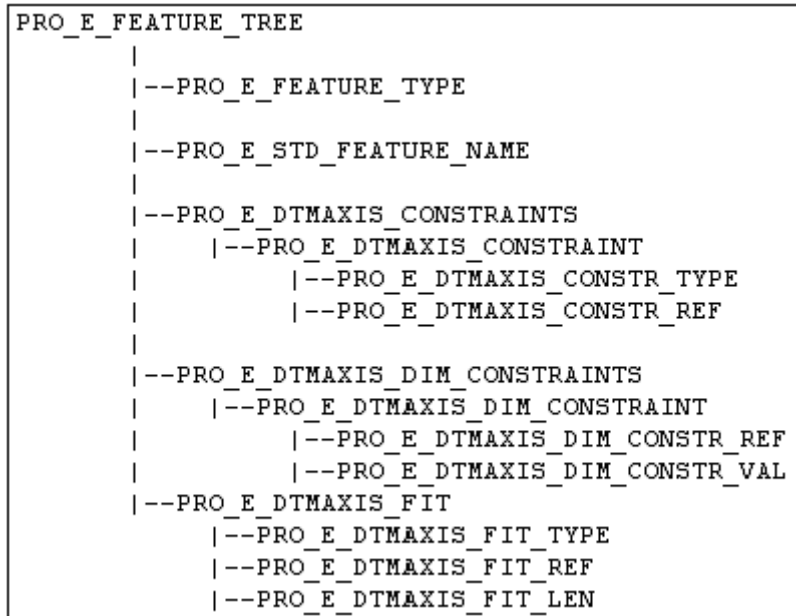
Element Trees:
Datum Features

Use the following values for constraint 2 if the offset surface is used to locate the point on curve.

Placement Constraint Element	Reference Element	Valid Value
PRO_E_DPOINT_PLA_CONSTRAINT (Constraint 2)	PRO_E_DPOINT_PLA_CONSTR_REF	Surface
	PRO_E_DPOINT_PLA_CONSTR_TYPE	PRO_DTMPNT_CONSTR_TYPE_OFFSET
	PRO_E_DPOINT_PLA_CONSTR_VAL	Offset value

Datum Axis Features

The basic element tree for creating axes is available in the include file *ProDtmAxis.h*. The following figure shows the basic structure of the element tree.



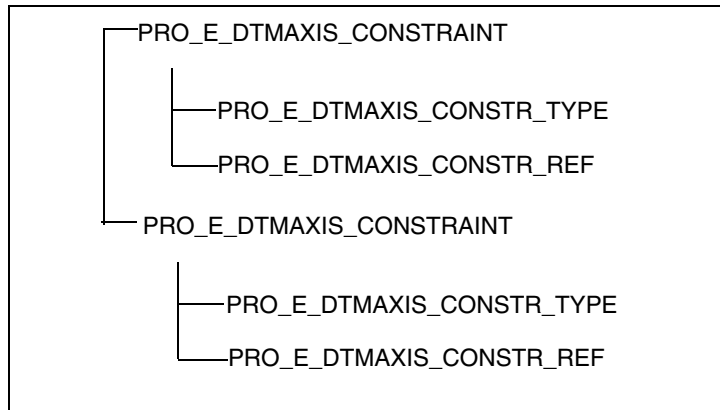
Pro/TOOLKIT supports creation of the following types of datum axes:

- Tangent
- Through Edge or Surface
- Two Planes
- Two Points
- Normal Planes

There is no single element that indicates the type in constraints element tree. The type is determined implicitly based on the constraint type and references.

Point on Surface

The element tree structure of the axis, created with type as `PRO_DTMAXIS_PNT_SURF`, is shown in the following figure.

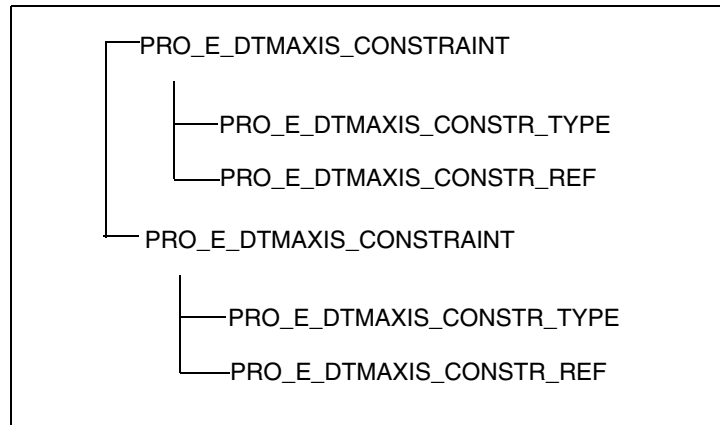


The following table specifies the constraints for the PRO_E_DTMAXIS_CONSTRAINT elements in the element tree for the point on surface type of axis.

Element ID	Data Type	Valid Value
PRO_E_DTMAXIS_CONSTRAINT (Constraint 1)	PRO_E_DTMAXIS_CONSTR_TYPE	PRO_DTMAXIS_CONSTRAINT_TYPE_THRU
	PRO_E_DTMAXIS_CONSTR_REF	PRO_POINT
PRO_E_DTMAXIS_CONSTRAINT (Constraint 2)	PRO_E_DTMAXIS_CONSTR_TYPE	PRO_DTMAXIS_CONSTRAINT_TYPE_NORMAL
	PRO_E_DTMAXIS_CONSTR_REF	PRO_SURFACE

Tangent

The element tree structure of the axis, created with type as Tangent, is shown in the following figure.

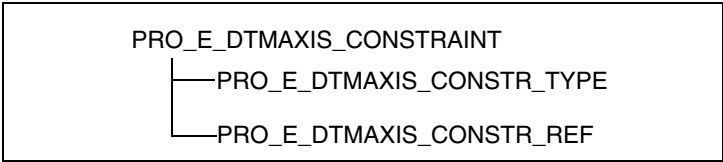


The following table specifies the constraints for the PRO_E_DTMAXIS_CONSTRAINT elements in the element tree for the tangent type of axis.

Element ID	Data Type	Valid Value
PRO_E_DTMAXIS_CONSTRAINT (Constraint 1)	PRO_E_DTMAXIS_CONSTR_TYPE	PRO_DTMAXIS_CONSTRAINT_TYPE_TANGENT
	PRO_E_DTMAXIS_CONSTR_REF	PRO_EDGE, PRO_CURVE
PRO_E_DTMAXIS_CONSTRAINT (Constraint 2)	PRO_E_DTMAXIS_CONSTR_TYPE	PRO_DTMAXIS_CONSTRAINT_TYPE_THRU
	PRO_E_DTMAXIS_CONSTR_REF	PRO_POINT, PRO_EDGE_START, PRO_EDGE_END,PRO_CRV_START, PRO_CRV_END

Through Edge or Surface

The element tree structure of the axis, created with type Through an Edge or a Surface, is shown in the following figure.

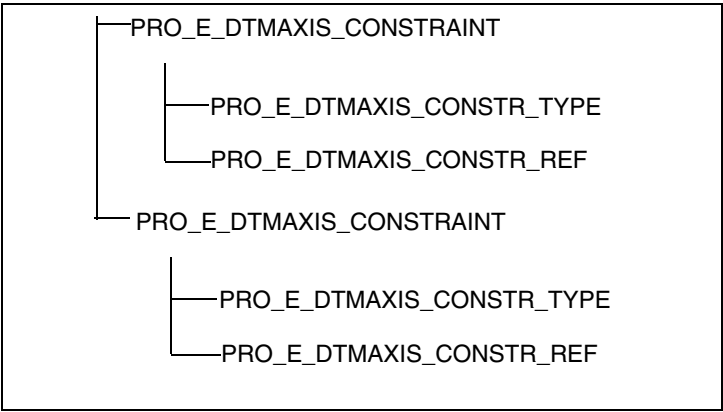


The following table specifies the constraints for the PRO_E_DTMAXIS_CONSTRAINT elements in the element tree for the through type of axis.

Element ID	Data Type	Valid Value
PRO_E_DTMAXIS_CONSTRAINT (Constraint 1)	PRO_E_DTMAXIS_CONSTR_TYPE	PRO_DTMAXIS_CONSTR_TYPE_THRU
	PRO_E_DTMAXIS_CONSTR_REF	PRO_EDGE (Straight), PRO_SURFACE (Cylinder)

Two Planes

The element tree structure of the axis, created using the type as two planes, is as shown in the following figure.



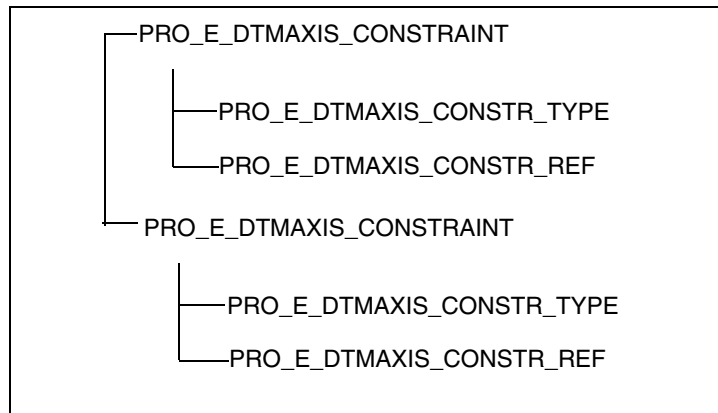
Element Trees:
Datum Features

The following table specifies the constraints for the PRO_E_DTMAXIS_CONSTRAINT elements in the element tree for the two planes reference scheme.

Element ID	Data Type	Valid Value
PRO_E_DTMAXIS_CONSTRAINT (Constraint 1)	PRO_E_DTMAXIS_CONSTR_TYPE	PRO_DTMAXIS_CONSTRAINT_TYPE_THRU
	PRO_E_DTMAXIS_CONSTR_REF	PRO_SURFACE (Planar)
PRO_E_DTMAXIS_CONSTRAINT (Constraint 2)	PRO_E_DTMAXIS_CONSTR_TYPE	PRO_DTMAXIS_CONSTRAINT_TYPE_THRU
	PRO_E_DTMAXIS_CONSTR_REF	PRO_SURFACE (Planar)

Two Points

The element tree structure of the axis, created using the type as two points, is shown in the following figure.



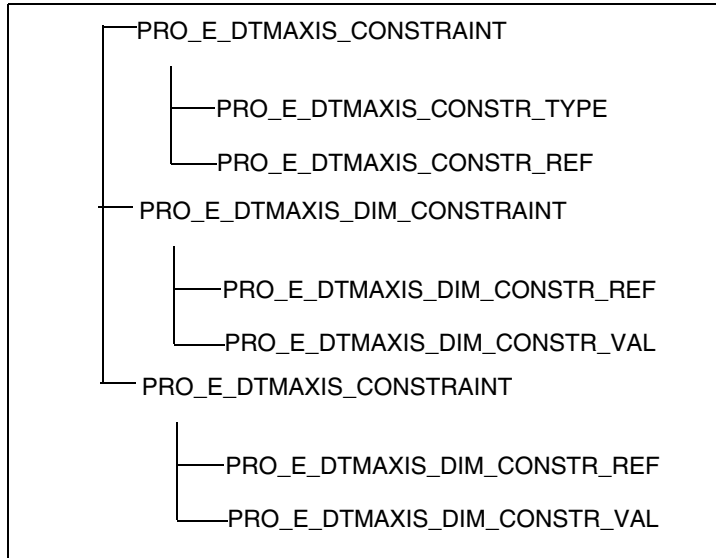
The following table specifies the constraints for the PRO_E_DTMAXIS_CONSTRAINT elements in the element tree for the two points type of axis.

Element ID	Data Type	Valid Value
PRO_E_DTMAXIS_CONSTRAINT (Constraint 1)	PRO_E_DTMAXIS_CONSTR_TYPE	PRO_DTMAXIS_CONSTR_TYPE_THRU
	PRO_E_DTMAXIS_CONSTR_REF	PRO_POINT, PRO_EDGE_START, PRO_EDGE_END, PRO_CRV_START, PRO_CRV_END
PRO_E_DTMAXIS_CONSTRAINT (Constraint 2)	PRO_E_DTMAXIS_CONSTR_TYPE	PRO_DTMAXIS_CONSTR_TYPE_THRU
	PRO_E_DTMAXIS_CONSTR_REF	PRO_POINT, PRO_EDGE_START, PRO_EDGE_END, PRO_CRV_START, PRO_CRV_END

Element Trees:
Datum Features

Normal Planes

The element tree structure of the axis, created using the type as Normal Planes, is shown in the following figure.



The following table specifies the constraints for the PRO_E_DTMAXIS_CONSTRAINT elements in the element tree for the normal plane type of axis.

Element ID	Data Type	Valid Value
PRO_E_DTMAXIS_CONSTRAINT (Constraint 1)	PRO_E_DTMAXIS_CONSTR_TYPE	PRO_DTMAXIS_CONSTR_TYPE_NORMAL
	PRO_E_DTMAXIS_CONSTR_REF	PRO_SURFACE (Planar)
PRO_E_DTMAXIS_DIM_CONSTRAINT (Constraint 2)	PRO_E_DTMAXIS_DIM_CONSTR_REF	PRO_SURFACE (Planar), PRO_AXIS, PRO_EDGE
	PRO_E_DTMAXIS_DIM_CONSTR_VAL	Valid dimension
PRO_E_DTMAXIS_DIM_CONSTRAINT (Constraint 3)	PRO_E_DTMAXIS_DIM_CONSTR_REF	PRO_SURFACE (Planar), PRO_AXIS, PRO_EDGE
	PRO_E_DTMAXIS_DIM_CONSTR_VAL	Valid dimension

Example 6: Creating a Datum Axis

This example shows how to create a datum axis at the intersection of two selected surfaces. The user is prompted to select the two surfaces.

```

/*=====*\
Creating a Datum Axis feature
\*=====*/
/*----- Pro/Toolkit Includes -----*/
#include "ProToolkit.h"
#include "ProFeature.h"
#include "ProElemId.h"
#include "ProExtrude.h"
#include "ProModFeat.h"
#include "ProStdSection.h"
#include "ProElement.h"
#include "ProElempath.h"
#include "ProFeatType.h"
#include "ProFeatForm.h"
#include "ProSelection.h"
#include "ProSection.h"

#include "ProDtmAxis.h"

#define C_LOG(a) ProTKPrintf ("Status for %s is = %d\n", a, status ); \

```

```

ERROR_CHECK( a, "UgSktExtrusionCreate.c", status );

#define C_PRINT(a) ProTKPrintf ( "%s\n", a);

static ProFileName message_file;

/*----- Function Prototypes -----*/
ProError ProDemoDatumAxisCreate();

/*----- External Data -----*/

/*----- Global Data -----*/

/*=====*\
FUNCTION : ProDemoDatumAxisCreate
PURPOSE  : Demonstrates the creation of Datum Axis Feature
\*=====*/
ProError ProDemoDatumAxisCreate()
{
    ProErrorlist      errors;
    ProMdl             model;
    ProModelitem       model_item;
    ProSelection       model_sel;
    ProFeature         feature;
    ProFeatureCreateOptions opts[1];
    ProAsmcomppath     *p_comp_path = NULL;
    ProValue           value;
    char               name[PRO_NAME_SIZE];
    ProError           status;

    ProElement pro_e_feature_tree;
    ProElement pro_e_feature_type;
    ProElement pro_e_std_feature_name;
    ProElement pro_e_dtmaxis_constraints;
    ProElement pro_e_dtmaxis_constraint;
    ProElement pro_e_dtmaxis_constr_type;
    ProElement pro_e_dtmaxis_constr_ref;

    ProName wide_string;
    ProValueData value_data;
    ProSelection * p_select;
    int n_select;
    ProBoolean is_interactive = PRO_B_TRUE;

    ProStringToWstring ( message_file, "utilities.txt" );

    /*-----*\
    Populating root element PRO_E_FEATURE_TREE
    \*-----*/
    C_PRINT( " *** Processing Element PRO_E_FEATURE_TREE *** " );

```

```

status = ProElementAlloc ( PRO_E_FEATURE_TREE, &pro_e_feature_tree );
C_LOG( " ProElementAlloc " );

/*-----*\
    Populating element PRO_E_FEATURE_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TYPE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TYPE, &pro_e_feature_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_FEAT_DATUM_AXIS; /* 926 */
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_feature_type, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_feature_type );
C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Populating element PRO_E_STD_FEATURE_NAME
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_FEATURE_NAME *** " );
status = ProElementAlloc ( PRO_E_STD_FEATURE_NAME,
                          &pro_e_std_feature_name );
C_LOG( " ProElementAlloc " );
ProStringToWstring ( wide_string, "MY_A_1" );
value_data.type = PRO_VALUE_TYPE_WSTRING;
value_data.v.w = wide_string;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_std_feature_name, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_std_feature_name );
C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Populating element PRO_E_DTMAXIS_CONSTRAINTS
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DTMAXIS_CONSTRAINTS *** " );
status = ProElementAlloc ( PRO_E_DTMAXIS_CONSTRAINTS,
                          &pro_e_dtmaxis_constraints );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_dtmaxis_constraints );
C_LOG( " ProElemtreeElementAdd " );

```

```

/*-----*\
    Populating element PRO_E_DTMAXIS_CONSTRAINTS
    -> PRO_E_DTMAXIS_CONSTRAINT
/*-----*/
C_PRINT( " *** Processing Element PRO_E_DTMAXIS_CONSTRAINT *** " );
status = ProElementAlloc ( PRO_E_DTMAXIS_CONSTRAINT,
                          &pro_e_dtmaxis_constraint );
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_dtmaxis_constraints, NULL,
                                pro_e_dtmaxis_constraint );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DTMAXIS_CONSTRAINTS
    -> PRO_E_DTMAXIS_CONSTRAINT
    -> PRO_E_DTMAXIS_CONSTR_TYPE
/*-----*/
C_PRINT( " *** Processing Element PRO_E_DTMAXIS_CONSTR_TYPE *** " );
status = ProElementAlloc ( PRO_E_DTMAXIS_CONSTR_TYPE,
                          &pro_e_dtmaxis_constr_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_DTMAXIS_CONSTR_TYPE_THRU;
/* 1 ProDtmaxisConstrType */
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_dtmaxis_constr_type, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_dtmaxis_constraint, NULL,
                                pro_e_dtmaxis_constr_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DTMAXIS_CONSTRAINTS
    -> PRO_E_DTMAXIS_CONSTRAINT
    -> PRO_E_DTMAXIS_CONSTR_REF
/*-----*/
C_PRINT( " *** Processing Element PRO_E_DTMAXIS_CONSTR_REF *** " );
status = ProMessageDisplay ( message_file, "Select a reference
                                Surface");
C_LOG( " ProMessageDisplay" );
ProTKPrintf ( "Please select datum,surface,sldface,qltface_ID_5 type of
                Modelitem\n");
status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL, NULL,
                    NULL, NULL, &p_select, &n_select );
C_LOG( " ProSelect" );
if ( n_select <= 0 ) return -1;

```

```

status = ProElementAlloc ( PRO_E_DTMAXIS_CONSTR_REF,
                           &pro_e_dtmaxis_constr_ref );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_dtmaxis_constr_ref, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_dtmaxis_constraint, NULL,
                                pro_e_dtmaxis_constr_ref );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DTMAXIS_CONSTRAINTS
    -> PRO_E_DTMAXIS_CONSTRAINT
    -> PRO_E_DTMAXIS_CONSTR_REF
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DTMAXIS_CONSTRAINT *** " );
status = ProElementAlloc ( PRO_E_DTMAXIS_CONSTRAINT,
                           &pro_e_dtmaxis_constraint );
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_dtmaxis_constraints, NULL,
                                pro_e_dtmaxis_constraint );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DTMAXIS_CONSTRAINTS
    -> PRO_E_DTMAXIS_CONSTRAINT
    -> PRO_E_DTMAXIS_CONSTR_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DTMAXIS_CONSTR_TYPE *** " );
status = ProElementAlloc ( PRO_E_DTMAXIS_CONSTR_TYPE,
                           &pro_e_dtmaxis_constr_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_DTMAXIS_CONSTR_TYPE_THRU;
/* 1 ProDtmaxisConstrType */
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_dtmaxis_constr_type, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_dtmaxis_constraint, NULL,
                                pro_e_dtmaxis_constr_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\

```



```

    Populating element PRO_E_DTMAXIS_CONSTRAINTS
    -> PRO_E_DTMAXIS_CONSTRAINT
    -> PRO_E_DTMAXIS_CONSTR_REF

/*-----*/
C_PRINT( " *** Processing Element PRO_E_DTMAXIS_CONSTR_REF *** " );
status = ProMessageDisplay ( message_file, "Select a reference
                               Surface");
C_LOG( " ProMessageDisplay" );
ProTKPrintf ( "Please select datum,surface,sldface,qltface_ID_5 type of
                Modelitem\n");
status = ProSelect ( "datum,surface,sldface,qltface_ID_5", 1, NULL,
                    NULL, NULL, NULL, &p_select, &n_select );
C_LOG( " ProSelect" );
if ( n_select <= 0 ) return -1;
status = ProElementAlloc ( PRO_E_DTMAXIS_CONSTR_REF,
                          &pro_e_dtmaxis_constr_ref );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_dtmaxis_constr_ref, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_dtmaxis_constraint, NULL,
                                pro_e_dtmaxis_constr_ref );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
Create the feature in the current model.
/*-----*/
status = ProMdlCurrentGet (&model);
if ( status != PRO_TK_NO_ERROR ) return ( status );
status = ProMdlToModelitem( model, &model_item );
status = ProSelectionAlloc (p_comp_path, &model_item, &model_sel);

opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMS;
status = ProFeatureCreate (model_sel, pro_e_feature_tree, opts, 1,
                          &feature, &errors);
C_LOG ( " ProFeatureCreate");

status = ProElementFree (&pro_e_feature_tree );
C_LOG ( " ProElementFree");

return (status);
}

```

Datum Coordinate System Features

The following figure illustrates the general structure of the element tree for coordinate system features.

```

PRO_E_FEATURE_TREE
|
|--PRO_E_FEATURE_TYPE
|
|--PRO_E_STD_FEATURE_NAME
|
|--PRO_E_CSYS_ORIGIN_CONSTRS
|   |--PRO_E_CSYS_ORIGIN_CONSTR
|       |--PRO_E_CSYS_ORIGIN_CONSTR_REF
|
|--PRO_E_CSYS_OFFSET_TYPE
|
|--PRO_E_CSYS_ONSURF_TYPE
|
|--PRO_E_CSYS_DIM_CONSTRS
|   |--PRO_E_CSYS_DIM_CONSTR
|       |--PRO_E_CSYS_DIM_CONSTR_REF
|       |--PRO_E_CSYS_DIM_CONSTR_TYPE
|       |--PRO_E_CSYS_DIM_CONSTR_VAL
|
|--PRO_E_CSYS_ORIENTMOVES
|   |--PRO_E_CSYS_ORIENTMOVE
|       |--PRO_E_CSYS_ORIENTMOVE_MOVE_TYPE
|       |--PRO_E_CSYS_ORIENTMOVE_MOVE_VAL
|
|--PRO_E_CSYS_NORMAL_TO_SCREEN
|
|--PRO_E_CSYS_ORIENT_BY_METHOD
|
|--PRO_E_CSYS_ORIENTSELAXIS1_REF
|--PRO_E_CSYS_ORIENTSELAXIS1_REF_OPT
|--PRO_E_CSYS_ORIENTSELAXIS1_OPT
|--PRO_E_CSYS_ORIENTSELAXIS1_FLIP
|--PRO_E_CSYS_ORIENTSELAXIS2_REF
|--PRO_E_CSYS_ORIENTSELAXIS2_REF_OPT
|--PRO_E_CSYS_ORIENTSELAXIS2_OPT
|--PRO_E_CSYS_ORIENTSELAXIS2_FLIP
|--PRO_E_CSYS_ORIENTSELAXIS2_ROT_OPT
|--PRO_E_CSYS_ORIENTSELAXIS2_ROT
|
|--PRO_E_CSYS_TYPE_MECH
|--PRO_E_CSYS_FOLLOW_SRF_OPT

```

Feature Elements

The following table describes the elements in the element tree for coordinate system feature.

Element Id	Element Name	Data Type	Valid Values
PRO_E_FEATURE_TYPE	Feature Type	PRO_VALUE_TYPE_INT	PRO_FEAT_CSYS
PRO_E_STD_FEATURE_NAME	Feature Name	PRO_VALUE_TYPE_WSTRING	-
PRO_E_CSYS_ORIGIN_CONSTRS	Origin Constraints	Array	-
PRO_E_CSYS_ORIGIN_CONSTR	Origin Constraint	Compound	-
PRO_E_CSYS_ORIGIN_CONSTR_REF	Origin Reference	PRO_VALUE_TYPE_SELECTION	-
PRO_E_CSYS_OFFSET_TYPE	Origin Offset Type	PRO_VALUE_TYPE_INT	ProCsysOffsetType
PRO_E_CSYS_ONSURF_TYPE	On Surface Type	PRO_VALUE_TYPE_INT	ProCsysOnSurfaceType
PRO_E_CSYS_DIM_CONSTRS	Dimension Constraints	Array	
PRO_E_CSYS_DIM_CONSTR	Dimension Constraint	Compound	
PRO_E_CSYS_DIM_CONSTR_REF	Dimension Constraint Reference	PRO_VALUE_TYPE_SELECTION	
PRO_E_CSYS_DIM_CONSTR_TYPE	Dimension Constraint Type	PRO_VALUE_TYPE_INT	ProCsysDimConstraintType
PRO_E_CSYS_DIM_CONSTR_VAL	Dimension Constraint Value	PRO_VALUE_TYPE_DOUBLE	
PRO_E_CSYS_ORIENTATION_MOVES	Orientation Moves	Array	-
PRO_E_CSYS_ORIENTATION_MOVE		Compound	-
PRO_E_CSYS_ORIENTATION_MOVE_MOVE_TYPE	Move Type	PRO_VALUE_TYPE_INT	ProCsysOrientationMoveMoveOpt

Element Id	Element Name	Data Type	Valid Values
PRO_E_CSYS_ORIENTMOVE_MOVE_VAL	Move Value	PRO_VALUE_TYPE_DOUBLE	-
PRO_E_CSYS_NORMAL_TO_SCREEN	Set Z Normal To Screen	PRO_VALUE_TYPE_INT	ProCsysOrientMovesNrmScrnOpt
PRO_E_CSYS_ORIENT_BY_METHOD	Orient By Method	PRO_VALUE_TYPE_INT	ProCsysOrientByMethod
PRO_E_CSYS_ORIENTSELAXIS1_REFERENCE	First Axis Reference	PRO_VALUE_TYPE_SELECTION	-
PRO_E_CSYS_ORIENTSELAXIS1_REFERENCE_OPT	First Axis Reference Option	PRO_VALUE_TYPE_INT	ProCsysDirCsysRefOpt
PRO_E_CSYS_ORIENTSELAXIS1_OPTION	First Axis Option	PRO_VALUE_TYPE_INT	ProCsysOrientMoveAxisOpt
PRO_E_CSYS_ORIENTSELAXIS1_FLIP	Flip first direction	-	-
PRO_E_CSYS_ORIENTSELAXIS2_REFERENCE	Second Axis Reference	PRO_VALUE_TYPE_SELECTION	-
PRO_E_CSYS_ORIENTSELAXIS2_REFERENCE_OPT	Second Axis Reference Option	PRO_VALUE_TYPE_INT	ProCsysDirCsysRefOpt
PRO_E_CSYS_ORIENTSELAXIS2_OPTION	Second Axis Option	PRO_VALUE_TYPE_INT	ProCsysOrientMoveAxisOpt
PRO_E_CSYS_ORIENTSELAXIS2_FLIP	Flip second direction	-	
PRO_E_CSYS_ORIENTSELAXIS2_ROTATION_OPT	Second Axis Rotation Option	PRO_VALUE_TYPE_INT	ProCsysOrientSelAxisRotOpt
PRO_E_CSYS_ORIENTSELAXIS2_ROTATION	Second Axis Rotation	PRO_VALUE_TYPE_DOUBLE	Axisopt1 != AxisOpt2

Element Id	Element Name	Data Type	Valid Values
PRO_E_CSYS_TYP E_MECH	Coordinate System Type (available in Mechanica mode only)	PRO_VALUE_TYPE_ INT	ProCsysType
PRO_E_CSYS_FOL LOW_SRF_OPT	Follow Surface Option (available in Sheetmetal mode only)	PRO_ELEM_TYPE_ OPTION	ProCsysFollo wSrfOpt

Note: To determine whether a coordinate system is a default coordinate system, query the number of PRO_E_CSYS_ORIGIN_CONSTRS and the number of PRO_E_CSYS_ORIENTMOVES. If both of the numbers are zero, then Csys is the default coordinate system.

The following elements are common for all the cases of the coordinate system feature creation:

Element ID	Value	Comments
PRO_E_FEATURE_TYPE	PRO_FEAT_CSYS	Mandatory
PRO_E_STD_FEATURE_NAME	Feature Name	Optional

Using Three Planes or Two Edges and Axes

Use the following elements if the origin of the coordinate system is defined using three planes or using two edges and axes:

Element ID	Comments
PRO_E_CSYS_ORIGIN_CONSTRS	Mandatory
PRO_E_CSYS_ORIGIN_CONSTR	Mandatory
PRO_E_CSYS_ORIGIN_CONSTR_REF	Mandatory
PRO_E_CSYS_ORIENTSELAXIS1_REF	Optional, using default if not set
PRO_E_CSYS_ORIENTSELAXIS1_REF_OPT	Optional, using default if not set
PRO_E_CSYS_ORIENTSELAXIS1_OPT	Mandatory if PRO_E_CSYS_ORIENTSELAXIS 1_REF is a Csys reference
PRO_E_CSYS_ORIENTSELAXIS1_FLIP	Optional
PRO_E_CSYS_ORIENTSELAXIS2_REF	Optional, using default if not set

Element ID	Comments
PRO_E_CSYS_ORIENTSELAXIS2_REF_OPT	Optional, using default if not set
PRO_E_CSYS_ORIENTSELAXIS1_OPT	Mandatory if PRO_E_CSYS_ORIENTSELAXIS 2_REF is a Csys reference
PRO_E_CSYS_ORIENTSELAXIS2_FLIP	Optional
Others	Not applicable

Using Curve, Edges, or Plane and Axis

Use the following elements if the origin of the coordinate system is defined with a plane and an axis, curve, or edges:

Element ID	Comments
PRO_E_CSYS_ORIGIN_CONSTRS	Mandatory
PRO_E_CSYS_ORIGIN_CONSTR	Mandatory
PRO_E_CSYS_ORIGIN_CONSTR_REF	Mandatory
PRO_E_CSYS_ORIENTSELAXIS1_REF	Optional, using default if not set
PRO_E_CSYS_ORIENTSELAXIS1_REF_OPT	Optional, using default if not set
PRO_E_CSYS_ORIENTSELAXIS1_OPT	Mandatory if PRO_E_CSYS_ORIENTSELAXIS 1_REF is a Csys reference
PRO_E_CSYS_ORIENTSELAXIS1_FLIP	Optional
PRO_E_CSYS_ORIENTSELAXIS2_REF	Mandatory
PRO_E_CSYS_ORIENTSELAXIS2_REF_OPT	Optional, using default if not set
PRO_E_CSYS_ORIENTSELAXIS1_OPT	Mandatory if PRO_E_CSYS_ORIENTSELAXIS 2_REF is a Csys reference
PRO_E_CSYS_ORIENTSELAXIS2_FLIP	Optional
Others	Not applicable

Using a Vertex or a Datum Point

Use the following elements if the origin of the coordinate system is defined using a vertex or a datum point:

Element ID	Comments
PRO_E_CSYS_ORIGIN_CONSTRS	Mandatory
PRO_E_CSYS_ORIGIN_CONSTR	Mandatory

Element ID	Comments
PRO_E_CSYS_ORIGIN_CONSTR_REF	Mandatory
PRO_E_CSYS_ORIENTSELAXIS1_REF	Mandatory
PRO_E_CSYS_ORIENTSELAXIS1_REF_OPT	Optional, using default if not set
PRO_E_CSYS_ORIENTSELAXIS1_OPT	Mandatory if PRO_E_CSYS_ORIENTSELAXIS1_REF is a Csys reference
PRO_E_CSYS_ORIENTSELAXIS1_FLIP	Optional
PRO_E_CSYS_ORIENTSELAXIS2_REF	Mandatory
PRO_E_CSYS_ORIENTSELAXIS2_REF_OPT	Optional, using default if not set
PRO_E_CSYS_ORIENTSELAXIS2_OPT	Mandatory if PRO_E_CSYS_ORIENTSELAXIS2_REF is a Csys reference
PRO_E_CSYS_ORIENTSELAXIS2_FLIP	Optional
Others	Not applicable

Orienting by Selecting References

Use the following elements if
PRO_E_CSYS_ORIENT_BY_METHOD is
PRO_CSYS_ORIENT_BY_SEL_REFS:

Element ID	Comments
PRO_E_CSYS_ORIENTSELAXIS1_REF	Mandatory
PRO_E_CSYS_ORIENTSELAXIS1_REF_OPT	Optional, using default if not set
PRO_E_CSYS_ORIENTSELAXIS1_OPT	Optional, using default if not set
PRO_E_CSYS_ORIENTSELAXIS1_FLIP	Optional, using default if not set
PRO_E_CSYS_ORIENTSELAXIS2_REF	Mandatory
PRO_E_CSYS_ORIENTSELAXIS2_REF_OPT	Optional, using default if not set
PRO_E_CSYS_ORIENTSELAXIS2_OPT	Optional, using default if not set
PRO_E_CSYS_ORIENTSELAXIS2_FLIP	Optional, using default if not set
Others	Not applicable

Orienting by Selecting Csys Axes

Use the following elements if
PRO_E_CSYS_ORIENT_BY_METHOD is
PRO_CSYS_ORIENT_BY_SEL_CSYS_AXES:

Element ID	Comments
PRO_E_CSYS_NORMAL_TO_SCREEN	Optional, valid only if PRO_E_CSYS_ORIENT_BY_METHOD = PRO_CSYS_ORIENT_BY_SEL_CSYS_ AXES. Otherwise, it is ignored.
Others	Not applicable

Using a Csys

Use the following elements if the origin of the coordinate system is
determined using a Csys:

Element ID	Comments
PRO_E_CSYS_ORIGIN_CONSTRS	Mandatory
PRO_E_CSYS_ORIGIN_CONSTR	Mandatory
PRO_E_CSYS_ORIGIN_CONSTR_REF	Mandatory
PRO_E_CSYS_OFFSET_TYPE	Optional, using default PRO_CSYS_OFFSET_CARTESIAN if not set
PRO_E_CSYS_ORIENTMOVES	Mandatory for non PRO_CSYS_OFFSET_CARTESIAN
PRO_E_CSYS_ORIENTMOVE	Mandatory for non PRO_CSYS_OFFSET_CARTESIAN
PRO_E_CSYS_ORIENTMOVE_MOVE_ TYPE	Mandatory for non PRO_CSYS_OFFSET_CARTESIAN

Element ID	Comments
PRO_E_CSYS_ORIENTMOVE_MOVE_VAL	Mandatory for non PRO_CSYS_OFFSET_CARTESIAN
	For PRO_CSYS_OFFSET_CYLINDRICAL, the elements PRO_CSYS_ORIENTMOVE_MOVE_O PT_RAD, PRO_CSYS_ORIENTMOVE_MOVE_O PT_THETA, and PRO_CSYS_ORIENTMOVE_MOVE_O PT_ZI are required.
	For PRO_CSYS_OFFSET_SPHERICAL, the elements PRO_CSYS_ORIENTMOVE_MOVE_O PT_RAD, PRO_CSYS_ORIENTMOVE_MOVE_O PT_PHI, and PRO_CSYS_ORIENTMOVE_MOVE_O PT_THETA are required.
PRO_E_CSYS_ORIENT_BY_METHOD	Mandatory, using default PRO_CSYS_ORIENT_BY_SEL_REFS if not set

The function **ProDtmcsysTransformfileRead()** allocates required steps of the element tree to create CSYS from a transformation file.

The input file name to **ProDtmcsysTransformfileRead()** should have the name of a .trf file, with the extension. The name must be lowercase only. The file should contain a coordinate transform such as:

```

X1      X2      X3      Tx
Y1      Y2      Y3      Ty
Z1      Z2      Z3      Tz

```

where

- X1 Y1 Z1 is the X-axis direction,
- X2 Y2 Z2 is the Y-axis direction,
- X3 Y3 Z3 is not used (the right hand rule determines the Z direction), and

- Tx Ty Tz is the origin of the coordinate system.

Example 7: Creating a Datum Coordinate System

The example shows how to create a General Csys at an offset to the specified Csys. The user is prompted to select a Csys.

```

/*----- Pro/Toolkit Includes -----*/
#include "ProToolkit.h"
#include "ProFeature.h"
#include "ProElemId.h"
#include "ProExtrude.h"
#include "ProModFeat.h"
#include "ProStdSection.h"
#include "ProElement.h"
#include "ProElemPath.h"
#include "ProFeatType.h"
#include "ProFeatForm.h"
#include "ProSelection.h"
#include "ProSection.h"

#include "ProDtmCsys.h"

#define C_LOG(a) ProTKPrintf ("Status for %s is = %d\n", a, status ); \
ERROR_CHECK( a, "UgSktExtrusionCreate.c", status );

#define C_PRINT(a) ProTKPrintf ( "%s\n", a);

static ProFileName message_file;

/*----- Function Prototypes -----*/
ProError ProDemoGeneralCsysCreate();

/*----- External Data -----*/
ProError ProDemoSectCreate();

/*----- Global Data -----*/

/*=====*\
FUNCTION : ProDemoGeneralCsysCreate
PURPOSE  : Demonstrates the creation of the extruded protrusion
           base feature.
\*=====*/
ProError ProDemoGeneralCsysCreate()
{
    ProErrorlist      errors;
    ProMdl             model;
    ProModelitem       model_item;
    ProSelection       model_sel;
    ProFeature         feature;
    ProFeatureCreateOptions opts[1];

```

```

ProAsmcomppath      *p_comp_path = NULL;
ProValue            value;
char                name[PRO_NAME_SIZE];
ProError    status;

ProElement pro_e_feature_tree;
ProElement pro_e_feature_type;
ProElement pro_e_std_feature_name;
ProElement pro_e_csys_origin_constrs;
ProElement pro_e_csys_origin_constr;
ProElement pro_e_csys_origin_constr_ref;
ProElement pro_e_csys_offset_type;
ProElement pro_e_csys_orientmoves;
ProElement pro_e_csys_orientmove;
ProElement pro_e_csys_orientmove_move_type;
ProElement pro_e_csys_orientmove_move_val;
ProElement pro_e_csys_orient_by_method;

ProName wide_string;
ProValueData value_data;
ProSelection * p_select;
int n_select;
ProBoolean is_interactive = PRO_B_TRUE;

ProStringToWstring ( message_file, "utilities.txt" );

/*-----*\
    Populating root element PRO_E_FEATURE_TREE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TREE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TREE, &pro_e_feature_tree );
C_LOG( " ProElementAlloc " );

/*-----*\
    Populating element PRO_E_FEATURE_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TYPE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TYPE, &pro_e_feature_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_FEAT_CSYS; /* 979 */
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_feature_type, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_feature_type );
C_LOG( " ProElemtreeElementAdd" );

```

```

/*-----*\
    Populating element PRO_E_STD_FEATURE_NAME
\*-----*/

C_PRINT( " *** Processing Element PRO_E_STD_FEATURE_NAME *** " );
status = ProElementAlloc ( PRO_E_STD_FEATURE_NAME,
                           &pro_e_std_feature_name );
C_LOG( " ProElementAlloc " );
ProStringToWstring ( wide_string, "MY_CS0" );
value_data.type = PRO_VALUE_TYPE_WSTRING;
value_data.v.w = wide_string;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_std_feature_name, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_std_feature_name );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_CSYS_ORIGIN_CONSTRS
\*-----*/

C_PRINT( " *** Processing Element PRO_E_CSYS_ORIGIN_CONSTRS *** " );
status = ProElementAlloc ( PRO_E_CSYS_ORIGIN_CONSTRS,
                           &pro_e_csys_origin_constrs );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_csys_origin_constrs );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_CSYS_ORIGIN_CONSTR
\*-----*/
C_PRINT( " *** Processing Element PRO_E_CSYS_ORIGIN_CONSTR *** " );
status = ProElementAlloc ( PRO_E_CSYS_ORIGIN_CONSTR,
                           &pro_e_csys_origin_constr );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_csys_origin_constrs, NULL,
                                pro_e_csys_origin_constr );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_CSYS_ORIGIN_CONSTR_REF
\*-----*/
C_PRINT( " *** Processing Element PRO_E_CSYS_ORIGIN_CONSTR_REF *** " );
status = ProMessageDisplay ( message_file, "Select a reference CSYS");
C_LOG( " ProMessageDisplay " );
ProTKPrintf ( "Please select csys_ID_25 type of Modelitem\n");

```

```

status = ProSelect ( "csys", 1, NULL, NULL, NULL, NULL,
                    &p_select, &n_select );
C_LOG( " ProSelect" );
if ( n_select <= 0 ) return -1;
status = ProElementAlloc ( PRO_E_CSYS_ORIGIN_CONSTR_REF,
                          &pro_e_csys_origin_constr_ref );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_csys_origin_constr_ref, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_csys_origin_constr, NULL,
                                pro_e_csys_origin_constr_ref );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_CSYS_OFFSET_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_CSYS_OFFSET_TYPE *** " );
status = ProElementAlloc ( PRO_E_CSYS_OFFSET_TYPE,
                          &pro_e_csys_offset_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_CSYS_OFFSET_CARTESIAN; /* 0 */
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_csys_offset_type, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_csys_offset_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_CSYS_ORIENTMOVES
\*-----*/
C_PRINT( " *** Processing Element PRO_E_CSYS_ORIENTMOVES *** " );
status = ProElementAlloc ( PRO_E_CSYS_ORIENTMOVES,
                          &pro_e_csys_orientmoves );
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_csys_orientmoves );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_CSYS_ORIENTMOVES

```

```

-> PRO_E_CSYS_ORIENTMOVE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_CSYS_ORIENTMOVE *** " );
status = ProElementAlloc ( PRO_E_CSYS_ORIENTMOVE,
                           &pro_e_csys_orientmove );
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_csys_orientmoves, NULL,
                                pro_e_csys_orientmove );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_CSYS_ORIENTMOVES
-> PRO_E_CSYS_ORIENTMOVE
-> PRO_E_CSYS_ORIENTMOVE_MOVE_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_CSYS_ORIENTMOVE_MOVE_TYPE *** "
);
status = ProElementAlloc ( PRO_E_CSYS_ORIENTMOVE_MOVE_TYPE,
                           &pro_e_csys_orientmove_move_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_CSYS_ORIENTMOVE_MOVE_OPT_ROT_X; /* 3 */
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_csys_orientmove_move_type, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_csys_orientmove, NULL,
                                pro_e_csys_orientmove_move_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_CSYS_ORIENTMOVES
-> PRO_E_CSYS_ORIENTMOVE
-> PRO_E_CSYS_ORIENTMOVE_MOVE_VAL
\*-----*/
C_PRINT( " *** Processing Element PRO_E_CSYS_ORIENTMOVE_MOVE_VAL *** "
);
status = ProElementAlloc ( PRO_E_CSYS_ORIENTMOVE_MOVE_VAL,
                           &pro_e_csys_orientmove_move_val );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 0.000000;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_csys_orientmove_move_val, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_csys_orientmove, NULL,

```

```

                                pro_e_csys_orientmove_move_val );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_CSYS_ORIENTMOVES
-> PRO_E_CSYS_ORIENTMOVE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_CSYS_ORIENTMOVE *** " );
status = ProElementAlloc ( PRO_E_CSYS_ORIENTMOVE,
                           &pro_e_csys_orientmove );
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_csys_orientmoves, NULL,
                                pro_e_csys_orientmove );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_CSYS_ORIENTMOVES
-> PRO_E_CSYS_ORIENTMOVE
-> PRO_E_CSYS_ORIENTMOVE_MOVE_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_CSYS_ORIENTMOVE_MOVE_TYPE *** "
);
status = ProElementAlloc ( PRO_E_CSYS_ORIENTMOVE_MOVE_TYPE,
                           &pro_e_csys_orientmove_move_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_CSYS_ORIENTMOVE_MOVE_OPT_ROT_Y; /* 4 */
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_csys_orientmove_move_type, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_csys_orientmove, NULL,
                                pro_e_csys_orientmove_move_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_CSYS_ORIENTMOVES
-> PRO_E_CSYS_ORIENTMOVE
-> PRO_E_CSYS_ORIENTMOVE_MOVE_VAL
\*-----*/
C_PRINT( " *** Processing Element PRO_E_CSYS_ORIENTMOVE_MOVE_VAL *** "
);
status = ProElementAlloc ( PRO_E_CSYS_ORIENTMOVE_MOVE_VAL,
                           &pro_e_csys_orientmove_move_val );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = -90.000000;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );

```



```

status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_csys_orientmove_move_val, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_csys_orientmove, NULL,
                                pro_e_csys_orientmove_move_val );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_CSYS_ORIENTMOVES
-> PRO_E_CSYS_ORIENTMOVE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_CSYS_ORIENTMOVE *** " );
status = ProElementAlloc ( PRO_E_CSYS_ORIENTMOVE,
                           &pro_e_csys_orientmove );
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_csys_orientmoves, NULL,
                                pro_e_csys_orientmove );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_CSYS_ORIENTMOVES
-> PRO_E_CSYS_ORIENTMOVE
-> PRO_E_CSYS_ORIENTMOVE_MOVE_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_CSYS_ORIENTMOVE_MOVE_TYPE *** "
);
status = ProElementAlloc ( PRO_E_CSYS_ORIENTMOVE_MOVE_TYPE,
                           &pro_e_csys_orientmove_move_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_CSYS_ORIENTMOVE_MOVE_OPT_ROT_Z; /* 5 */
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_csys_orientmove_move_type, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_csys_orientmove, NULL,
                                pro_e_csys_orientmove_move_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_CSYS_ORIENTMOVES
-> PRO_E_CSYS_ORIENTMOVE
-> PRO_E_CSYS_ORIENTMOVE_MOVE_VAL
\*-----*/
C_PRINT( " *** Processing Element PRO_E_CSYS_ORIENTMOVE_MOVE_VAL *** "
);
status = ProElementAlloc ( PRO_E_CSYS_ORIENTMOVE_MOVE_VAL,
                           &pro_e_csys_orientmove_move_val );

```

```

C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 0.000000;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_csys_orientmove_move_val, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_csys_orientmove, NULL,
                                pro_e_csys_orientmove_move_val );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_CSYS_ORIENTMOVES
-> PRO_E_CSYS_ORIENTMOVE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_CSYS_ORIENTMOVE *** " );
status = ProElementAlloc ( PRO_E_CSYS_ORIENTMOVE, &pro_e_csys_orientmove
);
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_csys_orientmoves, NULL,
                                pro_e_csys_orientmove );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_CSYS_ORIENTMOVES
-> PRO_E_CSYS_ORIENTMOVE
-> PRO_E_CSYS_ORIENTMOVE_MOVE_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_CSYS_ORIENTMOVE_MOVE_TYPE *** "
);
status = ProElementAlloc ( PRO_E_CSYS_ORIENTMOVE_MOVE_TYPE,
                            &pro_e_csys_orientmove_move_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_CSYS_ORIENTMOVE_MOVE_OPT_TRAN_X; /* 0 */
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_csys_orientmove_move_type, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_csys_orientmove, NULL,
                                pro_e_csys_orientmove_move_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_CSYS_ORIENTMOVES
-> PRO_E_CSYS_ORIENTMOVE
-> PRO_E_CSYS_ORIENTMOVE_MOVE_VAL

```

```

/*-----*/
C_PRINT( " *** Processing Element PRO_E_CSYS_ORIENTMOVE_MOVE_VAL *** "
);
status = ProElementAlloc ( PRO_E_CSYS_ORIENTMOVE_MOVE_VAL,
                           &pro_e_csys_orientmove_move_val );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 100.000000;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_csys_orientmove_move_val, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_csys_orientmove, NULL,
                                pro_e_csys_orientmove_move_val );
C_LOG( " ProElemtreeElementAdd " );

/*-----\
    Populating element PRO_E_CSYS_ORIENTMOVES
    -> PRO_E_CSYS_ORIENTMOVE
    -> PRO_E_CSYS_ORIENTMOVE_MOVE_VAL
    \-----*/
C_PRINT( " *** Processing Element PRO_E_CSYS_ORIENTMOVE *** " );
status = ProElementAlloc ( PRO_E_CSYS_ORIENTMOVE, &pro_e_csys_orientmove
);
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_csys_orientmoves, NULL,
                                pro_e_csys_orientmove );
C_LOG( " ProElemtreeElementAdd " );

/*-----\
    Populating element PRO_E_CSYS_ORIENTMOVES
    -> PRO_E_CSYS_ORIENTMOVE
    -> PRO_E_CSYS_ORIENTMOVE_MOVE_TYPE
    \-----*/
C_PRINT( " *** Processing Element PRO_E_CSYS_ORIENTMOVE_MOVE_TYPE *** "
);
status = ProElementAlloc ( PRO_E_CSYS_ORIENTMOVE_MOVE_TYPE,
                           &pro_e_csys_orientmove_move_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_CSYS_ORIENTMOVE_MOVE_OPT_TRAN_Y; /* 1 */
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_csys_orientmove_move_type, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_csys_orientmove, NULL,
                                pro_e_csys_orientmove_move_type );

```

```

C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_CSYS_ORIENTMOVES
    -> PRO_E_CSYS_ORIENTMOVE
    -> PRO_E_CSYS_ORIENTMOVE_MOVE_VAL
\*-----*/
C_PRINT( " *** Processing Element PRO_E_CSYS_ORIENTMOVE_MOVE_VAL *** "
);
status = ProElementAlloc ( PRO_E_CSYS_ORIENTMOVE_MOVE_VAL,
                           &pro_e_csys_orientmove_move_val );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 200.000000;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_csys_orientmove_move_val, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_csys_orientmove, NULL,
                                pro_e_csys_orientmove_move_val );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_CSYS_ORIENTMOVES
    -> PRO_E_CSYS_ORIENTMOVE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_CSYS_ORIENTMOVE *** " );
status = ProElementAlloc ( PRO_E_CSYS_ORIENTMOVE, &pro_e_csys_orientmove
);
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_csys_orientmoves, NULL,
                                pro_e_csys_orientmove );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_CSYS_ORIENTMOVES
    -> PRO_E_CSYS_ORIENTMOVE
    -> PRO_E_CSYS_ORIENTMOVE_MOVE_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_CSYS_ORIENTMOVE_MOVE_TYPE *** "
);
status = ProElementAlloc ( PRO_E_CSYS_ORIENTMOVE_MOVE_TYPE,
                           &pro_e_csys_orientmove_move_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_CSYS_ORIENTMOVE_MOVE_OPT_TRAN_Z; /* 2 */
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );

```

```

C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_csys_orientmove_move_type, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_csys_orientmove, NULL,
                                pro_e_csys_orientmove_move_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_CSYS_ORIENTMOVES
-> PRO_E_CSYS_ORIENTMOVE
-> PRO_E_CSYS_ORIENTMOVE_MOVE_VAL
\*-----*/
C_PRINT( " *** Processing Element PRO_E_CSYS_ORIENTMOVE_MOVE_VAL *** "
);
status = ProElementAlloc ( PRO_E_CSYS_ORIENTMOVE_MOVE_VAL,
                          &pro_e_csys_orientmove_move_val );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 300.000000;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_csys_orientmove_move_val, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_csys_orientmove, NULL,
                                pro_e_csys_orientmove_move_val );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_CSYS_ORIENT_BY_METHOD
\*-----*/
status = ProElementAlloc ( PRO_E_CSYS_ORIENT_BY_METHOD,
                          &pro_e_csys_orient_by_method );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_CSYS_ORIENT_BY_SEL_CSYS_AXES;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_csys_orient_by_method, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_csys_orient_by_method );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Creating the feature in the current model.
\*-----*/
status = ProMdlCurrentGet (&model);

```

```

if ( status != PRO_TK_NO_ERROR ) return ( status );
status = ProMdlToModelitem( model, &model_item );
status = ProSelectionAlloc (p_comp_path, &model_item,
                           &model_sel);

opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMS;
status = ProFeatureCreate (model_sel, pro_e_feature_tree, opts, 1,
                           &feature, &errors);
C_LOG (" ProFeatureCreate");

status = ProElementFree (&pro_e_feature_tree );
C_LOG (" ProElementFree");

return ( status );
}

```

Element Trees: Datum Curves

This chapter describes how to create, redefine, and access data for datum curve features using Pro/TOOLKIT. The chapter 'Element Trees: Datum Features' provides necessary background for creating features; we recommend you read that material first.

Topic	Page
Datum Curve Features	32 - 2
Datum Curve Types	32 - 2
Other Datum Curve Types	32 - 7

Datum Curve Features

The element trees for datum curve features supported in Pro/TOOLKIT are documented in the header file *ProDtmCrv.h*. Each datum feature type has a unique element tree containing the parameters and references necessary to create that type of feature.

Not all datum curve types are currently supported in Pro/TOOLKIT. Some curve feature types are yet to be converted into element tree form. Other curve types have element trees with data that is not yet accessible through Pro/TOOLKIT.

Common Elements

All datum curve features support the following common elements.

Table 1: Common Elements for Datum Curve Features

Element ID	Value
PRO_E_FEATURE_TYPE	PRO_FEAT_CURVE
PRO_E_CURVE_TYPE	As listed in ProCurveType. This element identifies the subtree to be used.
PRO_E_STD_FEATURE_NAME	Wstring (feature name)

Datum Curve Types

Pro/TOOLKIT considers the following curve types for providing element tree access:

- Sketched Datum Curves
- Trim Datum Curves
- Intersect Datum Curves
- Wrap Datum Curves
- Offset Datum Curves
- Tangent Offset Datum Curves

Sketched Datum Curves

Pro/TOOLKIT provides complete element tree access to the sketched datum curves. The sketched datum curves are sketched features, and therefore must be created using the techniques described in the chapter 'Element Trees: Sketched Features'.

Table 2: Elements for Sketched Datum Curve Features

Element ID	Value
PRO_E_CURVE_TYPE	PRO_CURVE_TYPE_SKETCHED
PRO_E_STD_SECTION	Section element tree
PRO_E_DTMCRV_DISPLAY_HATCH	Integer (PRO_B_TRUE, PRO_B_FALSE)
PRO_E_DTMCRV_HATCH_DENSITY	Double (if DISPLAY_HATCH = PRO_B_TRUE)

Trim Datum Curves

Pro/TOOLKIT provides complete element tree access to trim datum curves (previously called Split datum curve).

Table 3: Elements for Trim Datum Curve Features

Element ID	Value
PRO_E_CURVE_TYPE	PRO_CURVE_TYPE_SPLIT
PRO_E_STD_CRV_SPLIT_CURVE	The PRO_CURVE geometric item selected for splitting.
PRO_E_STD_CRV_DIVIDER	The geometric item used to divide the curve.
PRO_E_STD_CRV_SPLIT_SIDE	One of the ProSplitSides enumerations

Intersect Datum Curves

Pro/TOOLKIT provides complete element tree access to intersect datum curves. In the user interface, the intersect curve type results in one of the following curve types depending upon the references selected:

- A curve based on the intersection of two surfaces
- A curve based on the projections of two sections

The feature element tree for Intersect curve type contains two independent sets of elements to support both these feature types. The curve type determines which elements are required. As the two projections curve type contains two independent PRO_E_STD_SECTION elements, it must be created using the techniques described in the chapter 'Element Trees: Sketched Features'.

Table 4: Elements for Intersection of Surfaces Datum Curve Features

Element ID	Value
PRO_E_CURVE_TYPE	PRO_CURVE_TYPE_INTSRF
PRO_E_CRV_IP_REF_TYPE	PRO_CURVE_TYPE_INTSRF
PRO_E_CRV_IP_COMP_REF1	Compound
PRO_E_CRV_IP_REF_SEL1_TYPE	PRO_CURVE_TYPE_WHOLE for the <i>whole</i> surface selection; PRO_CURVE_TYPE_MULTIPLE_SEL for <i>multiple</i> independent surface selections.
PRO_E_CRV_IP_REF1	Based on the value of PRO_E_CRV_IP_SEL1_TYPE. • If the value is <i>whole</i>, specifies a single selection of a datum plane, quilt, or solid geometry entity. • If the value is <i>multiple</i>, specifies a multi-valued element containing any number of surface items.
PRO_E_CRV_IP_COMP_REF2	Compound
PRO_E_CRV_IP_REF_SEL2_TYPE	PRO_CURVE_TYPE_WHOLE for the <i>whole</i> surface selection; PRO_CURVE_TYPE_MULTIPLE_SEL for multiple independent surface selections.

Table 4: Elements for Intersection of Surfaces Datum Curve Features

Element ID	Value
PRO_E_CRV_IP_REF2	Based on the value of PRO_E_CRV_IP_SEL1_TYPE. - If the value is <i>whole</i>, specifies a single selection of a datum plane, quilt, or solid geometry entity. - If the value is multiple, specifies a multi valued element containing any number of surface items.

Table 5: Elements for Two Projections Datum Curve Features

Element ID	Value
PRO_E_CURVE_TYPE	PRO_CURVE_TYPE_TWO_PROJ
PRO_E_CRV_IP_REF_TYPE	PRO_CURVE_TYPE_TWO_PROJ
PRO_E_CRV_IP_COMP_SEC1	Compound
PRO_E_STD_SECTION	Section element tree
PRO_E_CRV_IP_COMP_SEC2	Compound
PRO_E_STD_SECTION	Section element tree

Wrap Datum Curves

Pro/TOOLKIT provides complete element tree access to wrap datum curves (also called Formed datum curves). Because the curve type contains a PRO_E_STD_SECTION element, you must create it using the techniques described in the chapter ‘Element Trees: Sketched Features’.

Table 6: Elements for Wrap Datum Curve Features

Element ID	Value
PRO_E_CURVE_TYPE	PRO_CURVE_TYPE_WRAP
PRO_E_CRV_WRAP_SRF_TYPE	One of ProWrapSrfType

Table 6: Elements for Wrap Datum Curve Features

Element ID	Value
PRO_E_CRV_WRAP_SRF	Selection containing the wrap surface (surface, quilt or solid geometry)
PRO_E_STD_SECTION	Section
PRO_E_CRV_WRAP_FLIP	One of ProWrapFlip
PRO_E_CRV_WRAP_COORD_SYS	ID of the section coordinate system

Offset Datum Curves

Pro/TOOLKIT provides partial element tree access to offset datum curves. In the user interface, the Offset curve type results in one of the following curve types depending upon the selected references:

- A curve offset normal to a surface
- A curve offset within a quilt

The feature element tree for Offset curve type contains two independent sets of elements to support both of these feature types. The curve type determines which elements are required:

- PRO_CURVE_TYPE_OFFSET is offset normal to a surface
- PRO_CURVE_TYPE_OFFSET_IN_QUILT is offset normal to a quilt

The offset datum curve type is not currently accessible for creation, redefinition, or inspection as the offset in quilt curve type contains elements requiring run-time data, that is not currently accessible to Pro/TOOLKIT. The elements used for the offset in quilt type should be ignored when using the offset normal to surface element tree.

Table 7: Elements for Offset (Normal) Datum Curve Features

Element ID	Value
PRO_E_CURVE_TYPE	PRO_CURVE_TYPE_OFFSET
PRO_E_CRV_OFFS_FEAT_TYPE	PRO_OFFSET_FROM_SURFACE
PRO_E_CRV_OFFS_SRF_REF	Selection of surface or quilt
PRO_E_CRV_OFFS_DIR_FLIP	One of ProOffsetDirFlip
PRO_E_DATUM_CURVE_OFFSET_VAL	The offset value or scale if a graph is used for offset

Table 7: Elements for Offset (Normal) Datum Curve Features

Element ID	Value
PRO_E_CRV_OFFS_CRV_REF	Selection of datum curve to be offset
PRO_E_CRV_OFFS_GRAPH_REF	Selection of graph used for offset calculation (optional)
PRO_E_CRV_OFFS_ST_END	One of ProOffsetStEnd

Tangent Offset Datum Curves

The curve type Tangent Offset is obsolete in Pro/ENGINEER Wildfire. As the existing models created in earlier releases may contain this curve type, Pro/TOOLKIT provides read and redefine access only for these curves.

Table 8: Elements for Tangent Offset Datum Curve Features

Element ID	Value
PRO_E_CURVE_TYPE	PRO_CURVE_TYPE_TANGENT_OFFSET
PRO_E_CRV_TANG_OFFSET_CURVE	Selection of curve to be offset
PRO_E_CRV_TANG_OFFSET_SURF	Selection of surface in which to create the offset
PRO_E_CRV_TANG_OFFSET_DIR	One of ProOffsetDirection
PRO_E_CRV_TANG_OFFSET_DIST	Offset value

Other Datum Curve Types

The following curve types contain run-time data in their element trees that is not currently accessible by Pro/TOOLKIT. Currently, Pro/TOOLKIT does not provide element tree access to the following curve types:

- Copy
- Project
- Boundary Offset

Some other curve types, including Thru Points, From File, Use Xsec, and From Equation do not currently use element trees in Pro/ENGINEER, and are therefore not accessible via Pro/TOOLKIT.

Element Trees: Edit Menu Features

This chapter describes how to construct and access the element tree for some Edit Menu features in Pro/TOOLKIT. It also shows how to redefine, create and access the properties of these features.

Topic	Page
Mirror Feature	33 - 2
Move Feature	33 - 4
Fill Feature	33 - 8
Intersect Feature	33 - 11
Merge Feature	33 - 11
Pattern Feature	33 - 14
Wrap Feature	33 - 14
Trim Feature	33 - 14
Offset Feature	33 - 21
Thicken Feature	33 - 21
Solidify Feature	33 - 24
Remove Feature	33 - 27

Mirror Feature

The Mirror feature creates a copy of the selected geometry by mirroring about a mirror plane. Pro/TOOLKIT supports Mirror features where the initial selection of items was of geometry (curves and /or surfaces). Pro/TOOLKIT does not support the element tree for Mirror features where the initial selection was of one or more features. Mirror features made from other features will have subfeatures listed under the Mirror entry in the Pro/ENGINEER model tree, while Mirror features made from geometry will not include any sub-features.

Mirroring

Figure 33-1: Entity before Mirroring

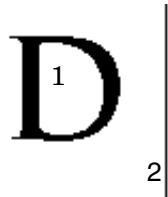
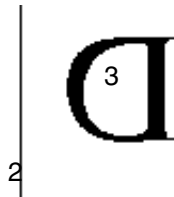


Figure 33-2: Entity after Mirroring



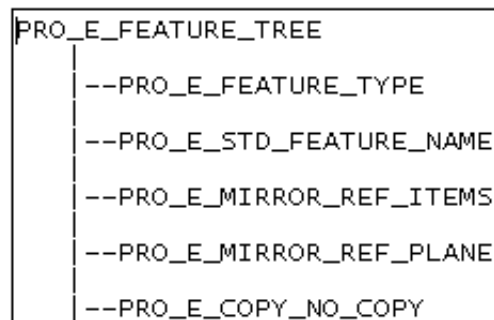
where,

- 1 —Specifies original entity such as plane, surfaces, axes or parts.
- 2 —Specifies mirror line (mirror plane).
- 3 —Specifies the copy of the original entity.

The Feature Element Tree for Mirror feature in Pro/ENGINEER

The element tree for a Mirror feature is documented in the header file *ProMirror.h*, and has a simple structure. The following figure demonstrates the feature element tree structure:

Figure 33-3: Feature Element Tree for Mirror Feature



The Mirror element tree contains no non-standard element types. The following list details special information about some of the elements in this tree:

- **PRO_E_FEATURE_TYPE**—Must be **PRO_FEAT_SRF_MDL**.
- **PRO_E_STD_FEATURE_NAME**—Specifies the feature name.
- **PRO_E_MIRROR_REF_ITEMS**—A multivalued element that includes the mirror item reference. It must be references of the following types: **PRO_CURVE**, **PRO_COMP_CRV**, **PRO_AXIS**, **PRO_QUILT**, **PRO_PART**.
- **PRO_E_MIRROR_REF_PLANE**—Specifies the mirror plane and is a mandatory element and must be **PRO_DATUM_PLANE** or **PRO_SURFACE** (only planar surfaces).
- **PRO_E_COPY_NO_COPY**—Specifies the option to create a copy or not and is a mandatory element. It can have either of the following values:
 - **PRO_MIRROR_KEEP_ORIGINAL**
 - **PRO_MIRROR_HIDE_ORIGINAL**

Creating a Mirror Feature

Function Introduced

- **ProFeatureCreate()**

Use the function **ProFeatureCreate()** to create a Mirror Feature based on element tree input. For more information about **ProFeatureCreate()**, refer to the section “Overview of Feature Creation” of chapter “Element Trees: Principles of Feature Creation”.

Redefining a Mirror Feature

Function Introduced

- **ProFeatureRedefine()**

Use the function **ProFeatureRedefine()** to redefine a Mirror Feature based on the changes made in the element tree. For more information about **ProFeatureRedefine()**, refer to the section “Feature Redefine” of chapter “Element Trees: Principles of Feature Creation”.

Accessing a Mirror Feature

Function Introduced

- **ProFeatureElemtreeExtract()**

Use the function **ProFeatureElemtreeExtract()** to create a feature element tree that describes the contents of a Mirror Feature and to retrieve the element tree description of a Mirror Feature. For more information about **ProFeatureElemtreeExtract()**, refer to the section “Feature Inquiry” of chapter “Element Trees: Principles of Feature Creation”.

Move Feature

The Move feature can be used to create and move a copy of an existing surface or curve rather than moving the original. Using the Move feature saves time because it enables you to create simple patterns with surfaces and curves.

The Move feature is available for both part and assembly modes. You can apply this feature in different modes:

- Translate mode—Translate surfaces, datum curves, and axes in a direction perpendicular to a reference plane. You can also translate along a linear edge, axis, perpendicular to plane, two points, or coordinate system.
- Rotate mode—Rotate surfaces, datum curves, and axes about an existing axis, linear edge, perpendicular to plane, two points, or coordinate system.

To move a surface or curve relative to its original position, define a move reference. In the Translate mode, the move reference is typically a plane or edge that is perpendicular to the direction in which you want to translate the moved feature. In Rotate mode, the move reference is typically an axis or edge about which you want to rotate the moved feature. In this mode, the moved object moves about the direction reference.

Following types of move references exists in Translate mode:

- Linear curve
- Linear edge
- Axis
- Axis of a coordinate system
- Plane
- Two points

Note: The direction reference is always perpendicular to the direction in which you want to move.

Following types of move references exists in Rotate mode:

- Linear curve
- Edge
- Axis
- Axis of a coordinate system
- Two points

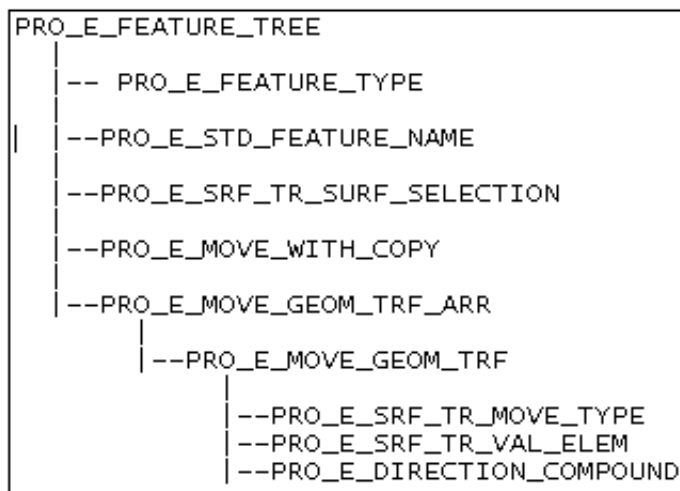
Note: You can flip the direction of the move in the Rotate mode.

Pro/TOOLKIT supports Move features where the initial selection of items was of geometry (curves and /or surfaces). Pro/TOOLKIT does not support the element tree for Move features where the initial selection was of one or more features. Move features made from other features will have subfeatures listed under the Move entry in the Pro/ENGINEER model tree, while Move features made from geometry will not include any sub-features.

The Feature Element Tree for Move feature in Pro/ENGINEER

The element tree for a Move feature is documented in the header file ProMove.h, and has a simple structure. The following figure demonstrates the feature element tree structure:

Figure 33-4: Feature Element Tree for Move Feature



The move element tree contains no non-standard element types. The following list details special information about some of the elements in this tree:

- PRO_E_FEATURE_TYPE—Must be PRO_FEAT_SRF_MDL.
- PRO_E_STD_FEATURE_NAME—Specifies the feature name.
- PRO_E_SRF_TR_SURF_SELECTION—Specifies whether to move or copy the selected geometry. Curves, composite curves, axes and quilts are eligible to for this element.

- **PRO_E_MOVE_NO_COPY**—Specifies an option to control copy of the original geometry. Provides the ability to transform geometry with or without a copy.
- **PRO_E_MOVE_WITH_COPY**—Specifies whether to hide or display the original geometry after copy.
- **PRO_E_MOVE_GEOM_TRF_ARR** - Contains an array of movements applied to the selected entities. This can be a combination of translational and rotational transformations.
- **PRO_E_DIRECTION_COMPOUND**—Specifies the direction reference for the translation or rotational movement. This compound element is a standard Pro/ENGINEER element subtree and is described in *ProDirection.h*.

Creating a Move Feature

Function Introduced

- **ProFeatureCreate()**

Use the function **ProFeatureCreate()** to create a Move Feature based on element tree input. For more information about **ProFeatureCreate()**, refer to the section “Overview of Feature Creation” of chapter Element Trees: Principles of Feature Creation.

Redefining a Move Feature

Function Introduced

- **ProFeatureRedefine()**

Use the function **ProFeatureRedefine()** to redefine a Move Feature based on the changes made in the element tree. For more information about **ProFeatureRedefine()**, refer to the section “Feature Redefine” of chapter “Element Trees: Principles of Feature Creation”.

Accessing a Move Feature

Function Introduced

- **ProFeatureElemtreeExtract()**

Use the function **ProFeatureElemtreeExtract()** to create a feature element tree that describes the contents of a Move Feature and to retrieve the element tree description of a Move Feature. For more information about **ProFeatureElemtreeExtract()**, refer to the section “Feature Inquiry” of chapter “Element Trees: Principles of Feature Creation”.

Fill Feature

Pro/TOOLKIT enables you to create and redefine flat surface features called fill features. A fill feature is simply a flat surface feature that is defined by its boundaries and is used to thicken surfaces.

Note: All fill features must consist of a flat, closed-loop sketched feature.

The Feature Element Tree for Fill feature in Pro/ENGINEER

The element tree for a Fill feature is documented in the header file *ProFlatSrf.h*, and has a simple structure.

The following figure demonstrates the feature element tree structure:

Figure 33-5: Feature Element Tree for Fill Feature

```
PRO_E_FEATURE_TREE
|
|--PRO_E_FEATURE_TYPE
|
|--PRO_E_FEATURE_FORM
|
|--PRO_E_STD_SECTION
|
|--PRO_E_STD_FEATURE_NAME
|
|--PRO_E_IS_UNATTACHED_WALL
|
|--PRO_E_STD_DIRECTION
|
|--PRO_E_STD_SMT_THICKNESS
|
|--PRO_E_STD_SMT_SWAP_DRV_SIDE
```

The fill element tree contains no non-standard element types. The following list details special information about some of the elements in this tree:

- PRO_E_FEATURE_TYPE—It should be PRO_FEAT_DATUM_SURF.
- PRO_E_FEATURE_FORM—Specifies feature form and should be of PRO_FLAT type only.
- PRO_E_STD_SECTION—Specifies a sketched section. Refer to the section Creating Features Containing Sections for details on how to create features that contain sketched sections.

The element tree for the Fill feature also supports access to Flat Sheetmetal Wall features. Refer to the chapter Production Applications: Sheetmetal for element details.

Creating a Fill Feature

Function Introduced

- **ProFeatureCreate()**

Use the function **ProFeatureCreate()** to create a Fill Feature based on element tree input. For more information about **ProFeatureCreate()**, refer to the section “Overview of Feature Creation” of chapter “Element Trees: Principles of Feature Creation”.

Redefining a Fill Feature

Function Introduced

- **ProFeatureRedefine()**

Use the function **ProFeatureRedefine()** to redefine a Fill Feature based on the changes made in the element tree. For more information about **ProFeatureRedefine()**, refer to the section “Feature Redefine” of chapter “Element Trees: Principles of Feature Creation”.

Accessing a Fill Feature

Function Introduced

- **ProFeatureElemtreeExtract()**

Use the function **ProFeatureElemtreeExtract()** to create a feature element tree that describes the contents of a Fill Feature and to retrieve the element tree description of a Fill Feature. For more information about **ProFeatureElemtreeExtract()**, refer to the section “Feature Inquiry” of chapter “Element Trees: Principles of Feature Creation”.

Intersect Feature

Refer to the section Intersect Datum Curves in the chapter Element Trees: Datum Curves for details about this feature.

Merge Feature

The Merge Feature is created by merging two or more selected quilts. In case of more than two quilts, every input quilt should have at least one of its edges adjacent to the edge of any other input quilt, and the surfaces must not overlap.

You can merge a number of input quilts by joining two adjacent quilts one after another, that is, by aligning the edges of one quilt to the edges of the other. The first quilt selected for the merge operation becomes the primary reference quilt. The second adjacent quilt is joined to the primary quilt, forming the main body or a newly formed primary quilt. The third quilt is, then, joined to the main body. This process continues until all the input quilts are joined together.

Note:

- For a successful merge, the selected quilts must be ordered based upon their adjacency.
- The merge operation fails in case of intersecting quilts, and quilts that are not adjacent to any other input quilt. In either case, remove the problematic quilt to complete the merge.

You can also select the input quilts for the merge operation using region selection. In case of region selection, only box selection is available and the quilts to be selected must be completely inside the region. The selected quilts are ordered based on the feature number of the quilt's parent feature.

You can merge only two quilts by intersecting. You can specify which portion of the quilt to include in the merge feature by selecting the sides for each of the quilts.

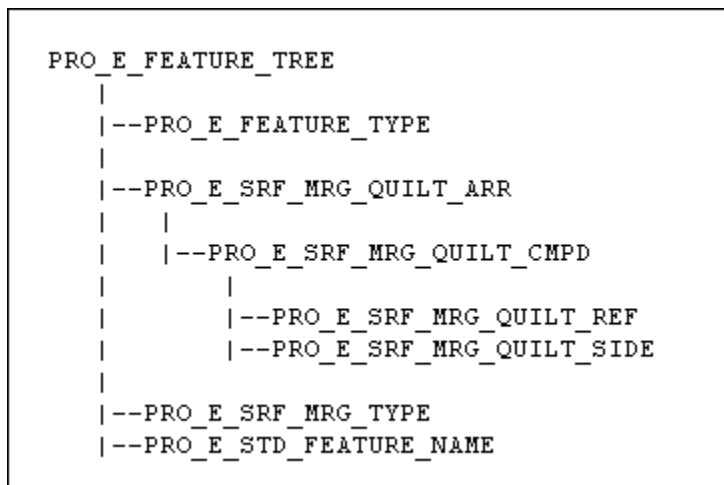
A merged quilt consists of two or more original quilts that provide the geometry, and a merge feature that contains the information for the surface joining. The input quilts are retained, even if you delete the Merge feature.

Note: In Assembly mode, you can merge only assembly-level quilts. If you want to create component-level merge features, you must first activate the component, and then merge the quilts in the component. Surface merge is available only for surfaces that belong to the same component.

Feature Element Tree for Merge feature in Pro/ENGINEER

The element tree for the Merge feature is documented in the header file *ProMerge.h*, and has a simple structure. The following figure demonstrates the feature element tree structure:

Figure 33-6: Feature Element Tree for Merge Feature



The following list details special information about the elements in this tree:

- **PRO_E_FEATURE_TYPE**—Specifies the feature type; must be **PRO_FEAT_DATUM_QUILT**.
- **PRO_E_SRF_MRG_QUILT_ARR**—Specifies an array of the following compound element. The array must have at least two elements to create the Merge feature.
 - **PRO_E_SRF_MRG_QUILT_CMPD**—Specifies the set of references and sides to be used for the merge operation. The set consists of the following two elements:

- PRO_E_SRF_MRG_QUILT_REF—Specifies the quilt of the type PRO_QUILT selected for the merge operation. This is a mandatory element.
- PRO_E_SRF_MRG_QUILT_SIDE—Specifies the side of the selected quilt that should be included in the merge feature in case of intersecting quilts. This element is ignored when the quilt array PRO_E_SRF_MRG_QUILT_ARR contains more than two elements.
- PRO_E_SRF_MRG_TYPE—Specifies the merge type and can have following merge type options:
 - PRO_SRF_MRG_JOIN—For joining quilts.
 - PRO_SRF_MRG_INTSCT—For merging two quilts at the intersection. This is the default option.

The merge type must be PRO_SRF_MRG_JOIN when the quilt array PRO_E_SRF_MRG_QUILT_ARR contains more than two elements.

- PRO_E_STD_FEATURE_NAME—Specifies the merge feature name.

Creating a Merge Feature

Function Introduced

- **ProFeatureCreate()**

Use the function **ProFeatureCreate()** to create a Merge Feature based on element tree input. For more information about **ProFeatureCreate()**, refer to the section “Overview of Feature Creation” of chapter “Element Trees: Principles of Feature Creation”.

Redefining a Merge Feature

Function Introduced

- **ProFeatureRedefine()**

Use the function **ProFeatureRedefine()** to redefine a Merge Feature based on the changes made in the element tree. For more information about **ProFeatureRedefine()**, refer to the section “Feature Redefine” of chapter “Element Trees: Principles of Feature Creation”.

Accessing a Merge Feature

Function Introduced

- **ProFeatureElemtreeExtract()**

Use the function **ProFeatureElemtreeExtract()** to create a feature element tree that describes the contents of a Merge Feature and to retrieve the element tree description of a Merge Feature. For more information about **ProFeatureElemtreeExtract()**, refer to the section “Feature Inquiry” of chapter “Element Trees: Principles of Feature Creation”.

Pattern Feature

Refer to the chapter Element Trees: Patterns for details about this feature.

Wrap Feature

Refer to the section Wrap Datum Curves in chapter Element Trees: Datum Curves for details about this feature.

Trim Feature

Trim feature is applied to remove portions of an existing surface feature. A trim feature is similar to a solid cut. Following are the existing types of feature creation for the trimming of a surface or a quilt.

- Use Quilts—Cuts a piece from a surface using an intersecting quilt. Pro/ENGINEER consumes the quilt that is used to trim a surface and allows you to keep either or both sides of the trimmed surface. You can trim quilts in the following ways:
 - By adding a cut or slot as you do to remove material from solid features
 - By trimming the quilt at its intersection with another quilt or to its own silhouette edge as it appears in a certain view
 - By filleting corners of the quilt
 - By trimming along a datum curve lying on the quilt

- Use Curves—Trims a surface using selected curves and edges.

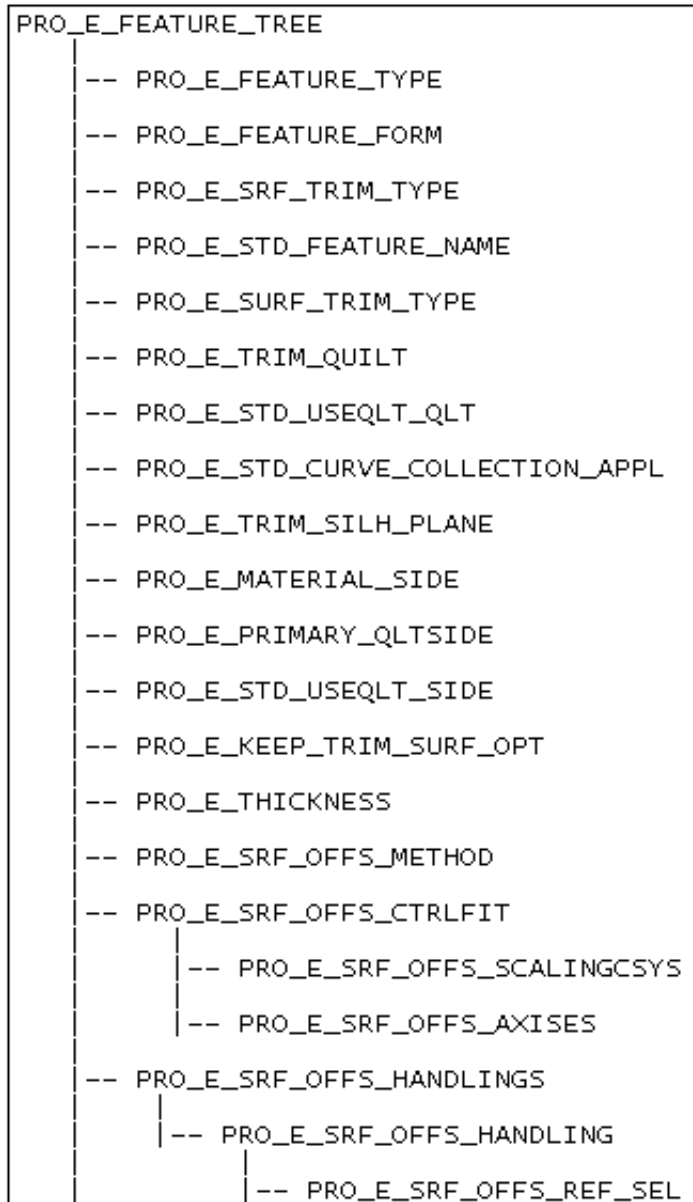
The rules for defining a surface trim using a datum curve are as follows:

- You can use a continuous chain of datum curves, inner surface edges, or solid model edges to trim a quilt.
- Datum curves used for trimming must lie on the quilt to be trimmed and should not extend beyond the boundaries of this quilt.
- If the curve does not extend to the boundaries of the quilt, the system calculates the shortest distance to the quilt boundary and continues the trim in this direction.
- Silhouette—Trims a surface at its silhouette edge from a specified direction.

The Feature Element Tree for Trim feature in Pro/ENGINEER

The element tree for a Trim feature is documented in the header file *ProTrim.h*, and has a simple structure. The following figure demonstrates the feature element tree structure:

Figure 33-7: Feature Element Tree for Trim Feature



The trim element tree contains no non-standard element types. The following list details special information about some of the elements in this tree:

- **PRO_E_FEATURE_TYPE**—It should be **PRO_FEAT_CUT**.

- **PRO_E_FEATURE_FORM**—Specifies feature form and is a mandatory element applicable only for use quilt and thin types. It has **PRO_USE_SURFS** and **PRO_NOTYPE** as their valid values.
- **PRO_E_SRF_TRIM_TYPE**—Specifies trim type identity and is a non-redefinable mandatory element. The trim type determines the required structure for the remaining elements in the element tree. Each type requires different mandatory elements. The different trim types are as follows:
 - **PRO_SURF_TRIM_USE_CRV**—use curves to trim the quilt.
 - **PRO_SURF_TRIM_USE_QLT**—use another quilt to trim the given quilt.
 - **PRO_SURF_TRIM_THIN**
 - **PRO_SURF_TRIM_SILH**—trim quilt using silhouette edges.
- **PRO_E_TRIM_QUILT**—Specifies trimmed quilt and should be of type **PRO_SURFACE** or **PRO_QUILT** only.
- **PRO_E_STD_USEQLT_QLT**—Specifies trimming quilts and trimming planes. It should have **PRO_SURFACE**, **PRO_QUILT** and **PRO_DATUM_PLANE** values only. It is applicable for use quilt and thin types only.
- **PRO_E_STD_CURVE_COLLECTION_APPL**—Specifies the trimming curves and is applicable for use curve only.
- **PRO_E_TRIM_SILH_PLANE**—Specifies silhouette plane and can be of type **PRO_SURFACE** or **PRO_DATUM_PLANE** only.
- **PRO_E_MATERIAL_SIDE**—Specifies material side options and has the following values:
 - **PRO_TRIM_MATL_SIDE_ONE**
 - **PRO_TRIM_MATL_SIDE_TWO**
 - **PRO_TRIM_MATL_BOTH_SIDES**
- **PRO_E_PRIMARY_QLTSIDE**—Specifies the primary quilt side options and they are listed as follows:
 - **PRO_TRIM_PRIM_QLT_SIDE_ONE**
 - **PRO_TRIM_PRIM_QLT_SIDE_TWO**
- **PRO_E_STD_USEQLT_SIDE**—Specifies thickness direction options for thin trims and can only be as follows:
 - **PRO_TRIM_STD_QUILT_SIDE_ONE**

- PRO_TRIM_STD_QUILT_SIDE_TWO
- PRO_TRIM_STD_QUILT_BOTH_SIDES
- PRO_E_KEEP_TRIM_SURF_OPT—Specifies to keep trimming surface option and is valid only when the trimming quilt is a surface or quilt. It has the following values:
 - PRO_KEEP_TRIM_SURF_OPT_NO
 - PRO_KEEP_TRIM_SURF_OPT_YES
- PRO_E_THICKNESS—Specifies thickness for thin trims.
- PRO_E_SRF_OFFS_METHOD—Specifies the types of offset for thin trims and are as follows:
 - PRO_TRIM_SRF_OFFS_METH_NORMTOSURF—The surface or quilt is thickened in a direction normal to the surface.
 - PRO_TRIM_SRF_OFFS_METH_AUTOSCALE—The surface or quilt is thickened by automatically determining scaling coordinate system and fit along all three axes.
 - PRO_TRIM_SRF_OFFS_METH_MANUALSCALE—The surface or quilt is thickened by specifying the scaling coordinate system and control fitting motion.
- PRO_E_SRF_OFFS_CTRLFIT—Specifies the control fit and is applicable for thin trims only. It offsets the surface by creating a best-fit offset that is scaled with respect to a selected coordinate system. It allows you to define the axis for translation. It consists of the following elements:
 - PRO_E_SRF_OFFS_SCALINGCSYS specifies the control fit coordinate system and should have a value of PRO_CSYS.
 - PRO_E_SRF_OFFS_AXISES specifies the control fit axes having the following values:
 - PRO_TRIM_OFFS_TRF_NONE
 - PRO_TRIM_OFFS_TRF_X
 - PRO_TRIM_OFFS_TRF_Y
 - PRO_TRIM_OFFS_TRF_Z
 - PRO_TRIM_OFFS_TRF_XY
 - PRO_TRIM_OFFS_TRF_YZ
 - PRO_TRIM_OFFS_TRF_ZX
 - PRO_TRIM_OFFS_TRF_ALL

- PRO_E_SRF_OFFS_HANDLINGS—Specifies an array of PRO_E_SRF_OFFS_HANDLING and is applicable for thin trims only.
- PRO_E_SRF_OFFS_HANDLING—Specifies special handling item and consists of PRO_E_SRF_OFFS_REF_SEL which are special handling faces which should be from the trimming quilt.

A list of elements required for different types of trims

Trim type	Element Id
Quilt Type	PRO_E_FEATURE_TYPE
	PRO_E_FEATURE_FORM
	PRO_E_SRF_TRIM_TYPE
	PRO_E_STD_FEATURE_NAME
	PRO_E_SURF_TRIM_TYPE
	PRO_E_TRIM_QUILT
	PRO_E_STD_USEQLT_QLT
	PRO_E_MATERIAL_SIDE
	PRO_E_PRIMARY_QLT SIDE
	PRO_E_KEEP_TRIM_SURF_OPT
Use Curve	PRO_E_FEATURE_TYPE
	PRO_E_SRF_TRIM_TYPE
	PRO_E_STD_FEATURE_NAME
	PRO_E_SURF_TRIM_TYPE
	PRO_E_TRIM_QUILT
	PRO_E_STD_CURVE_COLLECTION_APPL
	PRO_E_MATERIAL_SIDE
	PRO_E_PRIMARY_QLT SIDE

Trim type	Element Id
Thin	PRO_E_FEATURE_TYPE
	PRO_E_FEATURE_FORM
	PRO_E_SRF_TRIM_TYPE
	PRO_E_STD_FEATURE_NAME
	PRO_E_SURF_TRIM_TYPE
	PRO_E_TRIM_QUILT
	PRO_E_STD_USEQLT_QLT
	PRO_E_STD_USEQLT_SIDE
	PRO_E_KEEP_TRIM_SURF_OPT
	PRO_E_THICKNESS
	PRO_E_SRF_OFFS_METHOD
	PRO_E_SRF_OFFS_CTRLFIT
	PRO_E_SRF_OFFS_HANDLINGS
Silhouette	PRO_E_FEATURE_TYPE
	PRO_E_SRF_TRIM_TYPE
	PRO_E_STD_FEATURE_NAME
	PRO_E_SURF_TRIM_TYPE
	PRO_E_TRIM_QUILT
	PRO_E_TRIM_SILH_PLANE
	PRO_E_MATERIAL_SIDE
	PRO_E_PRIMARY_QLT SIDE
	PRO_E_KEEP_TRIM_SURF_OPT

Creating a Trim Feature

Function Introduced

- **ProFeatureCreate()**

Use the function **ProFeatureCreate()** to create a Trim Feature based on element tree input. For more information about **ProFeatureCreate()**, refer to the section “Overview of Feature Creation” of chapter “Element Trees: Principles of Feature Creation”.

Redefining a Trim Feature

Function Introduced

- **ProFeatureRedefine()**

Use the function **ProFeatureRedefine()** to redefine a Trim Feature based on the changes made in the element tree. For more information about **ProFeatureRedefine()**, refer to the section “Feature Redefine” of chapter “Element Trees: Principles of Feature Creation”.

Accessing a Trim Feature

Function Introduced

- **ProFeatureElemtreeExtract()**

Use the function **ProFeatureElemtreeExtract()** to create a feature element tree that describes the contents of a Trim Feature and to retrieve the element tree description of a Trim Feature. For more information about **ProFeatureElemtreeExtract()**, refer to the section “Feature Inquiry” of chapter Element Trees: Principles of Feature Creation.

Offset Feature

Refer to the section Offset Datum Curves in the chapter Element Trees: Datum Curves for details about this feature.

Thicken Feature

The Thicken feature is available for both part and for assembly modes and can be defined with respect to coordinate systems, axes, and surfaces. You can apply this feature to check the thickness of a part in the model. This feature also forms the basis for model analysis and thickness check. The result of the thickness analysis is as follows:

- The thickness is between the design range.
- The thickness exceeds the maximum design value.
- The thickness is below the minimum design value.

Thicken features use predetermined surface features or quilt geometry to add or remove thin material sections in the design. The surface features or quilt geometry provide greater flexibility within the design and enable you to transform the geometry to better meet the design needs.

The Feature Element Tree for Thicken feature in Pro/ENGINEER

The element tree for a Thicken feature is documented in the header file *ProThicken.h*, and has a simple structure. The following figure demonstrates the feature element tree structure:

Figure 33-8: Feature Element Tree for Thicken Feature

```

PRO_E_FEATURE_TREE
|
|-- PRO_E_FEATURE_TYPE
|
|-- PRO_E_FEATURE_FORM
|
|-- PRO_E_FEAT_FORM_ALWAYS_THIN
|
|-- PRO_E_STD_USEQLT_QLT
|
|-- PRO_E_THICKEN_TYPE
|
|-- PRO_E_STD_USEQLT_SIDE
|
|-- PRO_E_USEQLT_SOLID_SIDE
|
|-- PRO_E_THICKNESS
|
|-- PRO_E_SRF_OFFS_METHOD
|
|-- PRO_E_SRF_OFFS_CTRLFIT
|   |
|   |-- PRO_E_SRF_OFFS_SCALINGCSYS
|   |
|   |-- PRO_E_SRF_OFFS_AXISES
|
|-- PRO_E_SRF_OFFS_HANDLINGS
|   |
|   |-- PRO_E_SRF_OFFS_HANDLING
|       |
|       |-- PRO_E_SRF_OFFS_REF_SEL
|
|-- PRO_E_STD_FEATURE_NAME

```

The Thicken element tree contains no non-standard element types. The following list details special information about some of the elements in this tree:

- **PRO_E_FEATURE_TYPE**—**PRO_FEAT_PROTRUSION** or **PRO_FEAT_CUT**.
- **PRO_E_FEATURE_FORM**—Must be **PRO_E_SURFS**.
- **PRO_E_FEAT_FORM_ALWAYS_THIN**—Must be **PRO_THIN**.
- **PRO_SRF_OFFS_METHOD**—Offset method that can be enumerated as follows:
 - **PRO_OFFS_METH_NORMTOSURF**— Specifies the offset of the thickened surface normal to the original surface.
 - **PRO_OFFS_METH_AUTOSCALE**— Specifies autoscale and translates the thickened surface with respect to an automatically determined coordinate system.
 - **PRO_OFFS_METH_MANUALSCALE**— Specifies manual scale.

Note: If you change the offset method for a particular feature, all children of this feature fail.

Creating a Thicken Feature

Function Introduced

- **ProFeatureCreate()**

Use the function **ProFeatureCreate()** to create a Thicken Feature based on element tree input. For more information about **ProFeatureCreate()**, refer to the section “Overview of Feature Creation” of chapter “Element Trees: Principles of Feature Creation”.

Redefining a Thicken Feature

Function Introduced

- **ProFeatureRedefine()**

Use the function **ProFeatureRedefine()** to redefine a Thicken Feature based on the changes made in the element tree. For more information about **ProFeatureRedefine()**, refer to the section “Feature Redefine” of chapter “Element Trees: Principles of Feature Creation”.

Accessing a Thicken Feature

Function Introduced

- **ProFeatureElemtreeExtract()**

Use the function **ProFeatureElemtreeExtract()** to create a feature element tree that describes the contents of a Thicken Feature and to retrieve the element tree description of a Thicken Feature. For more information about **ProFeatureElemtreeExtract()**, refer to the section “Feature Inquiry” of chapter “Element Trees: Principles of Feature Creation”.

Solidify Feature

Solidify features are used to create complex geometry that would be difficult to create using regular solid features.

Solidify features use predetermined surface features or quilt geometry and convert them into solid geometry. You can use Solidify features to add, remove, or replace solid material in the designs. The quilt geometry provides greater flexibility within the design, and the Solidify feature enables the designer to transform the geometry to meet design needs.

The attributes of the Solidify feature are:

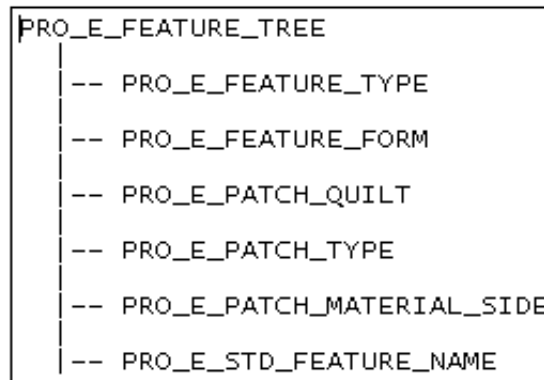
- Patch— Replaces a portion of the surface with quilt. The quilt boundary must lie on the surface.
- Add Material— Fills a quilt with solid material.
- Remove Material— Removes material from inside or outside a quilt.

Note: add material and remove material attributes are always available. Patch is available only if the selected quilt boundaries lie on solid geometry.

The Feature Element Tree for Solidify Feature in Pro/ENGINEER

The element tree for a Solidify feature is documented in the header file *ProSolidify.h*, and has a simple structure. The following figure demonstrates the feature element tree structure

Figure 33-9: Feature Element Tree for Solidify Feature



The solidify element tree contains no non-standard element types. The following list details special information about some of the elements in this tree:

- PRO_E_STD_FEATURE_TYPE—Must be one of the following types—
 - Protrusion— PRO_FEAT_PROTRUSION
 - Cut— PRO_FEAT_CUT
 - Patch— PRO_FEAT_PATCH
- PRO_E_FEATURE_FORM— Must be of the following types:
 - PRO_USE_SURFS— Represents protrusion (PRO_FEAT_PROTRUSION) and cut (PRO_FEAT_CUT).
 - PRO_NOTYPE— Represents patch (PRO_FEAT_PATCH).
- PRO_E_PATCH_QUILT— Specifies the reference quilt and is a mandatory element. A valid quilt or surface satisfies any of the following contexts:

Quilt is open

- CASE 1
 - All boundaries lie on solid surfaces.
 - Solid geometry does not intersect quilt.
 - Solid geometry and quilt form closed empty volume.
- CASE 2

- All boundaries lie on solid surfaces.
- Solid geometry intersects quilt.
- CASE 3
 - All one sided edges are inside solid geometry.
- CASE 4
 - All one sided edges are outside solid geometry.
 - Solid geometry intersects quilt.
- CASE 5
 - Solid geometry and quilt form closed empty volume.
 - At least, one sided edge intersects geometry.
- CASE 6
 - Quilt intersects geometry.
 - None of the one sided edges intersect geometry.
- CASE 7
 - Solid geometry does not intersect quilt.

Quilt is closed

- CASE 1
 - Solid geometry does not intersect quilt.
- CASE 2
 - Quilt completely buried inside Material.
- CASE 3
 - Solid geometry partly intersects quilt.

Creating a Solidify Feature

Function Introduced

- **ProFeatureCreate()**

Use the function **ProFeatureCreate()** to create a Solidify Feature based on element tree input. For more information about **ProFeatureCreate()**, refer to the section “Overview of Feature Creation” of chapter “Element Trees: Principles of Feature Creation”.

Redefining a Solidify Feature

Function Introduced

- **ProFeatureRedefine()**

Use the function **ProFeatureRedefine()** to redefine a Solidify Feature based on the changes made in the element tree. For more information about **ProFeatureRedefine()**, refer to the section “Feature Redefine” of chapter “Element Trees: Principles of Feature Creation”.

Accessing a Solidify Feature

Function Introduced

- **ProFeatureElemtreeExtract()**

Use the function **ProFeatureElemtreeExtract()** to create a feature element tree that describes the contents of a Solidify Feature and to retrieve the element tree description of a Solidify Feature. For more information about **ProFeatureElemtreeExtract()**, refer to the section “Feature Inquiry” of chapter “Element Trees: Principles of Feature Creation”.

Remove Feature

Use the Remove feature to remove geometry without disturbing the feature history. It allows you to modify existing imported geometry or delete some geometry from a part (not necessarily formed by a single feature) without having to reroute and refine a number of features.

You can select one of the following items for removal:

- Surfaces, surface sets, or intent surfaces
- One closed loop chain of one-sided edges from a single quilt

Geometry removal results in neighboring surfaces being extended or trimmed to converge and close the void. The extended geometry is attached as a solid or surface to the selected surfaces. In case of a one-sided edge chain selected as the reference, the extended geometry can be attached to the selected quilt, or created as a new quilt.

You can exclude contours in multi-contour surfaces from being removed. For periodic features separated by artificial edges (for example, extruded cylinders or revolved features), the feature internally references all the surfaces, even if one of them is selected.

Note:

- All surfaces that are extended or trimmed must be adjacent to the boundary defined by references.
- If an adjacent surface does not need to be extended, it will not be copied.
- Surfaces that are to be extended must be extendable.
- Extended surfaces must converge to form a defined volume.
- No new patches are created when the surfaces are extended.
- If the modified geometry cannot be attached as a solid, it can be manually attached as a quilt.
- Feature operations such as Suppress, Delete, Redefine, Reroute, Copy/Paste, and Pattern (limited to reference pattern if patterning by itself) are supported.
- Transformation of this feature by itself is not applicable and allowed.

Feature Element Tree for the Remove Feature

The element tree for the Remove feature is documented in the header file *ProRemoveSurf.h*. The following figure demonstrates the element tree structure:

Figure 33-10: Feature Element Tree for Remove Feature

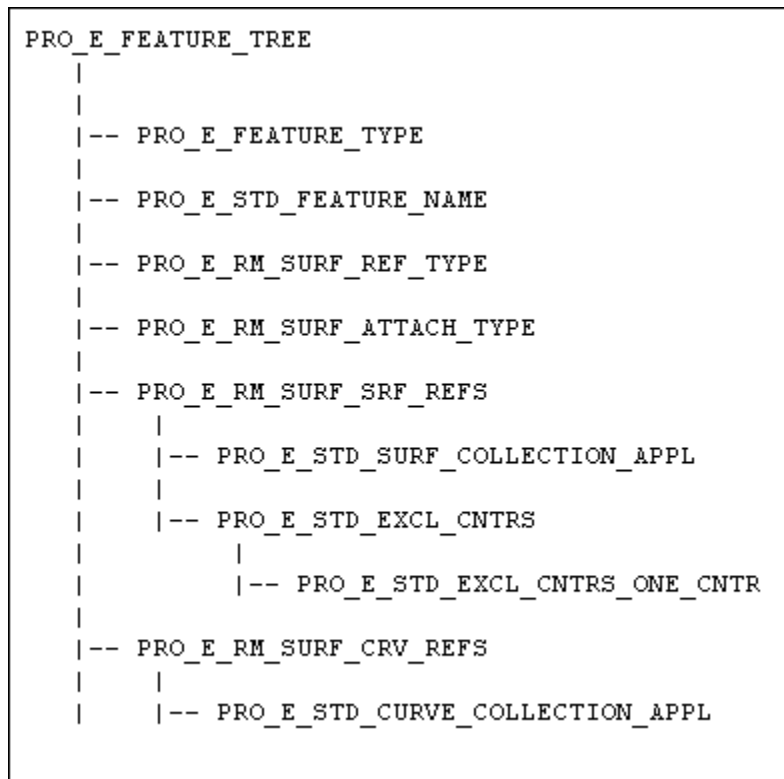
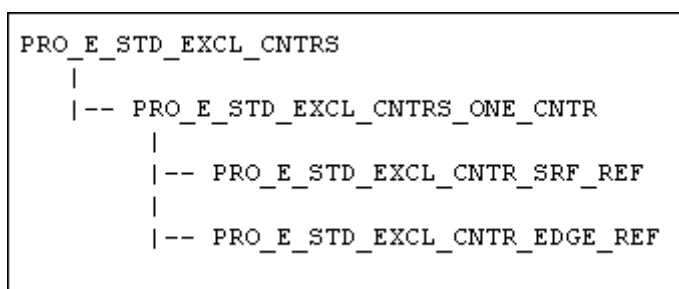


Figure 33-11: PRO_E_STD_EXCL_CNTRS_ONE_CNTR



The Remove element tree contains standard element types. The following list details special information about the elements in this tree:

- **PRO_E_FEATURE_TYPE**—Specifies the feature type and should always have the value **PRO_FEAT_RM_SURF**.

- PRO_E_STD_FEATURE_NAME—Specifies the name of the remove feature.
- PRO_E_RM_SURF_REF_TYPE—Specifies the reference type. This element decides the type of references that you can select for removal. It can have the following values:
 - PRO_RM_SURF_SRF_REF—Specifies that surface sets can be selected as the references.
 - PRO_RM_SURF_CRV_REF—Specifies that a chain of one-sided edges can be selected as the reference.
- PRO_E_RM_SURF_ATTACH_TYPE—Specifies the attachment type. It can have the following values:
 - FM_RM_SURF_ATTACH_SAME—Specifies that the extended geometry will be attached to the selected surfaces if PRO_E_RM_SURF_REF_TYPE is equal to PRO_RM_SURF_SRF_REF, or to the selected quilt if PRO_E_RM_SURF_REF_TYPE is equal to PRO_RM_SURF_CRV_REF.
 - FM_RM_SURF_ATTACH_SEP—Specifies that the extended geometry will be created as a separate quilt if PRO_E_RM_SURF_REF_TYPE is equal to PRO_RM_SURF_CRV_REF.
- PRO_E_RM_SURF_SRF_REFS—Specifies the set of surface references. This element is required if PRO_E_RM_SURF_REF_TYPE is equal to PRO_RM_SURF_SRF_REF. It consists of the following elements:
 - PRO_E_STD_SURF_COLLECTION_APPL—Specifies the surfaces selected for removal. You can select multiple surfaces. This element is required if PRO_E_RM_SURF_REF_TYPE is equal to PRO_RM_SURF_SRF_REF.
 - PRO_E_STD_EXCL_CNTRS—Specifies an array of excluded contours of the type PRO_E_STD_EXCL_CNTRS_ONE_CNTR.
- PRO_E_RM_SURF_CRV_REFS—Specifies the set of curve references. This element is required if PRO_E_RM_SURF_REF_TYPE is equal to PRO_RM_SURF_CRV_REF. It consists of the following element:

- PRO_E_STD_CURVE_COLLECTION_APPL—Specifies the single closed loop chain of one-sided edges. This element is required if PRO_E_RM_SURF_REF_TYPE is equal to PRO_RM_SURF_CRV_REF.

Element Details of PRO_E_STD_EXCL_CNTRS_ONE_CNTR

Each PRO_E_STD_EXCL_CNTRS_ONE_CNTR specifies one excluded contour. It consists of the following elements:

- PRO_E_STD_EXCL_CNTR_SRF_REF—Specifies the surface reference for the excluded contour.
- PRO_E_STD_EXCL_CNTR_EDGE_REF—Specifies the edge reference for the excluded contour. This element can be any edge of the contour.

Creating the Remove Feature

Function Introduced

- **ProFeatureCreate()**

Use the function **ProFeatureCreate()** to create the Remove feature based on the element tree input. For more information about **ProFeatureCreate()**, refer to the section “Overview of Feature Creation” in the chapter “Element Trees: Principles of Feature Creation”.

Redefining the Remove Feature

Function Introduced

- **ProFeatureRedefine()**

Use the function **ProFeatureRedefine()** to redefine the Remove feature based on the changes made in the element tree. For more information about **ProFeatureRedefine()**, refer to the section “Feature Redefine” of chapter “Element Trees: Principles of Feature Creation”.

Accessing the Remove Feature

Function Introduced

- **ProFeatureElemtreeExtract()**

Use the function **ProFeatureElemtreeExtract()** to create a feature element tree that describes the contents of the Remove feature, and to retrieve the element tree description of the Remove feature. For more information about **ProFeatureElemtreeExtract()**, refer to the section Feature Inquiry of chapter Element Trees: Principles of Feature Creation.

Example 1: Creating a Remove Surface Feature

The following example code shows you how to create a remove Surface feature.

```

/*=====*\
FILE      : UgRemoveSurfCreate.c
PURPOSE   : Creating a remove surface feature Edit -> Remove Surface
\*=====*/

/*----- Pro/Toolkit Includes -----*/
#include <ProToolkit.h>
#include <ProMdl.h>
#include <ProElement.h>
#include <ProFeatType.h>
#include <ProFeatForm.h>
#include <ProFeature.h>
#include <ProSection.h>
#include <ProRemoveSurf.h>

/*----- Pro/Develop Includes -----*/

/*----- Application Includes -----*/
#include <TestError.h>
#include <UtilTree.h>
#include <UtilFeats.h>

/*----- Application Typedefs -----*/
/*----- Prototypes -----*/
ProError UtilCreateRemoveSurfaceFeature(ProMdl model, ProSelection
                                         *p_sel);
ProError CreateFeature(ProAsmcomppath *p_comp_path ,ProMdl model,
                      ProElement pro_e_feature_tree,
                      ProFeature *feature);
ProError FeatRemoveSurfExample();

/*=====*\
FUNCTION  : FeatRemoveSurfExample
PURPOSE   : Example to create a remove surface feature
returns   : PRO_TK_NO_ERROR - Successfully executed
\*=====*/
ProError FeatRemoveSurfExample()
{
    ProError status;

```

```

ProMdl model;
ProSelection *surf_sels = NULL;
intn_select = 0;

status = ProMdlCurrentGet (&model);
ERROR_CHECK( "FeatRemoveSurfExample", "ProMdlCurrentGet", status );
if(status != PRO_TK_NO_ERROR)
    return status;

ProStringToWstring ( message_file, "msg_ugfeatcreat.txt" );
status = ProMessageDisplay ( message_file,
                            "USER Select remove surfaces : " );
ERROR_CHECK("ProMessageDisplay", "FeatRemoveSurfExample", status);

status = ProSelect ( "surface", -1, NULL, NULL,
                    NULL, NULL, &surf_sels, &n_select );
ERROR_CHECK("ProSelect", "FeatRemoveSurfExample", status);
if ( n_select <= 0 ) return PRO_TK_GENERAL_ERROR;

status = UtilCreateRemoveSurfaceFeature(model, surf_sels);
if(PRO_TK_NO_ERROR != status)
    status = ProMessageDisplay ( message_file,
                                "USER Feature created successfully" );
else
    status = ProMessageDisplay ( message_file,
                                "USER Feature creation failed");
return status;
}

/*=====*\
FUNCTION : UtilCreateRemoveSurfaceFeature
PURPOSE : Create a remove surface feature from the selection input.
\*=====*/
ProError UtilCreateRemoveSurfaceFeature(ProMdl model,
                                         ProSelection *p_sel)
{
    ProFeature          feature;
    ProSection          section;
    ProAsmcomppath      *p_comp_path = NULL;
    int no_sels;
    int i;
    ProValueData value_data;

    ProSrffcollinstr instr;
    ProReference reference;
    ProSrffcollref isntr_ref;
    ProName wide_string;

    ProError status, rem_surf_status;

```

```

ProElement pro_e_feature_tree;
ProElement pro_e_std_feature_name;
ProElement pro_e_rm_surf_ref_type;
ProElement pro_e_rm_surf_srf_refs;
ProElement pro_e_std_excl_cntrs;

ProCollection srf_collection = NULL;
ProCollection collection;

if(p_sel == NULL)
{
    return PRO_TK_BAD_INPUTS;
}

/*
PRO_E_FEATURE_TREE
|
|
|-- PRO_E_FEATURE_TYPE
|
|-- PRO_E_STD_FEATURE_NAME
|
|-- PRO_E_RM_SURF_REF_TYPE
|
|-- PRO_E_RM_SURF_ATTACH_TYPE */
/*-----*\
Populating root element PRO_E_FEATURE_TREE
\*-----*/
status = ProElementAlloc ( PRO_E_FEATURE_TREE,
                           &pro_e_feature_tree );

/*-----*\
Populating root element PRO_E_FEATURE_TYPE
\*-----*/
value_data.v.i = PRO_FEAT_RM_SURF;
status = ProUtilelemtreeElementAdd( pro_e_feature_tree,
    PRO_E_FEATURE_TYPE,
    ELEM_VALUE_TYPE_INT, &value_data.v.i);

/*-----*\
Populating element PRO_E_STD_FEATURE_NAME
\*-----*/
/*status = ProElementAlloc ( PRO_E_STD_FEATURE_NAME,
    &pro_e_std_feature_name );
ProStringToWstring ( wide_string, "REMOVE_1" );
status = ProElementWstringSet ( pro_e_std_feature_name,
    wide_string );
status = ProElemtreeElementAdd ( pro_e_feature_tree,
    NULL, pro_e_std_feature_name ); */

```



```

/*-----*\
Populating element PRO_E_RM_SURF_REF_TYPE
/*-----*/
value_data.v.i = PRO_RM_SURF_SRF_REF;
status = ProUtilelemtreeElementAdd( pro_e_feature_tree,
    PRO_E_RM_SURF_REF_TYPE,
    ELEM_VALUE_TYPE_INT, &value_data.v.i);

/*-----*\
Populating element PRO_E_RM_SURF_ATTACH_TYPE
/*-----*/
value_data.v.i = FM_RM_SURF_ATTACH_SAME;
status = ProUtilelemtreeElementAdd( pro_e_feature_tree,
    PRO_E_RM_SURF_ATTACH_TYPE,
    ELEM_VALUE_TYPE_INT, &value_data.v.i);

/*
    |
    |-- PRO_E_RM_SURF_SRF_REFS
    |   |
    |   |-- PRO_E_STD_SURF_COLLECTION_APPL
    |   |
    |   |-- PRO_E_STD_EXCL_CNTRS
    |       |
    |       |-- PRO_E_STD_EXCL_CNTRS_ONE_CNTR
    |           |
    |           |-- PRO_E_STD_EXCL_CNTR_SRF_REF
    |           |
    |           |-- PRO_E_STD_EXCL_CNTR_EDGE_REF
    |
    |-- PRO_E_RM_SURF_CRV_REFS
    |   |
    |   |-- PRO_E_STD_CURVE_COLLECTION_APPL
*/

/*-----*\
Populating element PRO_E_RM_SURF_SRF_REFS
/*-----*/
status = ProElementAlloc ( PRO_E_RM_SURF_SRF_REFS,
    &pro_e_rm_surf_srf_refs );
status = ProElemtreeElementAdd ( pro_e_feature_tree,
    NULL, pro_e_rm_surf_srf_refs );
/*-----*\
Creating the collection object and setting the instructions
/*-----*/
status = ProSrfcollectionAlloc ( &collection );
/* status = ProCrvcollinstrAlloc ( PRO_CURVCOLL_ADD_ONE_INSTR,
    &instr ); */
status = ProSrfcollinstrAlloc ( 1, PRO_B_TRUE, &instr );
status = ProSrfcollinstrIncludeSet ( instr, 1 );

```

```

status = ProArraySizeGet ( p_sel, &no_sels);
for (i=0; i<no_sels; i++)
{
    status = ProSelectionToReference ( p_sel[i], &reference );
    ERROR_CHECK("ProMdlToModelitem",
        "UtilCreateRemoveSurfaceFeature", status);
    status = ProSrfcollrefAlloc ( PRO_SURFCOLL_REF_SINGLE,
        reference, &isntr_ref );
    status = ProSrfcollinstrReferenceAdd ( instr, isntr_ref );
    ERROR_CHECK("ProSrfcollinstrReferenceAdd",
        "UtilCreateRemoveSurfaceFeature", status);
}
status = ProSrfcollectionInstructionAdd ( collection, instr );
status = ProSrfcollectionCopy ( collection, &srf_collection );
if ( status != PRO_TK_NO_ERROR )
{
    return status;
}

/*-----*\
Populating element PRO_E_STD_SURF_COLLECTION_APPL
\*-----*/
status = ProUtilElemtreeElementAdd( pro_e_rm_surf_srf_refs,
    PRO_E_STD_SURF_COLLECTION_APPL,
    ELEM_VALUE_TYPE_COLLECTION, srf_collection);

/*-----*\
Populating element PRO_E_STD_EXCL_CNTRS
\*-----*/
status = ProElementAlloc ( PRO_E_STD_EXCL_CNTRS,
    &pro_e_std_excl_cntrs );
status = ProElemtreeElementAdd ( pro_e_rm_surf_srf_refs,
    NULL, pro_e_std_excl_cntrs );

/*-----*\
Creating feature in the model.
\*-----*/
rem_surf_status = ProUtilFeatCreate(p_comp_path,
    model, pro_e_feature_tree, &feature);

/*-----*\
Free up the allocated memory.
\*-----*/
status = ProElementFree (&pro_e_feature_tree);
return rem_surf_status;
}

```

Element Trees: Replace

This chapter describes the basic principles of creating a tweak surface replacement feature. The chapter 'Element Trees: Principles of Feature Creation' is a necessary background for this topic. Read that chapter before this one.

Topic	Page
Introduction	34 - 2
The Feature Element Tree	34 - 2

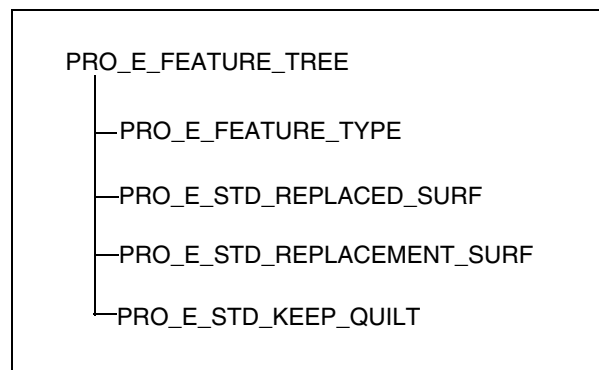
Introduction

The surface replacement functionality enables you to replace the specified surface on a model with a datum plane or quilt. This is similar to the **Replace** option from TWEAK menu in Pro/ENGINEER. See the corresponding section in the *Part Modeling User's Guide* for a detailed description of the restrictions and requirements of the feature.

The Feature Element Tree

The element tree for a surface replacement feature is documented in the header file *ProReplace.h*, and is shown in the following figure.

Figure 34-1: Element Tree for Surface Replacement



The following table describes the elements in the element tree for the surface replacement feature.

Element ID	Data Type	Description
PRO_E_FEATURE_TYPE	PRO_VALUE_TYPE_INT	PRO_FEAT_REPLACE_SURF
PRO_E_STD_REPLACED_SURF	PRO_VALUE_TYPE_SELECTION	The surface to be removed
PRO_E_STD_REPLACEMENT_SURF	PRO_VALUE_TYPE_SELECTION	The replacement surface
PRO_E_STD_KEEP_QUILT	PRO_VALUE_TYPE_INT	Specifies whether to keep the quilt

To keep the replacement surface (datum plane or quilt), add the element `PRO_E_STD_KEEP_QUILT` and set its value to 1. If you omit the element, or its value is 0, the replacement surface will be consumed by the replacement feature.

After you have defined the element tree, call the function **ProFeatureCreate()** to create the tweak surface replacement feature.

Element Trees: Draft Features

This chapter introduces and shows how to create, redefine and access draft features.

Topic	Page
Draft Feature	35 - 2
Variable Pull Direction Draft Feature	35 - 35

Draft Feature

The Draft feature adds a draft angle between -30° and $+30^\circ$ to individual surfaces or to a series of surfaces.

You can draft either solid surfaces or quilt surfaces, but not a combination of both. The first selected surface determines the type of additional surfaces, solid or quilts, that can be selected as draft surfaces for this feature.

Some of the terms associated with the Draft feature are:

- Draft surfaces— The surfaces of the model that are being drafted.
- Draft hinges— Lines or curves on the draft surfaces that the surfaces are pivoted about (also called neutral curves). Draft hinges can be defined by:
 - A plane, in which case the draft surfaces are pivoted about their intersection with the plane.
 - Selecting individual curve chains on the draft surfaces.
- Draft direction— Direction used to measure the draft angle and can be defined in terms of:
 - A plane, in which case the draft direction is normal to this plane.
 - A straight edge or a datum axis, in which case the draft direction is parallel to the edge or axis.
 - Two points, such as datum points or model vertices, in which case the draft direction is parallel to the line connecting the two points.
 - A coordinate system, in which case the draft direction initially defaults to the direction of its x-axis.
- Draft angle—The angle between the draft direction and the resulting drafted surfaces. If the draft surfaces are split, you can define two independent angles for each side of the drafted surface. Draft angles must be within the range of -30° and $+30^\circ$.

Feature Element Tree for the Draft Feature

The element tree for a draft feature is documented in the header file *ProDraft.h*, and has a simple structure. The following figure demonstrates the feature element tree structure:

Figure 35-1: Feature Element tree for Draft Features

```

PRO_E_FEATURE_TREE
|--PRO_E_FEATURE_TYPE
|--PRO_E_STD_FEATURE_NAME
|--PRO_E_DRAFT_TWEAK_OR_INTERSEC
|--PRO_E_DRAFT_EXTEND
|--PRO_E_DRAFT_SPLIT
|--PRO_E_STD_SURF_COLLECTION_APPL
|--PRO_E_DIRECTION_COMPOUND
|--PRO_E_DRAFT_CONSTANT_OR_VARIABLE
|--PRO_E_STD_CURVE_COLLECTION_APPL
|--PRO_E_DRAFT_SPLIT_GEOM
|--PRO_E_STD_SECTION
|  |--PRO_E_SEC_USE_SKETCH
|--PRO_E_DRAFT_INCLUDE_TANGENT
|--PRO_E_DRAFT_SIDE_1
|  |--PRO_E_DRAFT_NEUTRAL_OBJECT_TYPE_1
|  |--PRO_E_DRAFT_NEUTRAL_PLANE_1
|  |--PRO_E_STD_CURVE_COLLECTION_APPL
|  |--PRO_E_DRAFT_DEPENDENT_1
|  |--PRO_E_DRAFT_ANGLE_1
|  |--PRO_E_DRAFT_ANGLES
|  |  |--PRO_E_DRAFT_ANG_PNT
|  |  |  |--PRO_E_STD_POINT_COLLECTION_APPL
|  |  |  |--PRO_E_DRAFT_ANGLE
|--PRO_E_DRAFT_SIDE_2
|  |--PRO_E_DRAFT_NEUTRAL_OBJECT_TYPE_2
|  |--PRO_E_DRAFT_NEUTRAL_PLANE_2
|  |--PRO_E_STD_CURVE_COLLECTION_APPL
|  |--PRO_E_DRAFT_DEPENDENT_2
|  |--PRO_E_DRAFT_ANGLE_2
|  |--PRO_E_DRAFT_ANGLES
|  |  |--PRO_E_DRAFT_ANG_PNT
|  |  |  |--PRO_E_STD_POINT_COLLECTION_APPL
|  |  |  |--PRO_E_DRAFT_ANGLE

```

Element Trees: Draft
Features

The feature element tree contains no non-standard element types. The following list details special information about some of the elements in this tree:

- PRO_E_FEATURE_TYPE—Specifies the feature type.
- PRO_E_STD_FEATURE_NAME—Specifies the name of the feature.
- PRO_E_DRAFT_TWEAK_OR_INTERSEC—Specifies tweak or intersect depending on whether the resulting draft surface encounters an edge of the model or not suggesting the presence of the extend option. It can have any of the following values:
 - PRO_DRAFT_UI_TWEAK for creating regular draft geometry.
 - PRO_DRAFT_UI_INTERSECT for adjusting the draft geometry to intersect an existing edge of the model.
 - PRO_DRAFT_UI_INTERSECT_EXTEND specifies intersect with extend, when the draft does not extend to the adjacent model surface.

Note: It is an option for the earlier versions of Pro/ENGINEER.

- PRO_E_DRAFT_EXTEND—Specifies extend option of the draft. It is of the following types:
 - PRO_DRAFT_UI_NO_EXTEND—Intersect without Extend.
 - PRO_DRAFT_UI_EXTEND—Intersect with Extend.

Note: It is applicable for features created using Pro/ENGINEER version prior to Pro/ENGINEER Wildfire 2.0 Release and is available only when PRO_E_DRAFT_TWEAK_OR_INTERSEC is equal to PRO_DRAFT_UI_INTERSECT.

- PRO_E_DRAFT_SPLIT—Specifies split details of the draft. It can be any of the following types:
 - PRO_DRAFT_UI_SPLIT_NONE
 - PRO_DRAFT_UI_SPLIT_NEUT specifies split on draft hinge.
 - PRO_DRAFT_UI_SPLIT_SURF specifies split at surface.
 - PRO_DRAFT_UI_SPLIT_SCTCH specifies split at sketch.

Note: Draft surfaces can be split either by the draft hinge or by a different curve on the draft surface, such as an intersection with a quilt, or a sketched curve. If you are splitting by a sketch that does not lie on the draft surface, Pro/ENGINEER projects it on the draft surface in the direction normal to the sketching plane.

- PRO_E_STD_SURF_COLLECTION_APPL—Specifies drafted surfaces.
- PRO_E_DIRECTION_COMPOUND—Specifies the direction utility for the draft.
- PRO_E_DRAFT_CONSTANT_OR_VARIABLE—Specifies constant or variable draft. For variable draft one can specify more than one angle per draft side. It can be one of the following types:
 - PRO_DRAFT_UI_VARIABLE
 - PRO_DRAFT_UI_CONSTANT
- PRO_E_STD_CURVE_COLLECTION_APPL—Specifies the exclude loop or the draft hinges.
- PRO_E_DRAFT_SPLIT_GEOM—Specifies split geometry (sketch, plane, or quilt used for splitting) and is an optional element.
- PRO_E_STD_SECTION—Specifies the split geometry and is an optional element. It contains the following element:
 - PRO_E_SEC_USE_SKETCH—Specifies the selected split geometry and is an optional element.
- PRO_E_DRAFT_INCLUDE_TANGENT—Specifies included tangent. It can be any of the following types:
 - PRO_DRAFT_UI_NOT_INC_TANG specifies the non-included tangents.
 - PRO_DRAFT_UI_INC_TANG specifies the included tangents.
- PRO_E_DRAFT_SIDE_1—Specifies details about first draft's side.
- PRO_E_DRAFT_SIDE_2—Specifies details about second draft's side.

Element Details of PRO_E_DRAFT_SIDE_1

Each PRO_E_DRAFT_SIDE_1 has the following elements:

- PRO_E_DRAFT_NEUTRAL_OBJECT_TYPE_1—Specifies the type of draft hinge. It can be any of the following types:
 - PRO_DRAFT_UI_NO_NEUT specifies that no draft hinge have been fixed.
 - PRO_DRAFT_UI_PLANE specifies a plane where the draft surfaces are pivoted about their intersection with this plane.
 - PRO_DRAFT_UI_CURVE specifies a curve chain located on the draft surfaces.
- PRO_E_DRAFT_NEUTRAL_PLANE_1—Specifies the draft hinge.
- PRO_E_STD_CURVE_COLLECTION_APPL—Specifies exclude loop or draft hinges.
- PRO_E_DRAFT_DEPENDENT_1—Specifies the dependence and controls whether the corresponding sides are drafted and depends on the type of the draft hinge. It can be any of the following types:
 - PRO_DRAFT_UI_INDEPENDENT specifies that two independent draft angles for each side of the drafted surface.
 - PRO_DRAFT_UI_DEPENDENT specifies a single draft angle, with the second side drafted in the opposite direction.
 - PRO_DRAFT_UI_NONE specifies that none of the sides be drafted.
- PRO_E_DRAFT_ANGLE_1—Specifies the draft angle and is a constant value.
- PRO_E_DRAFT_ANGLES—This is an option for a variable draft. It specifies a collection of draft angles and points PRO_E_DRAFT_ANG_PNT. Each PRO_E_DRAFT_ANG_PNT consists of the following elements:
 - PRO_E_STD_POINT_COLLECTION_APPL—Specifies the point collection for the angle.
 - PRO_E_DRAFT_ANGLE—Specifies the draft angle.

Element Details of PRO_E_DRAFT_SIDE_2

Each PRO_E_DRAFT_SIDE_2 has the following elements:

- PRO_E_DRAFT_NEUTRAL_OBJECT_TYPE_2—Specifies the type of draft hinge. It can be any of the following types:
 - PRO_DRAFT_UI_NO_NEUT specifies that no draft hinge has been fixed.
 - PRO_DRAFT_UI_PLANE specifies a plane where the draft surfaces are pivoted about their intersection with this plane.
 - PRO_DRAFT_UI_CURVE specifies a curve chain located on the draft surfaces.
 - PRO_E_DRAFT_NEUTRAL_PLANE_2—Specifies the draft hinge.
- PRO_E_STD_CURVE_COLLECTION_APPL—Specifies exclude loop or draft hinges.
- PRO_E_DRAFT_DEPENDENT_2—Specifies the dependence and controls whether the corresponding sides are drafted and depends on the type of the draft hinge. It can be any of the following types:
 - PRO_DRAFT_UI_INDEPENDENT specifies that two independent draft angles for each side of the drafted surface.
 - PRO_DRAFT_UI_DEPENDENT specifies a single draft angle, with the second side drafted in the opposite direction.
 - PRO_DRAFT_UI_NONE specifies that none of the side be drafted.
- PRO_E_DRAFT_ANGLE_2—Specifies the draft angle and is a constant value.
- PRO_E_DRAFT_ANGLES—This is an option for a variable draft. It specifies a collection of draft angles and points PRO_E_DRAFT_ANG_PNT. Each PRO_E_DRAFT_ANG_PNT consists of the following elements:
 - PRO_E_STD_POINT_COLLECTION_APPL—Specifies the point collection for the angle.
 - PRO_E_DRAFT_ANGLE—Specifies the draft angle.

Creating a Draft

Function Introduced:

- **ProFeatureCreate()**

Use the function **ProFeatureCreate()** to create a Draft based on element tree input. For more information about **ProFeatureCreate()**, refer to the section “Overview of Feature Creation” of chapter “Element Trees: Principles of Feature Creation”.

Redefining a Draft

Function Introduced:

- **ProFeatureRedefine()**

Use the function **ProFeatureRedefine()** to redefine a Draft based on the changes made in the element tree. For more information about **ProFeatureRedefine()**, refer to the section “Feature Redefine” of chapter “Element Trees: Principles of Feature Creation”.

Accessing a Draft

Function Introduced:

- **ProFeatureElemtreeExtract()**

Use the function **ProFeatureElemtreeExtract()** to create a feature element tree that describes the contents of a Draft and to retrieve the element tree description of a Draft. For more information about **ProFeatureElemtreeExtract()**, refer to the section “Feature Inquiry” of chapter “Element Trees: Principles of Feature Creation”.

Example 1: Creation of a Draft Feature

The example shows how to create a Simple Draft feature.

The user is prompted to select

- 1) The surface to be drafted
- 2) The hinge (edge / curve)
- 3) The direction of the draft (axis, edge)

```

/*----- Pro/Toolkit Includes -----*/
#include "ProToolkit.h"
#include "ProFeature.h"
#include "ProElemId.h"
#include "ProExtrude.h"
#include "ProModFeat.h"
#include "ProStdSection.h"
#include "ProElement.h"
#include "ProElempath.h"
#include "ProFeatType.h"
#include "ProFeatForm.h"
#include "ProSelection.h"
#include "ProSection.h"
#include "ProModelitem.h"

#include "ProSrffcollection.h"
#include "ProCrvcollection.h"

#include "ProDraft.h"
#include "ProDirection.h"

static ProFileName message_file;

#define C_LOG(a) ProTKPrintf ("Status for %s is = %d\n", a, status ); \
ERROR_CHECK( a, "UgDraftCreate.c", status );

#define C_PRINT(a) ProTKPrintf ( "%s\n", a);

/*=====*\
FUNCTION : UserSimpleDraftCreate()
PURPOSE : Function for creation of Simple Draft features.
\*=====*/
ProError UserSimpleDraftCreate()
{
    ProErrorlist          errors;
    ProMdl                model;
    ProSelection           model_sel;
    ProFeature             feature;
    ProFeatureCreateOptions opts[1];
    ProElempath            path;
    ProElempathItem        path_items[2];
    ProSection             section;
    ProAsmcomppath         comp_path;
    ProAsmcomppath         *p_comp_path = NULL;

    ProCollection collection;
    ProCrvcollinstr crv_instr;
    ProModelitem model_item;
    ProReference reference;
    ProName wide_string;
    ProSrffcollinstr srf_instr;

```

```

ProSrffcollref isntr_ref;

ProElement pro_e_feature_tree;
ProElement pro_e_feature_type;
ProElement pro_e_std_feature_name;
ProElement pro_e_draft_tweak_or_intersec;
ProElement pro_e_draft_extend;
ProElement pro_e_draft_split;
ProElement pro_e_std_surf_collection_appl;
ProElement pro_e_direction_compound;
ProElement pro_e_direction_reference;
ProElement pro_e_direction_flip;
ProElement pro_e_std_section;
ProElement pro_e_std_sec_method;
ProElement pro_e_std_sec_select;
ProElement pro_e_std_curve_collection_appl;
ProElement pro_e_surf_chain_cmpnd;
ProElement pro_e_surf_chain_method;
ProElement pro_e_surf_chain_ref_surfs;
ProElement pro_e_surf_chain_surf;
ProElement pro_e_std_sec_setup_plane;
ProElement pro_e_std_sec_plane_view_dir;
ProElement pro_e_std_sec_plane_orient_dir;
ProElement pro_e_draft_constant_or_variable;
ProElement pro_e_draft_include_tangent;
ProElement pro_e_draft_side_1;
ProElement pro_e_draft_neutral_object_type_1;
ProElement pro_e_draft_dependent_1;
ProElement pro_e_draft_angle_1;
ProElement pro_e_draft_angles;
ProElement pro_e_draft_side_2;
ProElement pro_e_draft_neutral_object_type_2;
ProElement pro_e_draft_dependent_2;
ProElement pro_e_draft_angle_2;

ProStringToWstring ( message_file, "utilities.txt" );

status = ProMdlCurrentGet (&model);
if ( status != PRO_TK_NO_ERROR ) return ( status );

/*-----*\
   Populating root element PRO_E_FEATURE_TREE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TREE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TREE,
                          &pro_e_feature_tree );
C_LOG( " ProElementAlloc " );

/*-----*\
   Populating element PRO_E_FEATURE_TYPE
\*-----*/

```



```

C_PRINT( " *** Processing Element PRO_E_FEATURE_TYPE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TYPE,
                           &pro_e_feature_type );
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_feature_type,
                                PRO_FEAT_DRAFT );
C_LOG( " ProElementIntegerSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_feature_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
Populating element PRO_E_STD_FEATURE_NAME
- Optional
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_FEATURE_NAME *** " );
status = ProElementAlloc ( PRO_E_STD_FEATURE_NAME,
                           &pro_e_std_feature_name );
C_LOG( " ProElementAlloc " );
ProStringToWstring ( wide_string, "DRAFT_ID_XXX" );
status = ProElementWstringSet ( pro_e_std_feature_name,
                                wide_string );
C_LOG( " ProElementWstringSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_std_feature_name );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
Populating element PRO_E_DRAFT_EXTEND
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DRAFT_EXTEND *** " );
status = ProElementAlloc ( PRO_E_DRAFT_EXTEND,
                           &pro_e_draft_extend );
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_draft_extend,
                                PRO_DRAFT_UI_NO_EXTEND );
C_LOG( " ProElementIntegerSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_draft_extend );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
Populating element PRO_E_DRAFT_SPLIT
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DRAFT_SPLIT *** " );
status = ProElementAlloc ( PRO_E_DRAFT_SPLIT,
                           &pro_e_draft_split );
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_draft_split,
                                PRO_DRAFT_UI_SPLIT_NONE );
C_LOG( " ProElementIntegerSet" );

```

```

status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_draft_split );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
Populating element PRO_E_STD_SURF_COLLECTION_APPL
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SURF_COLLECTION_APPL
        *** " );
status = ProElementAlloc ( PRO_E_STD_SURF_COLLECTION_APPL,
                           &pro_e_std_surf_collection_appl );
C_LOG( " ProElementAlloc " );
status = ProSrfcollectionAlloc ( &collection );
C_LOG( " ProSrfcollectionAlloc" );
status = ProSrfcollinstrAlloc ( 1, PRO_B_FALSE, &srf_instr );
C_LOG( " ProSrfcollinstrAlloc" );
status = ProSrfcollinstrIncludeSet ( srf_instr, PRO_B_TRUE );
C_LOG( " ProSrfcollinstrIncludeSet" );
{
    ProSelection *sel_array;
    int n_sel_array;

    ProTKPrintf(" Please select surface,sldface,qltface
                SURFACE to be drafted \n");
    status = ProMessageDisplay ( message_file, "USER Select
                                reference." );
    C_LOG( " ProMessageDisplay" );
    status = ProSelect ( "surface,sldface,qltface", 1, NULL,
                        NULL, NULL, NULL, &sel_array,
                        &n_sel_array );
    C_LOG( " ProSelect" );

    if ( n_sel_array <= 0 )
        return PRO_TK_GENERAL_ERROR;

    status = ProSelectionToReference ( sel_array[0],
                                      &reference );
    C_LOG( " ProSelectionToReference" );
}
status = ProSrfcollrefAlloc ( PRO_SURFCOLL_REF_SINGLE,
                              reference, &isntr_ref );
C_LOG( " ProSrfcollrefAlloc" );
status = ProSrfcollinstrReferenceAdd ( srf_instr, isntr_ref );
C_LOG( " ProSrfcollinstrReferenceAdd" );
status = ProSrfcollectionInstructionAdd ( collection,
                                          srf_instr );
C_LOG( " ProSrfcollectionInstructionAdd" );
status = ProElementCollectionSet( pro_e_std_surf_collection_appl,
                                  collection );
C_LOG( " ProElementCollectionSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,

```

```

                                pro_e_std_surf_collection_appl );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
Populating element PRO_E_DIRECTION_COMPOUND
\*-----*/
C_PRINT( " *** Processing Element
          PRO_E_DIRECTION_COMPOUND *** " );
status = ProElementAlloc ( PRO_E_DIRECTION_COMPOUND,
                           &pro_e_direction_compound );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_direction_compound );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
Populating element PRO_E_DIRECTION_REFERENCE
\*-----*/
C_PRINT( " *** Processing
          Element PRO_E_DIRECTION_REFERENCE *** " );
status = ProElementAlloc ( PRO_E_DIRECTION_REFERENCE,
                           &pro_e_direction_reference );
C_LOG( " ProElementAlloc " );

{
ProSelection *sel_array;
int n_sel_array;

ProTKPrintf(" Please select edge,curve,
            axis DRAFT DIRECTION \n");
status = ProMessageDisplay ( message_file,
                            "USER Select reference." );
C_LOG( " ProMessageDisplay" );
status = ProSelect ( "edge,curve,axis", 1, NULL, NULL,
                    NULL, NULL,&sel_array, &n_sel_array );
C_LOG( " ProSelect" );

if ( n_sel_array <= 0 )
    return PRO_TK_GENERAL_ERROR;

status = ProSelectionToReference ( sel_array[0], &reference );
C_LOG( " ProSelectionToReference" );
}
status = ProElementReferenceSet( pro_e_direction_reference,
                                reference );
C_LOG( " ProElementReferenceSet " );
status = ProElemtreeElementAdd ( pro_e_direction_compound, NULL,
                                pro_e_direction_reference );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\

```

```

    Populating element PRO_E_DIRECTION_FLIP
    \*-----*/
    C_PRINT( " *** Processing Element PRO_E_DIRECTION_FLIP *** " );
    status = ProElementAlloc ( PRO_E_DIRECTION_FLIP, &pro_e_direction_flip
);
    C_LOG( " ProElementAlloc " );
    status = ProElementIntegerSet ( pro_e_direction_flip,
    PRO_DIRECTION_FLIP_ALONG );
    /* ProDirectionFlipOption :: ProDirection.h */
    C_LOG( " ProElementIntegerSet " );
    status = ProElemtreeElementAdd ( pro_e_direction_compound, NULL,
    pro_e_direction_flip );
    C_LOG( " ProElemtreeElementAdd " );

    \*-----*\
    Populating element PRO_E_DRAFT_CONSTANT_OR_VARIABLE
    \*-----*/
    C_PRINT( " *** Processing Element PRO_E_DRAFT_CONSTANT_OR_VARIABLE *** "
);
    status = ProElementAlloc ( PRO_E_DRAFT_CONSTANT_OR_VARIABLE,
&pro_e_draft_constant_or_variable );
    C_LOG( " ProElementAlloc " );
    status = ProElementIntegerSet ( pro_e_draft_constant_or_variable,
    PRO_DRAFT_UI_CONSTANT ); /* 1 */
    C_LOG( " ProElementIntegerSet " );
    status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
    pro_e_draft_constant_or_variable );
    C_LOG( " ProElemtreeElementAdd " );

    \*-----*\
    Populating element PRO_E_DRAFT_INCLUDE_TANGENT
    \*-----*/
    C_PRINT( " *** Processing Element PRO_E_DRAFT_INCLUDE_TANGENT *** " );
    status = ProElementAlloc ( PRO_E_DRAFT_INCLUDE_TANGENT,
&pro_e_draft_include_tangent );
    C_LOG( " ProElementAlloc " );
    status = ProElementIntegerSet ( pro_e_draft_include_tangent,
    PRO_DRAFT_UI_INC_TANG );
    C_LOG( " ProElementIntegerSet " );
    status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
    pro_e_draft_include_tangent );
    C_LOG( " ProElemtreeElementAdd " );

    \*-----*\
    Populating element PRO_E_DRAFT_SIDE_1
    \*-----*/
    C_PRINT( " *** Processing Element PRO_E_DRAFT_SIDE_1 *** " );
    status = ProElementAlloc ( PRO_E_DRAFT_SIDE_1, &pro_e_draft_side_1 );
    C_LOG( " ProElementAlloc " );
    status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
    pro_e_draft_side_1 );

```

```

C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DRAFT_NEUTRAL_OBJECT_TYPE_1
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DRAFT_NEUTRAL_OBJECT_TYPE_1 ***
" );
    status = ProElementAlloc ( PRO_E_DRAFT_NEUTRAL_OBJECT_TYPE_1,
&pro_e_draft_neutral_object_type_1 );
    C_LOG( " ProElementAlloc " );
    status = ProElementIntegerSet ( pro_e_draft_neutral_object_type_1,
PRO_DRAFT_UI_CURVE );
    C_LOG( " ProElementIntegerSet" );
    status = ProElemtreeElementAdd ( pro_e_draft_side_1, NULL,
pro_e_draft_neutral_object_type_1 );
    C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_CURVE_COLLECTION_APPL
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_CURVE_COLLECTION_APPL *** "
);
    status = ProElementAlloc ( PRO_E_STD_CURVE_COLLECTION_APPL,
&pro_e_std_curve_collection_appl );
    C_LOG( " ProElementAlloc " );
    status = ProCrvcollectionAlloc ( &collection );
    C_LOG( " ProCrvcollectionAlloc" );
    status = ProCrvcollinstrAlloc ( PRO_CURV_COLL_ADD_ONE_INSTR, &crv_instr
);
    C_LOG( " ProCrvcollinstrAlloc" );
    status = ProCrvcollinstrAttributeSet ( crv_instr, PRO_CURV_COLL_NO_ATTR
);
    C_LOG( " ProCrvcollinstrAttributeSet" );
    {
        ProSelection *sel_array;
        int n_sel_array;

        ProTKPrintf(" Please select edge,curve,axis DRAFT HINGE \n");
        status = ProMessageDisplay ( message_file, "USER Select reference." );
        C_LOG( " ProMessageDisplay" );
        status = ProSelect ( "edge,curve", 1, NULL, NULL, NULL, NULL,
        &sel_array, &n_sel_array );
        C_LOG( " ProSelect" );

        if ( n_sel_array <= 0 )
            return PRO_TK_GENERAL_ERROR;

        status = ProSelectionToReference ( sel_array[0], &reference );
        C_LOG( " ProSelectionToReference" );
    }
    status = ProCrvcollinstrReferenceAdd ( crv_instr, reference );

```

```

C_LOG( " ProCrvcollinstrReferenceAdd" );
status = ProCrvcollectionInstructionAdd ( collection, crv_instr );
C_LOG( " ProCrvcollectionInstructionAdd" );
status = ProElementCollectionSet( pro_e_std_curve_collection_appl,
collection );
C_LOG( " ProElementCollectionSet " );
status = ProElemtreeElementAdd ( pro_e_draft_side_1, NULL,
pro_e_std_curve_collection_appl );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DRAFT_DEPENDENT_1
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DRAFT_DEPENDENT_1 *** " );
status = ProElementAlloc ( PRO_E_DRAFT_DEPENDENT_1,
&pro_e_draft_dependent_1 );
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_draft_dependent_1,
PRO_DRAFT_UI_INDEPENDENT );
C_LOG( " ProElementIntegerSet" );
status = ProElemtreeElementAdd ( pro_e_draft_side_1, NULL,
pro_e_draft_dependent_1 );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DRAFT_ANGLE_1
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DRAFT_ANGLE_1 *** " );
status = ProElementAlloc ( PRO_E_DRAFT_ANGLE_1, &pro_e_draft_angle_1 );
C_LOG( " ProElementAlloc " );
status = ProElementDoubleSet ( pro_e_draft_angle_1, 10.000000 );
C_LOG( " ProElementDoubleSet" );
status = ProElemtreeElementAdd ( pro_e_draft_side_1, NULL,
pro_e_draft_angle_1 );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Creating feature in the current model.
\*-----*/
status = ProMdlToModelitem( model, &model_item );
status = ProSelectionAlloc ( p_comp_path, &model_item,
    &model_sel);

opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMS;
status = ProFeatureCreate (model_sel, pro_e_feature_tree, opts, 1,
    &feature, &errors);
C_LOG ( " ProFeatureCreate");

/*-----*\
    Free up the allocated memory.
\*-----*/

```

```

    status = ProElementFree (&pro_e_feature_tree );
    C_LOG ( " ProElementFree");

    return (status);
}

```

Example 2: Creation of a Draft Feature using interactive collection

The example shows how to create a Draft feature using interactive collection.

The user is prompted to create a collection of

- 1) The surface to be drafted ("ProSurfacesCollect")
- 2) The direction of the draft (axis, edge - "ProSelect")
- 3) The hinge (edge / curve - "ProCurvesCollect")

```

/*----- Pro/Toolkit Includes -----*/
#include "ProToolkit.h"
#include "ProFeature.h"
#include "ProElemId.h"
#include "ProExtrude.h"
#include "ProModFeat.h"
#include "ProStdSection.h"
#include "ProElement.h"
#include "ProElempath.h"
#include "ProFeatType.h"
#include "ProFeatForm.h"
#include "ProSelection.h"
#include "ProSection.h"
#include "ProModelitem.h"

#include "ProCollect.h"
#include "ProSrfcollection.h"
#include "ProCrvcollection.h"

#include "ProDraft.h"
#include "ProDirection.h"

static ProFileName message_file;

#define C_LOG(a) ProTKPrintf ("Status for %s is = %d\n", a, status ); \
ERROR_CHECK( a, "UgIntcollectionDraftCreate.c", status );

#define C_PRINT(a) ProTKPrintf ( "%s\n", a);

/*=====*\
FUNCTION : UserIntcollectionDraftCreate()

```

```

PURPOSE : Function for creation of Draft features.
\*=====*/
ProError UserIntcollectionDraftCreate()
{
    ProError          status;
    ProErrorlist      errors;
    ProMdl            model;
    ProSelection      model_sel;
    ProFeature        feature;
    ProFeatureCreateOptions opts[1];
    ProElempath       path;
    ProElempathItem   path_items[2];
    ProSection        section;
    ProAsmcomppath    comp_path;
    ProAsmcomppath    *p_comp_path = NULL;

    ProCollection collection;
    ProCrvcollinstr crv_instr;
    ProModelitem model_item;
    ProReference reference;
    ProName wide_string;
    ProSrvcollinstr srf_instr;
    ProSrvcollref isnr_ref;

    ProSelection * coll_sels;
    int coll_sels_size;
    ProSelection *sel_array;
    int n_sel_array;
    ProCollection tmp_collection;
    ProCollectioninstrType srf_coll_filters[] = { PRO_SURFCOLL_SINGLE_SURF
};
    ProCollectioninstrType coll_flags[6] = {
        PRO_CURV_COLL_ONE_BY_ONE,
        PRO_CURV_COLL_TAN_CHAIN,
        PRO_CURV_COLL_CURVE_CHAIN,
        PRO_CURV_COLL_BNDRY_CHAIN,
        PRO_CURV_COLL_SURF_CHAIN,
        PRO_CURV_COLL_LOG_EDGE };

    ProElement pro_e_feature_tree;
    ProElement pro_e_feature_type;
    ProElement pro_e_std_feature_name;
    ProElement pro_e_draft_tweak_or_intersec;
    ProElement pro_e_draft_extend;
    ProElement pro_e_draft_split;
    ProElement pro_e_std_surf_collection_appl;
    ProElement pro_e_direction_compound;
    ProElement pro_e_direction_reference;
    ProElement pro_e_direction_flip;
    ProElement pro_e_std_section;
    ProElement pro_e_std_sec_method;

```



```

ProElement pro_e_std_sec_select;
ProElement pro_e_std_curve_collection_appl;
ProElement pro_e_surf_chain_cmpnd;
ProElement pro_e_surf_chain_method;
ProElement pro_e_surf_chain_ref_surfs;
ProElement pro_e_surf_chain_surf;
ProElement pro_e_std_sec_setup_plane;
ProElement pro_e_std_sec_plane_view_dir;
ProElement pro_e_std_sec_plane_orient_dir;
ProElement pro_e_draft_constant_or_variable;
ProElement pro_e_draft_include_tangent;
ProElement pro_e_draft_side_1;
ProElement pro_e_draft_neutral_object_type_1;
ProElement pro_e_draft_dependent_1;
ProElement pro_e_draft_angle_1;
ProElement pro_e_draft_angles;
ProElement pro_e_draft_side_2;
ProElement pro_e_draft_neutral_object_type_2;
ProElement pro_e_draft_dependent_2;
ProElement pro_e_draft_angle_2;

ProStringToWstring ( message_file, "msg_ugfeatcreat.txt" );

status = ProMdlCurrentGet (&model);
if ( status != PRO_TK_NO_ERROR ) return ( status );

/*-----*\
   Populating root element PRO_E_FEATURE_TREE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TREE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TREE, &pro_e_feature_tree );
C_LOG( " ProElementAlloc " );

/*-----*\
   Populating element PRO_E_FEATURE_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TYPE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TYPE, &pro_e_feature_type );
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_feature_type, PRO_FEAT_DRAFT );
C_LOG( " ProElementIntegerSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_feature_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
   Populating element PRO_E_DRAFT_EXTEND
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DRAFT_EXTEND *** " );
status = ProElementAlloc ( PRO_E_DRAFT_EXTEND, &pro_e_draft_extend );
C_LOG( " ProElementAlloc " );

```

```

    status = ProElementIntegerSet ( pro_e_draft_extend,
PRO_DRAFT_UI_NO_EXTEND );
    C_LOG( " ProElementIntegerSet" );
    status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_draft_extend );
    C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DRAFT_SPLIT
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DRAFT_SPLIT *** " );
status = ProElementAlloc ( PRO_E_DRAFT_SPLIT, &pro_e_draft_split );
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_draft_split,
PRO_DRAFT_UI_SPLIT_NONE );
C_LOG( " ProElementIntegerSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_draft_split );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_SURF_COLLECTION_APPL
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SURF_COLLECTION_APPL *** "
);
status = ProElementAlloc ( PRO_E_STD_SURF_COLLECTION_APPL,
&pro_e_std_surf_collection_appl );
C_LOG( " ProElementAlloc " );

status = ProSrfcollectionAlloc ( &collection );
C_LOG( " ProSrfcollectionAlloc" );

/*-----*\
Interactive Surface Collection
\*-----*/
status = ProMessageDisplay(message_file, "USER Collect surfaces :");
C_LOG ( " ProMessageDisplay")
status = ProSurfacesCollect ( srf_coll_filters,
                                0, ( ProCollFilter ) NULL, ( ProAppData )
                                NULL, collection, &coll_sels,
                                &coll_sels_size );

C_LOG( " ProSurfacesCollect" );

status = ProElementCollectionSet( pro_e_std_surf_collection_appl,
collection );
C_LOG( " ProElementCollectionSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_std_surf_collection_appl );
C_LOG( " ProElemtreeElementAdd" );

```

```

/*-----*\
    Freeing Memory
\*-----*/
status = ProCollectionFree ( &collection );
C_LOG( " ProCollectionFree" );
status = ProSelectionarrayFree ( coll_sels );
C_LOG( " ProSelectionarrayFree" );

/*-----*\
    Populating element PRO_E_DIRECTION_COMPOUND
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DIRECTION_COMPOUND *** " );
status = ProElementAlloc ( PRO_E_DIRECTION_COMPOUND,
&pro_e_direction_compound );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_direction_compound );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DIRECTION_REFERENCE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DIRECTION_REFERENCE *** " );
status = ProElementAlloc ( PRO_E_DIRECTION_REFERENCE,
&pro_e_direction_reference );
C_LOG( " ProElementAlloc " );

ProTKPrintf(" Please select edge,curve,axis DRAFT DIRECTION \n");
status = ProMessageDisplay ( message_file, "USER Select draft direction
:" );
C_LOG( " ProMessageDisplay" );
status = ProSelect ( "edge,curve,axis", 1, NULL, NULL, NULL, NULL,
&sel_array, &n_sel_array );
C_LOG( " ProSelect" );

if ( n_sel_array <= 0 )
    return PRO_TK_GENERAL_ERROR;

/*-----*\
    Converting ProSelection to ProReference
\*-----*/
status = ProSelectionToReference ( sel_array[0], &reference );
C_LOG( " ProSelectionToReference" );

status = ProElementReferenceSet( pro_e_direction_reference, reference );
C_LOG( " ProElementReferenceSet " );
status = ProElemtreeElementAdd ( pro_e_direction_compound, NULL,
pro_e_direction_reference );
C_LOG( " ProElemtreeElementAdd" );

status = ProReferenceFree ( reference );

```

```

C_LOG( " ProReferenceFree" );

/*-----*\
    Populating element PRO_E_DIRECTION_FLIP
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DIRECTION_FLIP *** " );
status = ProElementAlloc ( PRO_E_DIRECTION_FLIP, &pro_e_direction_flip
);
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_direction_flip,
PRO_DIRECTION_FLIP_ALONG );
/* ProDirectionFlipOption :: ProDirection.h */
C_LOG( " ProElementIntegerSet" );
status = ProElemtreeElementAdd ( pro_e_direction_compound, NULL,
pro_e_direction_flip );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DRAFT_CONSTANT_OR_VARIABLE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DRAFT_CONSTANT_OR_VARIABLE *** "
);
status = ProElementAlloc ( PRO_E_DRAFT_CONSTANT_OR_VARIABLE,
&pro_e_draft_constant_or_variable );
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_draft_constant_or_variable,
PRO_DRAFT_UI_CONSTANT ); /* 1 */
C_LOG( " ProElementIntegerSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_draft_constant_or_variable );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DRAFT_INCLUDE_TANGENT
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DRAFT_INCLUDE_TANGENT *** " );
status = ProElementAlloc ( PRO_E_DRAFT_INCLUDE_TANGENT,
&pro_e_draft_include_tangent );
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_draft_include_tangent,
PRO_DRAFT_UI_INC_TANG );
C_LOG( " ProElementIntegerSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_draft_include_tangent );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DRAFT_SIDE_1
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DRAFT_SIDE_1 *** " );
status = ProElementAlloc ( PRO_E_DRAFT_SIDE_1, &pro_e_draft_side_1 );

```

```

C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_draft_side_1 );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DRAFT_NEUTRAL_OBJECT_TYPE_1
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DRAFT_NEUTRAL_OBJECT_TYPE_1 ***
" );
status = ProElementAlloc ( PRO_E_DRAFT_NEUTRAL_OBJECT_TYPE_1,
&pro_e_draft_neutral_object_type_1 );
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_draft_neutral_object_type_1,
PRO_DRAFT_UI_CURVE );
C_LOG( " ProElementIntegerSet" );
status = ProElemtreeElementAdd ( pro_e_draft_side_1, NULL,
pro_e_draft_neutral_object_type_1 );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_CURVE_COLLECTION_APPL
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_CURVE_COLLECTION_APPL *** "
);
status = ProElementAlloc ( PRO_E_STD_CURVE_COLLECTION_APPL,
&pro_e_std_curve_collection_appl );
C_LOG( " ProElementAlloc " );

/*-----*\
Interactive Curve Collection
\*-----*/
status = ProCollectionAlloc ( &tmp_collection );
C_LOG( " ProCollectionAlloc" );

status = ProMessageDisplay(message_file, "USER Collect curves :");
C_LOG ( " ProMessageDisplay")

status = ProCurvesCollect ( (ProChaincollUIControl *)coll_flags,
    6,
    ( ProCrvcollFilter ) NULL,
    ( ProAppData ) NULL,
    &tmp_collection,
    &coll_sels,
    &coll_sels_size );
C_LOG( " ProCurvesCollect" );

/*-----*\
    Converting collection from PRO_COLLECTION_LEGACY to
    PRO_COLLECTION_CRVCOLL

```

```

        type as supported by Wildfire 2.0 feature element trees
/*-----*/
    status = ProCrvcollectionCopy ( tmp_collection, &collection );
    C_LOG( " ProCrvcollectionCopy" );

    status = ProElementCollectionSet( pro_e_std_curve_collection_appl,
collection );
    C_LOG( " ProElementCollectionSet " );
    status = ProElemtreeElementAdd ( pro_e_draft_side_1, NULL,
pro_e_std_curve_collection_appl );
    C_LOG( " ProElemtreeElementAdd" );

/*-----*/
    Freeing Memory
/*-----*/
    status = ProCollectionFree ( &tmp_collection );
    C_LOG( " ProCollectionFree" );
    status = ProCollectionFree ( &collection );
    C_LOG( " ProCollectionFree" );
    status = ProSelectionarrayFree ( coll_sels );
    C_LOG( " ProSelectionarrayFree" );

/*-----*/
    Populating element PRO_E_DRAFT_DEPENDENT_1
/*-----*/
    C_PRINT( " *** Processing Element PRO_E_DRAFT_DEPENDENT_1 *** " );
    status = ProElementAlloc ( PRO_E_DRAFT_DEPENDENT_1,
&pro_e_draft_dependent_1 );
    C_LOG( " ProElementAlloc " );
    status = ProElementIntegerSet ( pro_e_draft_dependent_1,
PRO_DRAFT_UI_INDEPENDENT );
    C_LOG( " ProElementIntegerSet" );
    status = ProElemtreeElementAdd ( pro_e_draft_side_1, NULL,
pro_e_draft_dependent_1 );
    C_LOG( " ProElemtreeElementAdd" );

/*-----*/
    Populating element PRO_E_DRAFT_ANGLE_1
/*-----*/
    C_PRINT( " *** Processing Element PRO_E_DRAFT_ANGLE_1 *** " );
    status = ProElementAlloc ( PRO_E_DRAFT_ANGLE_1, &pro_e_draft_angle_1 );
    C_LOG( " ProElementAlloc " );
    status = ProElementDoubleSet ( pro_e_draft_angle_1, 10.000000 );
    C_LOG( " ProElementDoubleSet" );
    status = ProElemtreeElementAdd ( pro_e_draft_side_1, NULL,
pro_e_draft_angle_1 );
    C_LOG( " ProElemtreeElementAdd" );

/*-----*/
    Writing the elemtree in XML format
/*-----*/

```

```

status = ProElemtreeWrite ( pro_e_feature_tree, PRO_ELEMTREE_XML,
    L"draft_tree.out" );
C_LOG( " ProElemtreeWrite" );

/*-----*\
    Creating feature in the current model.
\*-----*/
status = ProMdlToModelitem( model, &model_item );
status = ProSelectionAlloc (p_comp_path, &model_item,
    &model_sel);

opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMS;
status = ProFeatureCreate (model_sel, pro_e_feature_tree, opts, 1,
    &feature, &errors);
C_LOG ( " ProFeatureCreate");

/*-----*\
    Free up the allocated memory.
\*-----*/

status = ProSelectionFree ( &model_sel );
C_LOG ( " ProSelectionFree");

status = ProElementFree ( &pro_e_feature_tree );
C_LOG ( " ProElementFree");

return (status);
}

```

Example 3: Creation of a Draft Feature based on the Object-Action paradigm

The example demonstrates creation of a simple Draft feature based on the Object-Action paradigm using selection buffer access. The user is expected to populate the selection buffer in Pro/ENGINEER UI with

- 1) A surface collection for the surfaces to be drafted
- 2) A curve collection for the hinge (edge / curve)
- 3) A selection for the direction of the draft (axis, edge)

This example parses the selection buffer and uses the information for the programmatic creation of draft feature. The selection buffer may be populated with the above three entries in any order. The success of feature creation depends upon the appropriateness of the data in the selection buffer.

```

/*----- Pro/Toolkit Includes -----*/
#include "ProToolkit.h"

```

```

#include "ProFeature.h"
#include "ProElemId.h"
#include "ProExtrude.h"
#include "ProModFeat.h"
#include "ProStdSection.h"
#include "ProElement.h"
#include "ProElempath.h"
#include "ProFeatType.h"
#include "ProFeatForm.h"
#include "ProSelection.h"
#include "ProSection.h"
#include "ProModelitem.h"

#include "ProSelbuffer.h"
#include "ProCollect.h"
#include "ProSrfcollection.h"
#include "ProCrvcollection.h"

#include "ProDraft.h"
#include "ProDirection.h"

static ProFileName message_file;

#define C_LOG(a) ProTKPrintf ("Status for %s is = %d\n", a, status ); \
ERROR_CHECK( a, "UgOADraftCreate.c", status );

#define C_PRINT(a) ProTKPrintf ( "%s\n", a);

/*=====*\
FUNCTION : UserOADraftCreate()
PURPOSE : Function for OA creation of Draft features.
\*=====*/
ProError UserOADraftCreate()
{
    ProErrorlist          errors;
    ProMdl                model;
    ProSelection          model_sel;
    ProFeature            feature;
    ProFeatureCreateOptions opts[1];
    ProElempath           path;
    ProElempathItem       path_items[2];
    ProSection            section;
    ProAsmcomppath        comp_path;
    ProAsmcomppath        *p_comp_path = NULL;

    ProCollection srf_collection = NULL;
    ProCollection crv_collection = NULL;

    ProModelitem model_item;
    ProReference reference, direction_ref = NULL;
    ProSelection direction_ref_sel = NULL;

```



```

ProName wide_string;

ProSelection * selbuff_sels;
int selbuff_sels_size, selbuff_count;

ProElement pro_e_feature_tree;
ProElement pro_e_feature_type;
ProElement pro_e_std_feature_name;
ProElement pro_e_draft_tweak_or_intersec;
ProElement pro_e_draft_extend;
ProElement pro_e_draft_split;
ProElement pro_e_std_surf_collection_appl;
ProElement pro_e_direction_compound;
ProElement pro_e_direction_reference;
ProElement pro_e_direction_flip;
ProElement pro_e_std_section;
ProElement pro_e_std_sec_method;
ProElement pro_e_std_sec_select;
ProElement pro_e_std_curve_collection_appl;
ProElement pro_e_surf_chain_cmpnd;
ProElement pro_e_surf_chain_method;
ProElement pro_e_surf_chain_ref_surfs;
ProElement pro_e_surf_chain_surf;
ProElement pro_e_std_sec_setup_plane;
ProElement pro_e_std_sec_plane_view_dir;
ProElement pro_e_std_sec_plane_orient_dir;
ProElement pro_e_draft_constant_or_variable;
ProElement pro_e_draft_include_tangent;
ProElement pro_e_draft_side_1;
ProElement pro_e_draft_neutral_object_type_1;
ProElement pro_e_draft_dependent_1;
ProElement pro_e_draft_angle_1;
ProElement pro_e_draft_angles;
ProElement pro_e_draft_side_2;
ProElement pro_e_draft_neutral_object_type_2;
ProElement pro_e_draft_dependent_2;
ProElement pro_e_draft_angle_2;

ProStringToWstring ( message_file, "msg_ugfeatcreat.txt" );

status = ProMdlCurrentGet (&model);
if ( status != PRO_TK_NO_ERROR ) return ( status );

/*-----*\
Processing the contents of the selection buffer
+ Allows only 3 entries in the selection buffer as follows -
  One selection, One curve collection & one surface collection
+ The 3 entries can be in any order, the code is capable of
  extracting the right-ones to get assigned appropriately
  i.e. selection for draft direction, surface collection for
  draft surface and curve collection for draft hinge elements

```

```

        in the final element tree
/*-----*/

status = ProSelbufferSelectionsGet( &selbuff_sels );
C_LOG( " ProSelbufferSelectionsGet " );
if ( status != PRO_TK_NO_ERROR )
    return status;

status = ProArraySizeGet ( selbuff_sels, &selbuff_sels_size );
C_LOG ( " ProArraySizeGet" )

    ProTKPrintf ( "Selections in selbuff = %d\n", selbuff_sels_size );
/*-----*\
    allowing exactly 3 entities in the selection buffer
/*-----*/
    if ( selbuff_sels_size != 3 )
    {
status = ProMessageDisplay(message_file,
                        "USER Invalid selection buffer context:");
C_LOG ( " ProMessageDisplay" )

return PRO_TK_BAD_CONTEXT;
    }

    for ( selbuff_count = 0; selbuff_count < selbuff_sels_size;
selbuff_count++ )
    {
        ProCollection collection;
        ProCollectionType coll_type;
        ProTKPrintf ( "Count in selbuff = %d\n", selbuff_count );
        status = ProSelectionCollectionGet ( selbuff_sels[selbuff_count],
                                            &collection );

        C_LOG ( " ProSelectionCollectionGet" )

/*-----*\
        Processing selection for draft direction
/*-----*/
if ( status != PRO_TK_NO_ERROR )
    {
        if ( direction_ref_sel != NULL )
        {
/*-----*\
            Only one selection allowed for this context
/*-----*/
status = ProMessageDisplay(message_file,
                        "USER Invalid selection buffer context:");
C_LOG ( " ProMessageDisplay" )

return PRO_TK_BAD_CONTEXT;
        }
    }

```

```

/*-----*\
  Extracting the selection in the selection buffer - draft direction
/*-----*/
  status = ProSelectionCopy ( selbuff_sels[selbuff_count],
                             &direction_ref_sel );
  C_LOG ( " ProSelectionCopy" )

/*-----*\
  Converting to ProReference as required by the elemtree
/*-----*/
  status = ProSelectionToReference ( direction_ref_sel,
                                    &direction_ref );
  C_LOG ( " ProSelectionToReference" )
}
else
{
/*-----*\
  Checking for collection types
/*-----*/
  status = ProCollectionTypeGet ( collection, &coll_type );
  C_LOG ( " ProCollectionTypeGet" )

  if ( coll_type == PRO_COLLECTION_SRFCOLL )
{
if ( srf_collection != NULL )
{
  status = ProMessageDisplay(message_file,
                             "USER Invalid selection buffer context:");
  C_LOG ( " ProMessageDisplay" )
  return PRO_TK_BAD_CONTEXT;
}

/*-----*\
  Surface collection for draft surface
/*-----*/
status = ProSrfcollectionCopy ( collection, &srf_collection );
C_LOG ( " ProSrfcollectionCopy" )
  if ( status != PRO_TK_NO_ERROR )
    return status;
}

  else
{
if ( crv_collection != NULL )
{
  status = ProMessageDisplay(message_file,
                             "USER Invalid selection buffer context:");
  C_LOG ( " ProMessageDisplay" )
  return PRO_TK_BAD_CONTEXT;
}
}
}

```

```

/*-----*\
    Curve collection for draft hinge
/*-----*/
status = ProCrvcollectionCopy ( collection, &crv_collection );
C_LOG ( " ProCrvcollectionCopy" )
    if ( status != PRO_TK_NO_ERROR )
        return status;
}

/*-----*\
    Freeing the temporary copy of collection
/*-----*/
status = ProCollectionFree ( &collection );
C_LOG ( " ProCollectionFree" )

}
}

/*-----*\
    Confirming for the required contents processed
/*-----*/

if ( ( direction_ref_sel == NULL ) || ( srf_collection == NULL ) ||
    ( crv_collection == NULL ) )
{
    status = ProMessageDisplay(message_file,
                                "USER Invalid selection buffer context:");
    C_LOG ( " ProMessageDisplay" )
    return PRO_TK_BAD_CONTEXT;
}

/*-----*\
    Populating root element PRO_E_FEATURE_TREE
/*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TREE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TREE, &pro_e_feature_tree );
C_LOG( " ProElementAlloc " );

/*-----*\
    Populating element PRO_E_FEATURE_TYPE
/*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TYPE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TYPE, &pro_e_feature_type );
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_feature_type, PRO_FEAT_DRAFT );
C_LOG( " ProElementIntegerSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_feature_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DRAFT_EXTEND

```

```

\*-----*/
C_PRINT( " *** Processing Element PRO_E_DRAFT_EXTEND *** " );
status = ProElementAlloc ( PRO_E_DRAFT_EXTEND, &pro_e_draft_extend );
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_draft_extend,
                                PRO_DRAFT_UI_NO_EXTEND );
C_LOG( " ProElementIntegerSet" );

status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_draft_extend );
C_LOG( " ProElemtreeElementAdd" );

\*-----*\
    Populating element PRO_E_DRAFT_SPLIT
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DRAFT_SPLIT *** " );
status = ProElementAlloc ( PRO_E_DRAFT_SPLIT, &pro_e_draft_split );
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_draft_split,
                                PRO_DRAFT_UI_SPLIT_NONE );
C_LOG( " ProElementIntegerSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_draft_split );
C_LOG( " ProElemtreeElementAdd" );

\*-----*\
    Populating element PRO_E_STD_SURF_COLLECTION_APPL
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SURF_COLLECTION_APPL *** "
);
status = ProElementAlloc ( PRO_E_STD_SURF_COLLECTION_APPL,
                            &pro_e_std_surf_collection_appl );
C_LOG( " ProElementAlloc " );

\*-----*\
    Assigning the surface collection
\*-----*/
status = ProElementCollectionSet( pro_e_std_surf_collection_appl,
                                srf_collection );
C_LOG( " ProElementCollectionSet " );
status = ProElemtreeElementAdd( pro_e_feature_tree, NULL,
                                pro_e_std_surf_collection_appl );
C_LOG( " ProElemtreeElementAdd" );

\*-----*\
    Populating element PRO_E_DIRECTION_COMPOUND
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DIRECTION_COMPOUND *** " );
status = ProElementAlloc ( PRO_E_DIRECTION_COMPOUND,
                            &pro_e_direction_compound );
C_LOG( " ProElementAlloc " );

```

```

status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_direction_compound );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DIRECTION_REFERENCE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DIRECTION_REFERENCE *** " );
status = ProElementAlloc ( PRO_E_DIRECTION_REFERENCE,
                          &pro_e_direction_reference );
C_LOG( " ProElementAlloc " );

/* Assigning the direction reference */
status = ProElementReferenceSet( pro_e_direction_reference,
                                direction_ref );
C_LOG( " ProElementReferenceSet " );
status = ProElemtreeElementAdd ( pro_e_direction_compound, NULL,
                                pro_e_direction_reference );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DIRECTION_FLIP
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DIRECTION_FLIP *** " );
status = ProElementAlloc ( PRO_E_DIRECTION_FLIP, &pro_e_direction_flip
);
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_direction_flip,
                               PRO_DIRECTION_FLIP_ALONG );
/* ProDirectionFlipOption :: ProDirection.h */
C_LOG( " ProElementIntegerSet" );
status = ProElemtreeElementAdd ( pro_e_direction_compound, NULL,
                                pro_e_direction_flip );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DRAFT_CONSTANT_OR_VARIABLE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DRAFT_CONSTANT_OR_VARIABLE *** "
);
status = ProElementAlloc ( PRO_E_DRAFT_CONSTANT_OR_VARIABLE,
                          &pro_e_draft_constant_or_variable );
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_draft_constant_or_variable,
                               PRO_DRAFT_UI_CONSTANT ); /* 1 */
C_LOG( " ProElementIntegerSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_draft_constant_or_variable );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\

```

```

    Populating element PRO_E_DRAFT_INCLUDE_TANGENT
/*-----*/
C_PRINT( " *** Processing Element PRO_E_DRAFT_INCLUDE_TANGENT *** " );
status = ProElementAlloc ( PRO_E_DRAFT_INCLUDE_TANGENT,
                           &pro_e_draft_include_tangent );
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_draft_include_tangent,
                               PRO_DRAFT_UI_INC_TANG );
C_LOG( " ProElementIntegerSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_draft_include_tangent );
C_LOG( " ProElemtreeElementAdd" );

/*-----*/
    Populating element PRO_E_DRAFT_SIDE_1
/*-----*/
C_PRINT( " *** Processing Element PRO_E_DRAFT_SIDE_1 *** " );
status = ProElementAlloc ( PRO_E_DRAFT_SIDE_1, &pro_e_draft_side_1 );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_draft_side_1 );
C_LOG( " ProElemtreeElementAdd" );

/*-----*/
    Populating element PRO_E_DRAFT_NEUTRAL_OBJECT_TYPE_1
/*-----*/
C_PRINT( " *** Processing Element PRO_E_DRAFT_NEUTRAL_OBJECT_TYPE_1 ***
" );
status = ProElementAlloc ( PRO_E_DRAFT_NEUTRAL_OBJECT_TYPE_1,
                           &pro_e_draft_neutral_object_type_1 );
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_draft_neutral_object_type_1,
                               PRO_DRAFT_UI_CURVE );
C_LOG( " ProElementIntegerSet" );
status = ProElemtreeElementAdd ( pro_e_draft_side_1, NULL,
                                pro_e_draft_neutral_object_type_1 );
C_LOG( " ProElemtreeElementAdd" );

/*-----*/
    Populating element PRO_E_STD_CURVE_COLLECTION_APPL
/*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_CURVE_COLLECTION_APPL *** "
);
status = ProElementAlloc ( PRO_E_STD_CURVE_COLLECTION_APPL,
                           &pro_e_std_curve_collection_appl );
C_LOG( " ProElementAlloc " );

/*-----*/
    Assigning the curve collection
/*-----*/
status = ProElementCollectionSet( pro_e_std_curve_collection_appl,

```

```

                                crv_collection );
C_LOG( " ProElementCollectionSet " );
status = ProElemtreeElementAdd ( pro_e_draft_side_1, NULL,
                                pro_e_std_curve_collection_appl );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DRAFT_DEPENDENT_1
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DRAFT_DEPENDENT_1 *** " );
status = ProElementAlloc ( PRO_E_DRAFT_DEPENDENT_1,
                            &pro_e_draft_dependent_1 );
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_draft_dependent_1,
PRO_DRAFT_UI_INDEPENDENT );
C_LOG( " ProElementIntegerSet" );
status = ProElemtreeElementAdd ( pro_e_draft_side_1, NULL,
                                pro_e_draft_dependent_1 );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_DRAFT_ANGLE_1
\*-----*/
C_PRINT( " *** Processing Element PRO_E_DRAFT_ANGLE_1 *** " );
status = ProElementAlloc ( PRO_E_DRAFT_ANGLE_1, &pro_e_draft_angle_1 );
C_LOG( " ProElementAlloc " );
status = ProElementDoubleSet ( pro_e_draft_angle_1, 10.000000 );
C_LOG( " ProElementDoubleSet" );
status = ProElemtreeElementAdd ( pro_e_draft_side_1, NULL,
                                pro_e_draft_angle_1 );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Writing the elemtree in XML format
\*-----*/
status = ProElemtreeWrite ( pro_e_feature_tree, PRO_ELEMTREE_XML,
                            L"draft_tree.out" );
C_LOG( " ProElemtreeWrite" );

/*-----*\
    Creating feature in the current model.
\*-----*/
status = ProMdlToModelitem( model, &model_item );
status = ProSelectionAlloc ( p_comp_path, &model_item,
                            &model_sel);

opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMS;
status = ProFeatureCreate (model_sel, pro_e_feature_tree, opts, 1,
                            &feature, &errors);
C_LOG ( " ProFeatureCreate");

```



```

/*-----*\
    Freeing allocated memory.
\*-----*/

status = ProSelectionFree (&model_sel );
C_LOG (" ProSelectionFree");

status = ProElementFree (&pro_e_feature_tree );
C_LOG (" ProElementFree");

return (status);
}

```

Variable Pull Direction Draft Feature

The Variable Pull Direction Draft (VPDD) feature differs from the regular Draft feature where the pull direction is restricted to be constant. A VPDD is defined by an edge or curve, a surface that specifies the variable pull direction, a depth option, and optionally, splitting surfaces.

Feature Element Tree for the Variable Pull Direction Draft Feature

The element tree for the Variable Pull Direction Draft feature is documented in the header file *ProVPDD.h*. The following figure demonstrates the feature element tree structure:

Figure 35-2: Feature Element Tree for Variable Pull Direction Draft Feature

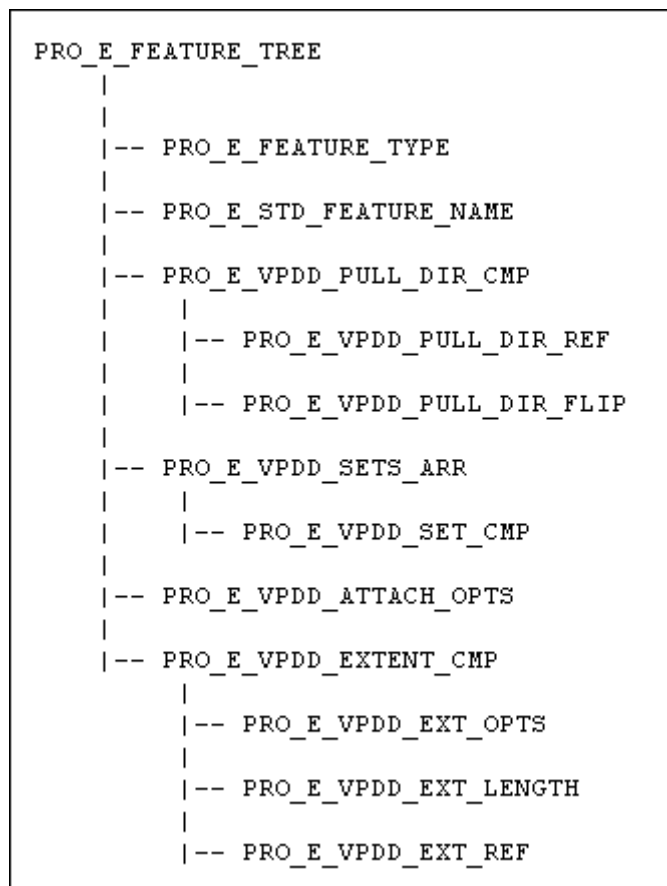
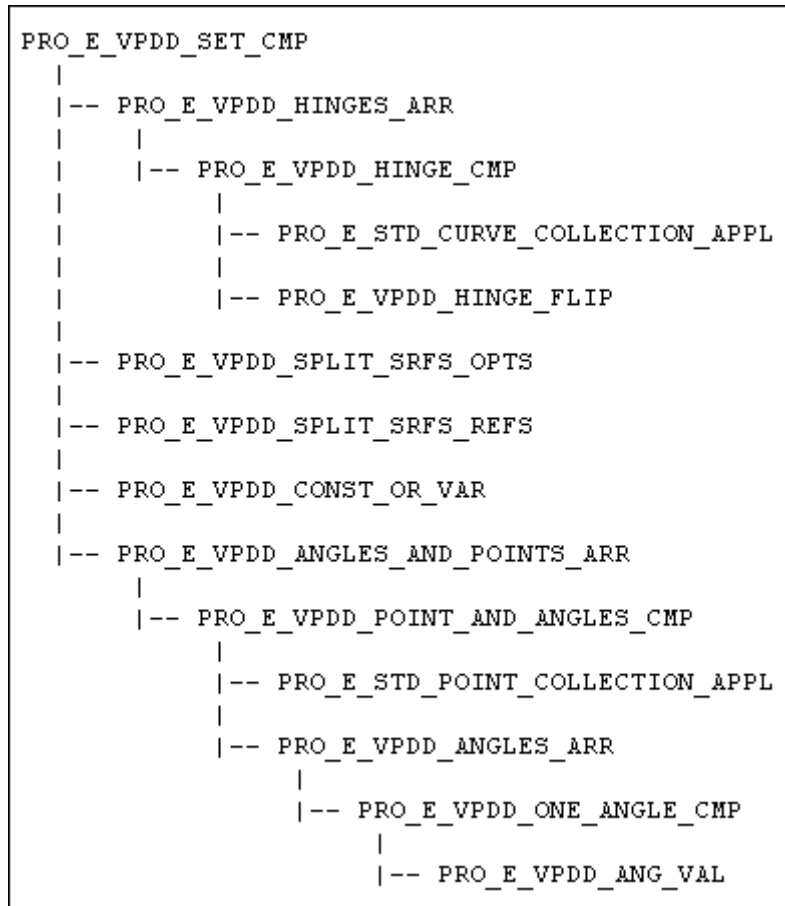


Figure 35-3: PRO_E_VPDD_SET_CMP



The elements in this tree are as follows:

- PRO_E_FEATURE_TYPE—Specifies the feature type and should be PRO_FEAT_VPDD.
- PRO_E_STD_FEATURE_NAME—Specifies the name of the feature.
- PRO_E_VPDD_PULL_DIR_CMP—Specifies the pull direction reference. This compound element consists of the following elements:
 - PRO_E_VPDD_PULL_DIR_REF—Specifies the selected reference surface. It could be a single PRO_QUILT, a single PRO_DATUM_PLANE, a single PRO_LOG_SRF, or multiple PRO_SURFACE references that are tangent to each other.

- PRO_E_VPDD_PULL_DIR_FLIP—Specifies the pull direction defined by the normal vectors of the selected reference surfaces. By default, this element is PRO_B_TRUE and the pull direction is along the normal vectors. The pull direction can be flipped.
- PRO_E_VPDD_SETS_ARR—Specifies an array of the elements of the type PRO_E_VPDD_SET_CMP.
PRO_E_VPDD_SET_CMP is a compound element and specifies a draft set. For more information on the elements contained by this compound element, refer to the section “Element Details of the Subtree PRO_E_VPDD_SET_CMP”.
- PRO_E_VPDD_ATTACH_OPTS—Specifies the attachment option. When all the draft hinges are two-sided edges, you can specify whether to attach the draft geometry to the existing solid or quilt, or create the draft geometry as a separate quilt. The attachment options, specified by the enumerated type *Pro_vpdd_attach_type*, are as follows:
 - PRO_VPDD_ATTACH_NEW_QUILT—Creates the draft geometry as a separate quilt.
 - PRO_VPDD_ATTACH_SAME_QUILT—Attaches the draft geometry to the existing solid or quilt.
- PRO_E_VPDD_EXTENT_CMP—Specifies the extent (depth) option when unattached draft geometry is created. This compound element is available only if the element PRO_E_VPDD_ATTACH_OPTS is set to PRO_VPDD_ATTACH_NEW_QUILT.
- PRO_E_VPDD_EXT_OPTS—Specifies the extent options. These options, specified by the enumerated type *Pro_vpdd_extent_type*, are as follows:
 - PRO_VPDD_EXT_LENGTH—Specifies the length dimension option.
 - PRO_VPDD_EXT_TO_SEL—Select the reference bottom surface upto which the draft geometry is extended.
 - PRO_VPDD_EXT_TO_NEXT—The draft geometry is extended up to the next surface it intersects. This excludes any surface or quilt used as a parting surface for a different draft set.
 - PRO_VPDD_EXT_UNATTACHED—Creates the draft geometry ready for attachment but as a separate quilt.

- PRO_E_VPDD_EXT_LENGTH—Specifies the length value. This element is available only when the element PRO_E_VPDD_EXT_OPTS is set to PRO_VPDD_EXT_LENGTH.
- PRO_E_VPDD_EXT_REF—Specifies the selected bottom surface upto which the draft geometry is extended. The valid references are PRO_QUILT, PRO_SURFACE, and PRO_DATUM_PLANE. This element is available only when the element PRO_E_VPDD_EXT_OPTS is set to PRO_VPDD_EXT_TO_SEL.

Element Details of the Subtree PRO_E_VPDD_SET_CMP

The compound element PRO_E_VPDD_SET_CMP contains the following elements:

- PRO_E_VPDD_HINGES_ARR—Specifies an array of elements of the type PRO_E_VPDD_HINGE_CMP. PRO_E_VPDD_HINGE_CMP is a compound element and specifies a collection of draft hinges used to generate the draft geometry.
 - PRO_E_STD_CURVE_COLLECTION_APPL—Specifies a chain of two-sided edges, one-sided edges, curves, or a combination of one-sided edges and curves.
 - PRO_E_VPDD_HINGE_FLIP—Specifies the *ProBoolean* option to select the sets of surfaces to be drafted on each side of a draft hinge defined by two-sided edges.
- PRO_E_VPDD_SPLIT_SRFS_OPTS—Specifies the *ProBoolean* option to specify if splitting surfaces are used while generating the draft geometry. If this option is set to the default value PRO_B_FALSE, then the element PRO_E_VPDD_SPLIT_SRFS_REFS becomes unavailable.
- PRO_E_VPDD_SPLIT_SRFS_REFS—Specifies a collection of splitting surface references. You can select upto two references that are either PRO_QUILT or PRO_DATUM_PLANE.
- PRO_E_VPDD_CONST_OR_VAR—Specifies a constant or variable draft. For a variable draft, there are variable angle attachment points. The values for this element, specified by the enumerated type *Pro_vpdd_const_var_type*, are as follows:
 - PRO_VPDD_CONST
 - PRO_VPDD_VAR

- **PRO_E_VPDD_ANGLES_AND_POINTS_ARR**—Specifies an array of elements of the type **PRO_E_VPDD_POINT_AND_ANGLES_CMP**. **PRO_E_VPDD_POINT_AND_ANGLES_CMP** is a compound element and specifies a collection of points and angles.
 - **PRO_E_STD_POINT_COLLECTION_APPL**—Specifies the selection references (curves, edges, or datum points) where the draft angles are defined.

Note:

- If **PRO_E_VPDD_CONST_OR_VAR** is set to **PRO_VPDD_CONST**, then the element **PRO_E_STD_POINT_COLLECTION_APPL** becomes unavailable and the element **PRO_E_VPDD_ANGLES_AND_POINTS_ARR** contains only a single element.
- A reference is valid when it belongs to or lies on a draft hinge that belongs to the same draft set.
- **PRO_E_VPDD_ANGLES_ARR**—Specifies an array of elements of the type **PRO_E_VPDD_ONE_ANGLE_CMP**. The size of the array is one plus the number of splitting references. **PRO_E_VPDD_ONE_ANGLE_CMP** is a compound element and specifies the draft angle for the active draft set.
 - **PRO_E_VPDD_ANG_VAL**—Specifies the value of the draft angle. This value lies in the range of -90 degrees through 90 degrees.

Creating a VPDD

Function Introduced:

- **ProFeatureCreate()**

Use the function **ProFeatureCreate()** to create a VPDD based on element tree input. For more information about **ProFeatureCreate()**, refer to the section “Overview of Feature Creation” of chapter “Element Trees: Principles of Feature Creation”.

Redefining a VPDD

Function Introduced:

- **ProFeatureRedefine()**

Use the function **ProFeatureRedefine()** to redefine a VPDD based on the changes made in the element tree. For more information about **ProFeatureRedefine()**, refer to the section “Feature Redefine” of chapter “Element Trees: Principles of Feature Creation”.

Accessing a VPDD

Function Introduced:

- **ProFeatureElemtreeExtract()**

Use the function **ProFeatureElemtreeExtract()** to create a feature element tree that describes the contents of a VPDD and to retrieve the element tree description of a Draft. For more information about **ProFeatureElemtreeExtract()**, refer to the section “Feature Inquiry” of chapter “Element Trees: Principles of Feature Creation”.

Element Trees: Round and Chamfer

This chapter describes how to create, redefine, and query round and chamfer features through element trees and element tree functions. The section Overview of Feature Creation of the chapter Element Trees: Principles of Feature Creation provides a necessary background information for this topic.

See the Part Modeling module in Pro/ENGINEER for further details on round and chamfer feature creation.

The following sections describe the procedure for accessing the chamfers and round features in detail.

Topic	Page
Round Feature	36 - 2
Auto Round Feature	36 - 21
Chamfer Feature	36 - 22
Corner Chamfer Feature	36 - 46

Round Feature

Feature Element Tree for Round Feature

The element tree for a round is documented in the header file *ProRound.h*, and has a simple structure. The following figure demonstrates the feature element tree structure:

Figure 36-1: Feature Element Tree for Round

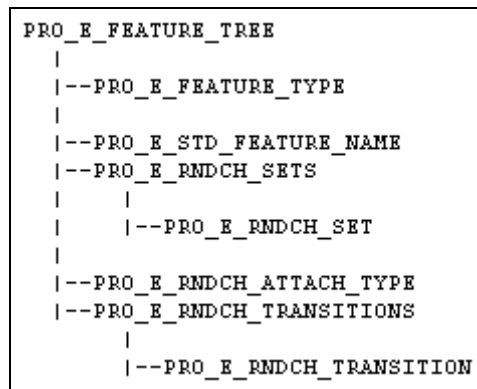


Figure 36-2: PRO_E_RNDCH_SETS

```

|--PRO_E_RNDCH_SETS
|
|   |--PRO_E_RNDCH_SET
|   |
|   |   |--PRO_E_RNDCH_SHAPE_OPTIONS
|   |   |--PRO_E_RNDCH_VARIABLE_RADIUS
|   |   |--PRO_E_RNDCH_COMPOUND_CONIC
|   |   |
|   |   |   |--PRO_E_RNDCH_CONIC_TYPE
|   |   |   |--PRO_E_RNDCH_CONIC_VALUE
|   |   |   |--PRO_E_RNDCH_CONIC_DEP_OPT
|   |
|   |   |--PRO_E_RNDCH_REFERENCES
|   |   |
|   |   |   |--PRO_E_RNDCH_REFERENCE_TYPE
|   |   |   |--PRO_E_STD_CURVE_COLLECTION_APPL
|   |   |   |--PRO_E_RNDCH_REFERENCE_SURFACE1
|   |   |   |--PRO_E_RNDCH_REFERENCE_SURFACE2
|   |   |   |--PRO_E_RNDCH_REFERENCE_EDGE1
|   |   |   |--PRO_E_RNDCH_REFERENCE_EDGE2
|   |   |   |--PRO_E_RNDCH_REPLACE_SURFACE
|   |
|   |   |--PRO_E_RNDCH_COMPOUND_SPINE
|   |   |
|   |   |   |--PRO_E_RNDCH_BALL_SPINE
|   |   |   |--PRO_E_STD_CURVE_COLLECTION_APPL
|   |
|   |   |--PRO_E_RNDCH_AUTO_CONTINUE
|   |   |--PRO_E_RNDCH_COMPOUND_EXT_OPTIONS
|   |   |
|   |   |   |--PRO_E_RNDCH_AUTO_BLEND
|   |   |   |--PRO_E_RNDCH_TERM_SURFACE
|   |
|   |   |--PRO_E_RNDCH_RADII
|   |   |
|   |   |   |--PRO_E_RNDCH_RADIUS
|   |
|   |   |--PRO_E_STD_CURVE_COLLECTION_APPL
|   |   |--PRO_E_RNDCH_AMBIGUITY

```

Element Trees:
Round and Chamfer

Figure 36-3: PRO_E_RNDCH_RADIUS

```
--PRO_E_RNDCH_RADIUS
|
|  |--PRO_E_STD_POINT_COLLECTION_APPL
|  |--PRO_E_RNDCH_LEG1
|  |
|  |  |--PRO_E_RNDCH_LEG_TYPE
|  |  |--PRO_E_RNDCH_LEG_VALUE
|  |  |--PRO_E_RNDCH_REFERENCE_EDGE
|  |  |--PRO_E_RNDCH_REFERENCE_POINT
|  |
|  |--PRO_E_RNDCH_LEG2
|  |
|  |  |--PRO_E_RNDCH_LEG_TYPE
|  |  |--PRO_E_RNDCH_LEG_VALUE
|  |  |--PRO_E_RNDCH_REFERENCE_EDGE
|  |  |--PRO_E_RNDCH_REFERENCE_POINT
```

Figure 36-4: PRO_E_RNDCH_TRANSITIONS

```
|--PRO_E_RNDCH_TRANSITIONS
|
|  |--PRO_E_RNDCH_TRANSITION
|  |
|  |  |--PRO_E_RNDCH_TRANS_TYPE
|  |  |--PRO_E_RNDCH_TRANS_CAP
|  |  |--PRO_E_RNDCH_TRANS_SPHERE_DATA
|  |  |
|  |  |  |--PRO_E_RNDCH_TRANS_RADIUS_OPTIONS
|  |  |  |--PRO_E_RNDCH_TRANS_SPHERE_RADIUS
|  |  |  |--PRO_E_RNDCH_TRANS_LEG1_OPTIONS
|  |  |  |--PRO_E_RNDCH_TRANS_LEG1_VALUE
|  |  |  |--PRO_E_RNDCH_TRANS_LEG2_OPTIONS
|  |  |  |--PRO_E_RNDCH_TRANS_LEG2_VALUE
|  |  |  |--PRO_E_RNDCH_TRANS_LEG3_OPTIONS
|  |  |  |--PRO_E_RNDCH_TRANS_LEG3_VALUE
|  |  |
|  |  |--PRO_E_RNDCH_TRANS_PATCH_DATA
|  |  |
|  |  |  |--PRO_E_RNDCH_TRANS_PATCH_REF_SURF
|  |  |  |--PRO_E_RNDCH_TRANS_PATCH_RAD_OPT
|  |  |  |--PRO_E_RNDCH_TRANS_ARC_RADIUS
|  |  |
|  |  |--PRO_E_RNDCH_TRANS_STOP_DATA
|  |  |
|  |  |  |--PRO_E_RNDCH_TRANS_STOP_REF_TYPE
|  |  |  |--PRO_E_RNDCH_TRANS_STOP_REFERENCE
```

The following list details special information about the elements in this tree:

- **PRO_E_FEATURE_TYPE**—Specifies feature type and should have the value as **PRO_FEAT_ROUND** only.
- **PRO_E_STD_FEATURE_NAME**—Specifies the name to the feature. This is an optional element. If not specified then a default name will be assigned internally to the feature.
- **PRO_E_RNDCH_SETS**—Specifies an array of **PRO_E_RNDCH_SET**.
- **PRO_E_RNDCH_ATTACH_TYPE**—Specifies the attachment type and has the following values:
 - **PRO_ROUND_ATTACHED**—Specifying this option, the created round feature consumes the model geometry.
 - **PRO_ROUND_UNATTACHED**
 - **PRO_ROUND_CAPPED_ENDS**
- **PRO_E_RNDCH_TRANSITIONS**—Specifies a set of transition **PRO_E_RNDCH_TRANSITION**.

Element Details of **PRO_E_RNDCH_SET** for Round

Each **PRO_E_RNDCH_SET** specifies a round set, which is a round piece (geometry) created as per the placement references and consists of the following elements:

- **PRO_E_RNDCH_SHAPE_OPTIONS**—Specifies the shape options and have the following values:
 - **PRO_ROUND_TYPE_CONSTANT**—Specifies a round piece having a constant radius.
 - **PRO_ROUND_TYPE_VARIABLE**—Specifies a round piece having multiple radii.
 - **PRO_ROUND_TYPE_FULL**—Specifies full round which replaces the selected surface.
 - **PRO_ROUND_TYPE_THROUGH_CURVE**—Allows the radius of the active round to be driven by the selected datum curve.
- **PRO_E_RNDCH_VARIABLE_RADIUS**—Specifies if the round is of a constant or variable type.
- **PRO_E_RNDCH_COMPOUND_CONIC**—Specifies if the round uses a conic section for the shape. It can be defined as:

- PRO_E_RNDCH_CONIC_TYPE—Specifies conic type and can have the following valid values
 - PRO_ROUND_CONIC_ENABLE
 - PRO_ROUND_CONIC_DISABLE
 - PRO_ROUND_CURV_CONTINUOUS—Used to specify curvature continuous rounds.
- PRO_E_RNDCH_CONIC_VALUE—Specifies conic value, that controls the sharpness of the conic shape and is required if PRO_E_RNDCH_CONIC_TYPE is equal to PRO_ROUND_CONIC_ENABLE.
- PRO_E_RNDCH_CONIC_DEP_OPT—Specifies independent options and is required if PRO_E_RNDCH_CONIC_TYPE is equal to PRO_ROUND_CONIC_ENABLE.

The values of *ProRoundConicIndependentType* are as follows:

- PRO_ROUND_CONIC_DEPENDENT
 - PRO_ROUND_CONIC_INDEPENDENT
- PRO_ROUND_CONIC_INDEPENDENT is the default type.
- PRO_E_RNDCH_REFERENCES—Specifies a set of valid references of the round feature.
 - PRO_E_RNDCH_COMPOUND_SPINE—This is another option for defining the shape of the round. Specifies the spine and has the following elements:
 - PRO_E_RNDCH_BALL_SPINE—Specifies rolling ball or normal to spine. Valid values of PRO_E_RNDCH_BALL_SPINE are:
 - PRO_ROUND_ROLLING_BALL—Specifies a round surface and is created by rolling a spherical ball along the surfaces.
 - PRO_ROUND_NORMAL_TO_SPINE—Specifies a round surface and is created by sweeping a conic or arc cross-section normal to a spine.
 - PRO_E_STD_CURVE_COLLECTION_APPL—Specifies the spine curve and is required if PRO_E_RNDCH_BALL_SPINE is equal to PRO_ROUND_NORMAL_TO_SPINE.

- **PRO_E_RNDCH_AUTO_CONTINUE**—Specifies whether surfaces will be extended to meet the designated round radius. The valid values are:
 - **PRO_ROUND_AUTO_CONT_DISABLE**—This is the default value.
 - **PRO_ROUND_AUTO_CONT_ENABLE**

This element is required if
PRO_E_RNDCH_REFERENCE_TYPE =
PRO_ROUND_REF_EDGE
- **PRO_E_RNDCH_COMPOUND_EXT_OPTIONS**—Specifies the external options. This is an optional element. It has the following elements:
 - **PRO_E_RNDCH_AUTO_BLEND**—Specifies auto blend.
 - **PRO_E_RNDCH_TERM_SURFACE**—Specifies the terminating surface.
- **PRO_E_RNDCH_RADII**—Specifies the radii, as an array of radius or **PRO_E_RNDCH_RADIUS** and is required if **PRO_E_RNDCH_SHAPE_OPTIONS** is not equal to **PRO_ROUND_TYPE_THROUGH_CURVE** and **PRO_E_RNDCH_SHAPE_OPTIONS** is not equal to **PRO_ROUND_TYPE_FULL**.
- **PRO_E_STD_CURVE_COLLECTION_APPL**—Specifies the curve collection and is required if **PRO_E_RNDCH_SHAPE_OPTIONS** is equal to **PRO_ROUND_TYPE_THROUGH_CURVE**.
- **PRO_E_RNDCH_AMBIGUITY**—Specifies round set ambiguity.

Note: Ambiguity occurs in round features when other placement locations exist for the round set. The ambiguity condition occurs when two surfaces intersect in multiple locations.

Element Details of **PRO_E_RNDCH_REFERENCES** for Round

Each **PRO_E_RNDCH_REFERENCES** specifies a set of valid references of the round feature and has the following elements:

- **PRO_E_RNDCH_REFERENCE_TYPE**—Specifies the reference types and valid values are:
 - **PRO_ROUND_REF_EDGE**—Specifies that the surfaces border the edge reference and form the rolling tangent attachment for the round.

- PRO_ROUND_REF_SURF_SURF—Specifies that the edges of the round remain tangent to the reference surfaces.
- PRO_ROUND_REF_EDGE_SURF—Specifies that the round remains tangent to the surface. The edge reference does not maintain tangency.
- PRO_ROUND_REF_EDGE_EDGE—Specifies that the surfaces bordering the edge reference form the rolling tangent attachment for the round.
- PRO_E_STD_CURVE_COLLECTION_APPL—Specifies the reference edges of the chain collection. It is required if PRO_E_RNDCH_REFERENCE_TYPE is equal to PRO_ROUND_REF_EDGE.
- PRO_E_RNDCH_REFERENCE_SURFACE1—Specifies the first reference surface and is required if PRO_E_RNDCH_REFERENCE_TYPE is equal to PRO_ROUND_REF_SURF_SURF.
- PRO_E_RNDCH_REFERENCE_SURFACE2—Specifies the second reference surface and is required if PRO_E_RNDCH_REFERENCE_TYPE is equal to PRO_ROUND_REF_SURF_SURF or PRO_E_RNDCH_REFERENCE_TYPE is equal to PRO_ROUND_REF_EDGE_SURF.
- PRO_E_RNDCH_REFERENCE_EDGE1—Specifies the first reference edge and is required if PRO_E_RNDCH_REFERENCE_TYPE is equal to PRO_ROUND_REF_EDGE_SURF or PRO_E_RNDCH_REFERENCE_TYPE is equal to PRO_ROUND_REF_EDGE_EDGE and PRO_E_RNDCH_SHAPE_OPTIONS is equal to PRO_ROUND_TYPE_FULL.
- PRO_E_RNDCH_REFERENCE_EDGE2—Specifies the second reference edge and is required if PRO_E_RNDCH_REFERENCE_TYPE is equal to PRO_ROUND_REF_EDGE_EDGE and PRO_E_RNDCH_SHAPE_OPTIONS is equal to PRO_ROUND_TYPE_FULL.
- PRO_E_RNDCH_REPLACE_SURFACE—Specifies the surface to be replaced and is required if PRO_E_RNDCH_REFERENCE_TYPE is equal to PRO_ROUND_REF_SURF_SURF and PRO_E_RNDCH_SHAPE_OPTIONS is equal to PRO_ROUND_TYPE_FULL.

Element Details of PRO_E_RNDCH_RADIUS for Round

Each PRO_E_RNDCH_RADIUS has the following elements:

- PRO_E_STD_POINT_COLLECTION_APPL—Specifies a point, which governs the value of the radius.
- PRO_E_RNDCH_LEG1—Specifies the leg1.
- PRO_E_RNDCH_LEG2—Specifies the leg2 and is required if PRO_E_RNDCH_CONIC_DEP_OPT is equal to PRO_ROUND_CONIC_INDEPENDENT.

Each PRO_E_RNDCH_LEG1 or PRO_E_RNDCH_LEG2 has the following elements

- PRO_E_RNDCH_LEG_TYPE—*ProRoundRadiusType* specifies leg type and is a mandatory element. It is of the following types:
 - PRO_ROUND_RADIUS_TYPE_VALUE
 - PRO_ROUND_RADIUS_THROUGH_POINT
- PRO_E_RNDCH_LEG_VALUE—Specifies leg value and is required if PRO_E_RNDCH_LEG_TYPE is equal to PRO_ROUND_RADIUS_TYPE_VALUE.
- PRO_E_RNDCH_REFERENCE_EDGE—Specifies reference edge having the value as PRO_E_EDGE and is required if PRO_E_RNDCH_LEG_TYPE is equal to PRO_ROUND_RADIUS_THROUGH_POINT.
- PRO_E_RNDCH_REFERENCE_POINT—Specifies reference point having the value as PRO_E_POINT and is required if PRO_E_RNDCH_LEG_TYPE is equal to PRO_ROUND_RADIUS_THROUGH_POINT.

Element Details of PRO_E_RNDCH_TRANSITION for Round

Each PRO_E_RNDCH_TRANSITION represents user-defined transitions for the entire round feature and consists of the following elements:

- PRO_E_RNDCH_TRANS_TYPE specifies the type of the transition type. Valid values are:
 - PRO_ROUND_TRANS_INTERSECT—Used to extend two or more overlapping round pieces towards each other until they merge forming a sharp boundary.
 - PRO_ROUND_TRANS_BLEND—Used to create a fillet surface between the round pieces using an edge reference.

- PRO_ROUND_TRANS_STOP—Used to terminate the round geometry at the specified datum point or datum plane.
- PRO_ROUND_TRANS_CONTINUE —Used to extend round geometry into two round pieces.
- PRO_ROUND_TRANS_SPHERE_CORNER—Used to create a round from the corner transition formed by three overlapping pieces by a spherical corner.
- PRO_ROUND_TRANS_PATCH—Used to create a patched surface at the location where three or four round pieces overlap at a corner.
- PRO_ROUND_TRANS_BLEND_3SRF—Used to create a triangular patch as a transition of three rounds.
- PRO_ROUND_TRANS_RBALL—Used to create a rolling ball transition for three or more rounds.
- PRO_ROUND_TRANS_STOP_0_SIDE —Used to terminate the round using geometry configured by Pro/ENGINEER.
- PRO_ROUND_TRANS_STOP_1_SIDE —Used to terminate the round using geometry configured by Pro/ENGINEER.
- PRO_ROUND_TRANS_STOP_2_SIDE —Used to terminate the round using geometry configured by Pro/ENGINEER.
- PRO_ROUND_TRANS_STOP_AT_REF—Used to terminate round geometry at the selected datum point or datum plane.
- PRO_ROUND_TRANS_STOP_FULL—Used to keep the stop transition close to the boundary of the removed surface.
- PRO_ROUND_TRANS_STOP_2_WE—Used to create a stop transition with maximum possible extension of the round by freezing one of round's references and changing the other reference at intersection with tangent edges.
- PRO_E_RNDCH_TRANS_CAP—Specifies the capping surface for round feature.

It has the following values:

- PRO_ROUND_CAPPING_SURF_DISABLE = PRO_B_FALSE,
- PRO_ROUND_CAPPING_SURF_ENABLE = PRO_B_TRUE

- PRO_E_RNDCH_TRANS_SPHERE_DATA—Specifies sphere data and consists of the following elements:
 - PRO_E_RNDCH_TRANS_RADIUS_OPTIONS—Specifies radius options and is a mandatory element.
 - PRO_E_RNDCH_TRANS_SPHERE_RADIUS—Specifies sphere radius and is required if PRO_E_RNDCH_TRANS_RADIUS_OPTIONS is equal to PRO_ROUND_TRANS_RADIUS_ENTER_VALUE.
 - PRO_E_RNDCH_TRANS_LEG1_OPTIONS—Specifies leg1 options and is a mandatory element.
 - PRO_E_RNDCH_TRANS_LEG1_VALUE—Specifies the value of leg1 and is required if PRO_E_RNDCH_TRANS_LEG1_OPTIONS is equal to PRO_ROUND_TRANS_RADIUS_ENTER_VALUE.
 - PRO_E_RNDCH_TRANS_LEG2_OPTIONS—Specifies leg2 options and is a mandatory element.
 - PRO_E_RNDCH_TRANS_LEG2_VALUE—Specifies the value of leg2 and is required if PRO_E_RNDCH_TRANS_LEG2_OPTIONS is equal to PRO_ROUND_TRANS_RADIUS_ENTER_VALUE.
 - PRO_E_RNDCH_TRANS_LEG3_OPTIONS—Specifies leg3 options and is a mandatory element.
 - PRO_E_RNDCH_TRANS_LEG3_VALUE—Specifies the value of leg3 and is required if PRO_E_RNDCH_TRANS_LEG3_OPTIONS is equal to PRO_ROUND_TRANS_RADIUS_ENTER_VALUE.
- PRO_E_RNDCH_TRANS_PATCH_DATA—Specifies the patch data and is required if PRO_E_RNDCH_TRANS_TYPE is equal to PRO_ROUND_TRANS_PATCH. It has the following elements:
 - PRO_E_RNDCH_TRANS_PATCH_REF_SURF—Specifies the arc surface, which indicates that a valid surface reference has been selected to place a fillet for the active patch transition. It should have the value as PRO_SURFACE only.
 - PRO_E_RNDCH_TRANS_PATCH_RAD_OPT—Specifies the arc radius options. It indicates the fillet radius for the active patch transition and has following options:
 - PRO_ROUND_TRANS_RADIUS_ENTER_VALUE—Specifies a new radius value.

- PRO_ROUND_TRANS_RADIUS_AUTOMATIC—Specifies the most recently used radius value.
- PRO_E_RNDCH_TRANS_ARC_RADIUS—Specifies the arc radius.
- PRO_E_RNDCH_TRANS_STOP_DATA—Specifies the capping surface. It has the following elements:
 - PRO_E_RNDCH_TRANS_STOP_REF_TYPE—Specifies the reference type. Valid values are:
 - PRO_ROUND_TRANS_REF_NO_REF
 - PRO_ROUND_TRANS_REF_GEOM
 - PRO_ROUND_TRANS_REF_PNTVTX
 - PRO_ROUND_TRANS_REF_DTMLPN
 - PRO_ROUND_TRANS_REF_ISOLINE
 - PRO_E_RNDCH_TRANS_STOP_REFERENCE—Specifies the references for the active stop at reference transition. It can either be PRO_SURFACE or PRO_POINT.

Creating a Round

Function Introduced:

- **ProFeatureCreate()**

Use the function **ProFeatureCreate()** to create a round based on element tree input. For more information about **ProFeatureCreate()**, refer to the section “Overview of Feature Creation” in the “Element Trees: Principles of Feature Creation” chapter.

Note: In Pro/ENGINEER Wildfire 2.0,

- Pro/TOOLKIT does not support the temporary geometry required for user-specified ambiguity and non-default transitions. Therefore, these elements cannot be used for creation of new rounds.
- If transitions are specified in the input element tree, a round feature with the default transition will be created.
- In case of ambiguous situation (where more than one valid solutions exist, e.g. for surface-surface round - surfaces having discontinuous edges of intersection), a round feature with default solution will be created.

Redefining a Round

Function Introduced:

- **ProFeatureRedefine()**

Use the function **ProFeatureRedefine()** to redefine a round based on the changes made in the element tree. For more information about **ProFeatureRedefine()**, refer to the section “Feature Redefine” in the “Element Trees: Principles of Feature Creation” chapter.

Note: In Pro/ENGINEER Wildfire 2.0,

- A round feature having default transition, can not be redefined to have any transition.
- A round feature having a Pro/ENGINEER user interface defined transition can be redefined to other type of transition, for example, from intersect type to spherical type. The input element tree must have a valid transition of the required type.

Accessing a Round

Function Introduced:

- **ProFeatureElemtreeExtract()**

Use the function **ProFeatureElemtreeExtract()** to create a feature element tree that describes the contents of a round and to retrieve the element tree description of a round. For more information about **ProFeatureElemtreeExtract()**, refer to the section “Feature Inquiry” in the “Element Trees: Principles of Feature Creation” chapter.

Example 1: Sample code for creation of a Round Feature

The example shows how to create an Edge Round feature. The user is prompted for three edges to be rounded.

```
/*----- Pro/Toolkit Includes -----*/
#include "ProToolkit.h"
#include "ProFeature.h"
#include "ProElemId.h"
#include "ProExtrude.h"
#include "ProModFeat.h"
#include "ProStdSection.h"
#include "ProElement.h"
#include "ProElemPath.h"
#include "ProFeatType.h"
```

```

#include "ProFeatForm.h"
#include "ProSelection.h"
#include "ProSection.h"

#include "ProSrfcollection.h"
#include "ProCrvcollection.h"
#include "ProRound.h"

static ProFileName message_file;

#define C_LOG(a) ProTKPrintf ("Status for %s is = %d\n", a, status ); \
ERROR_CHECK( a, "UgRoundCreate.c", status );

#define C_PRINT(a) ProTKPrintf ( "%s\n", a);

/*=====*\
FUNCTION : UserEdgeRoundCreate()
PURPOSE  : Function for creation of 3-Edge Round features.
\*=====*/
ProError UserEdgeRoundCreate()
{
    ProErrorlist          errors;
    ProMdl                model;
    ProSelection          model_sel;
    ProFeature            feature;
    ProFeatureCreateOptions opts[1];
    ProElempath           path;
    ProElempathItem       path_items[2];
    ProSection            section;
    ProAsmcomppath        comp_path;
    ProAsmcomppath        *p_comp_path = NULL;

    ProCollection collection;
    ProCrvcollinstr instr;
    ProModelitem model_item;
    ProReference reference;

    ProElement pro_e_feature_tree;
    ProElement pro_e_feature_type;
    ProElement pro_e_std_feature_name;
    ProElement pro_e_rndch_sets;
    ProElement pro_e_rndch_set;
    ProElement pro_e_rndch_dimensional_schema;
    ProElement pro_e_rndch_chamfer_shape;
    ProElement pro_e_rndch_shape_options;
    ProElement pro_e_rndch_variable_radius;
    ProElement pro_e_rndch_compound_conic;
    ProElement pro_e_rndch_conic_type;
    ProElement pro_e_rndch_conic_value;
    ProElement pro_e_rndch_conic_dep_opt;
    ProElement pro_e_rndch_references;

```

```

ProElement pro_e_rndch_reference_type;
ProElement pro_e_std_curve_collection_appl;
ProElement pro_e_rndch_compound_spine;
ProElement pro_e_rndch_ball_spine;
ProElement pro_e_rndch_compound_ext_options;
ProElement pro_e_rndch_auto_blend;
ProElement pro_e_rndch_radii;
ProElement pro_e_rndch_radius;
ProElement pro_e_rndch_leg1;
ProElement pro_e_rndch_leg_type;
ProElement pro_e_rndch_leg_value;
ProElement pro_e_rndch_leg2;
ProElement pro_e_rndch_attach_type;
ProElement pro_e_rndch_transitions;

ProName wide_string;
ProSelection *sketch_refs;
ProError status;
ProValueData value_data;
ProSelection * p_select;
int n_select;

ProStringToWstring ( message_file, "utilities.txt" );

status = ProMdlCurrentGet (&model);
if ( status != PRO_TK_NO_ERROR ) return ( status );

/*-----*\
   Populating root element PRO_E_FEATURE_TREE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TREE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TREE, &pro_e_feature_tree );
C_LOG( " ProElementAlloc " );

/*-----*\
   Populating element PRO_E_FEATURE_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TYPE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TYPE, &pro_e_feature_type );
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_feature_type, PRO_FEAT_ROUND );
C_LOG( " ProElementIntegerSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_feature_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
   Populating element PRO_E_STD_FEATURE_NAME
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_FEATURE_NAME *** " );
status = ProElementAlloc ( PRO_E_STD_FEATURE_NAME,

```

```

                                &pro_e_std_feature_name );
C_LOG( " ProElementAlloc " );
ProStringToWstring ( wide_string, "MY_ROUND" );
status = ProElementWstringSet ( pro_e_std_feature_name, wide_string );
C_LOG( " ProElementWstringSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_std_feature_name );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_RNDCH_SETS
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_SETS *** " );
status = ProElementAlloc ( PRO_E_RNDCH_SETS, &pro_e_rndch_sets );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_rndch_sets );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_RNDCH_SET
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_SET *** " );
status = ProElementAlloc ( PRO_E_RNDCH_SET, &pro_e_rndch_set );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_rndch_sets, NULL, pro_e_rndch_set
);
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_RNDCH_SHAPE_OPTIONS
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_SHAPE_OPTIONS *** " );
status = ProElementAlloc ( PRO_E_RNDCH_SHAPE_OPTIONS,
                            &pro_e_rndch_shape_options );
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_rndch_shape_options,
PRO_ROUND_TYPE_CONSTANT );
C_LOG( " ProElementIntegerSet " );
status = ProElemtreeElementAdd ( pro_e_rndch_set, NULL,
                                pro_e_rndch_shape_options );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_RNDCH_COMPOUND_CONIC
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_COMPOUND_CONIC *** " );
status = ProElementAlloc ( PRO_E_RNDCH_COMPOUND_CONIC,
                            &pro_e_rndch_compound_conic );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_rndch_set, NULL,

```



```

                                pro_e_rndch_compound_conic );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_RNDCH_CONIC_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_CONIC_TYPE *** " );
status = ProElementAlloc ( PRO_E_RNDCH_CONIC_TYPE,
                            &pro_e_rndch_conic_type );
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_rndch_conic_type,
PRO_ROUND_CONIC_DISABLE );
C_LOG( " ProElementIntegerSet " );
status = ProElemtreeElementAdd ( pro_e_rndch_compound_conic, NULL,
                                pro_e_rndch_conic_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_RNDCH_REFERENCES
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_REFERENCES *** " );
status = ProElementAlloc ( PRO_E_RNDCH_REFERENCES,
                            &pro_e_rndch_references );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_rndch_set, NULL,
                                pro_e_rndch_references );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_RNDCH_REFERENCE_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_REFERENCE_TYPE *** " );
status = ProElementAlloc ( PRO_E_RNDCH_REFERENCE_TYPE,
                            &pro_e_rndch_reference_type );
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_rndch_reference_type,
PRO_ROUND_REF_EDGE );
C_LOG( " ProElementIntegerSet " );
status = ProElemtreeElementAdd ( pro_e_rndch_references, NULL,
                                pro_e_rndch_reference_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_CURVE_COLLECTION_APPL
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_CURVE_COLLECTION_APPL *** "
);
status = ProElementAlloc ( PRO_E_STD_CURVE_COLLECTION_APPL,
                            &pro_e_std_curve_collection_appl );

C_LOG( " ProElementAlloc " );

```

```

status = ProCrvcollectionAlloc ( &collection );
C_LOG( " ProCrvcollectionAlloc" );
status = ProCrvcollinstrAlloc ( PRO_CURVCOLL_ADD_ONE_INSTR, &instr );
C_LOG( " ProCrvcollinstrAlloc" );

{
    ProSelection *sel_array;
    int n_sel_array;

    ProTKPrintf(" Please select 1st edge for ROUND \n");
    status = ProMessageDisplay ( message_file, "USER Select reference." );
    C_LOG( " ProMessageDisplay" );
    status = ProSelect ( "edge,curve", 1, NULL, NULL, NULL, NULL,
        &sel_array, &n_sel_array );
    C_LOG( " ProSelect" );

    if ( n_sel_array <= 0 )
        return PRO_TK_GENERAL_ERROR;

    status = ProSelectionToReference ( sel_array[0], &reference );
    C_LOG( " ProSelectionToReference" );
}
status = ProCrvcollinstrReferenceAdd ( instr, reference );
C_LOG( " ProCrvcollinstrReferenceAdd" );
status = ProCrvcollectionInstructionAdd ( collection, instr );
C_LOG( " ProCrvcollectionInstructionAdd" );
status = ProCrvcollinstrAlloc ( PRO_CURVCOLL_ADD_ONE_INSTR, &instr );
C_LOG( " ProCrvcollinstrAlloc" );

{
    ProSelection *sel_array;
    int n_sel_array;

    ProTKPrintf(" Please select 2nd edge for ROUND \n");
    status = ProMessageDisplay ( message_file, "USER Select reference." );
    C_LOG( " ProMessageDisplay" );
    status = ProSelect ( "edge,curve", 1, NULL, NULL, NULL, NULL,
        &sel_array, &n_sel_array );
    C_LOG( " ProSelect" );

    if ( n_sel_array <= 0 )
        return PRO_TK_GENERAL_ERROR;

    status = ProSelectionToReference ( sel_array[0], &reference );
    C_LOG( " ProSelectionToReference" );
}
status = ProCrvcollinstrReferenceAdd ( instr, reference );
C_LOG( " ProCrvcollinstrReferenceAdd" );
status = ProCrvcollectionInstructionAdd ( collection, instr );
C_LOG( " ProCrvcollectionInstructionAdd" );
status = ProCrvcollinstrAlloc ( PRO_CURVCOLL_ADD_ONE_INSTR, &instr );

```

```

C_LOG( " ProCrvcollinstrAlloc" );
{
    ProSelection *sel_array;
    int n_sel_array;

    ProTKPrintf(" Please select 3rd edge for ROUND \n");
    status = ProMessageDisplay ( message_file, "USER Select reference." );
    C_LOG( " ProMessageDisplay" );
    status = ProSelect ( "edge,curve", 1, NULL, NULL, NULL, NULL,
        &sel_array, &n_sel_array );
    C_LOG( " ProSelect" );

    if ( n_sel_array <= 0 )
        return PRO_TK_GENERAL_ERROR;

    status = ProSelectionToReference ( sel_array[0], &reference );
    C_LOG( " ProSelectionToReference" );
}
status = ProCrvcollinstrReferenceAdd ( instr, reference );
C_LOG( " ProCrvcollinstrReferenceAdd" );
status = ProCrvcollectionInstructionAdd ( collection, instr );
C_LOG( " ProCrvcollectionInstructionAdd" );
status = ProCrvcollinstrAlloc ( PRO_CURVCOLL_ADD_TANGENT_INSTR, &instr
);
C_LOG( " ProCrvcollinstrAlloc" );
status = ProCrvcollectionInstructionAdd ( collection, instr );
C_LOG( " ProCrvcollectionInstructionAdd" );
status = ProElementCollectionSet( pro_e_std_curve_collection_appl,
collection );
C_LOG( " ProElementCollectionSet " );
status = ProElemtreeElementAdd ( pro_e_rndch_references, NULL,
pro_e_std_curve_collection_appl );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_RNDCH_RADII
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_RADII *** " );
status = ProElementAlloc ( PRO_E_RNDCH_RADII, &pro_e_rndch_radii );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_rndch_set, NULL,
    pro_e_rndch_radii );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_RNDCH_RADIUS
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_RADIUS *** " );
status = ProElementAlloc ( PRO_E_RNDCH_RADIUS, &pro_e_rndch_radius );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_rndch_radii, NULL,

```

```

                                pro_e_rndch_radius );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_RNDCH_LEG1
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_LEG1 *** " );
status = ProElementAlloc ( PRO_E_RNDCH_LEG1, &pro_e_rndch_leg1 );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_rndch_radius, NULL,
                                pro_e_rndch_leg1 );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_RNDCH_LEG_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_LEG_TYPE *** " );
status = ProElementAlloc ( PRO_E_RNDCH_LEG_TYPE, &pro_e_rndch_leg_type
);
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_rndch_leg_type,
PRO_ROUND_RADIUS_TYPE_VALUE );
C_LOG( " ProElementIntegerSet " );
status = ProElemtreeElementAdd ( pro_e_rndch_leg1, NULL,
                                pro_e_rndch_leg_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_RNDCH_LEG_VALUE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_LEG_VALUE *** " );
status = ProElementAlloc ( PRO_E_RNDCH_LEG_VALUE, &pro_e_rndch_leg_value
);
C_LOG( " ProElementAlloc " );
status = ProElementDoubleSet ( pro_e_rndch_leg_value, 50.000000 );
C_LOG( " ProElementDoubleSet " );
status = ProElemtreeElementAdd ( pro_e_rndch_leg1, NULL,
                                pro_e_rndch_leg_value );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_RNDCH_ATTACH_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_ATTACH_TYPE *** " );
status = ProElementAlloc ( PRO_E_RNDCH_ATTACH_TYPE,
                            &pro_e_rndch_attach_type );
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_rndch_attach_type,
PRO_ROUND_ATTACHED );
C_LOG( " ProElementIntegerSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,

```

```

                                pro_e_rndch_attach_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Creating feature in the current model.
\*-----*/
status = ProMdlToModelitem( model, &model_item );
status = ProSelectionAlloc (p_comp_path, &model_item, &model_sel);

opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMS;
status = ProFeatureCreate (model_sel, pro_e_feature_tree, opts, 1,
&feature, &errors);
C_LOG (" ProFeatureCreate");

/*-----*\
    Free up the allocated memory.
\*-----*/
status = ProElementFree (&pro_e_feature_tree );
C_LOG (" ProElementFree");

return (status);
}

```

Auto Round Feature

The Auto Round feature enables you to create round geometry of a constant radius on solid geometry or on a quilt of a part or assembly. The Auto Round Feature creates Round features called Auto-Round Members (ARMs) that are represented in the Model Tree as subnodes of the Auto Round feature, or as groups of individual independent round features. Refer to the Pro/ENGINEER Online Help for more details about this feature type.

Pro/TOOLKIT does not provide access to the Auto Round feature via an element tree for the Wildfire 4.0 release. Some Pro/TOOLKIT functionality such as feature deletion, suppression and redefinition are not supported for individual ARMs within an Auto Round feature.

Functions Introduced:

- **ProRoundIsAutoRoundMember()**

The function **ProRoundIsAutoRoundMember()** identifies if the specified round feature is a member of the Auto Round feature.

Chamfer Feature

Feature Element Tree for Chamfer Feature

The element tree for a chamfer is documented in the header file *ProChamfer.h*, and has a simple structure. The following figure demonstrates the feature element tree structure:

Figure 36-5: Feature Element Tree for Chamfer

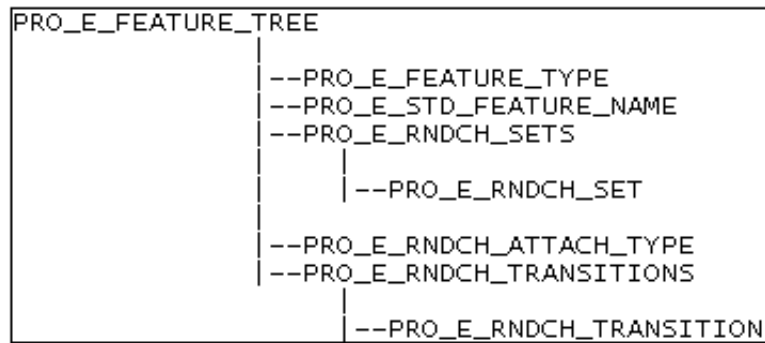
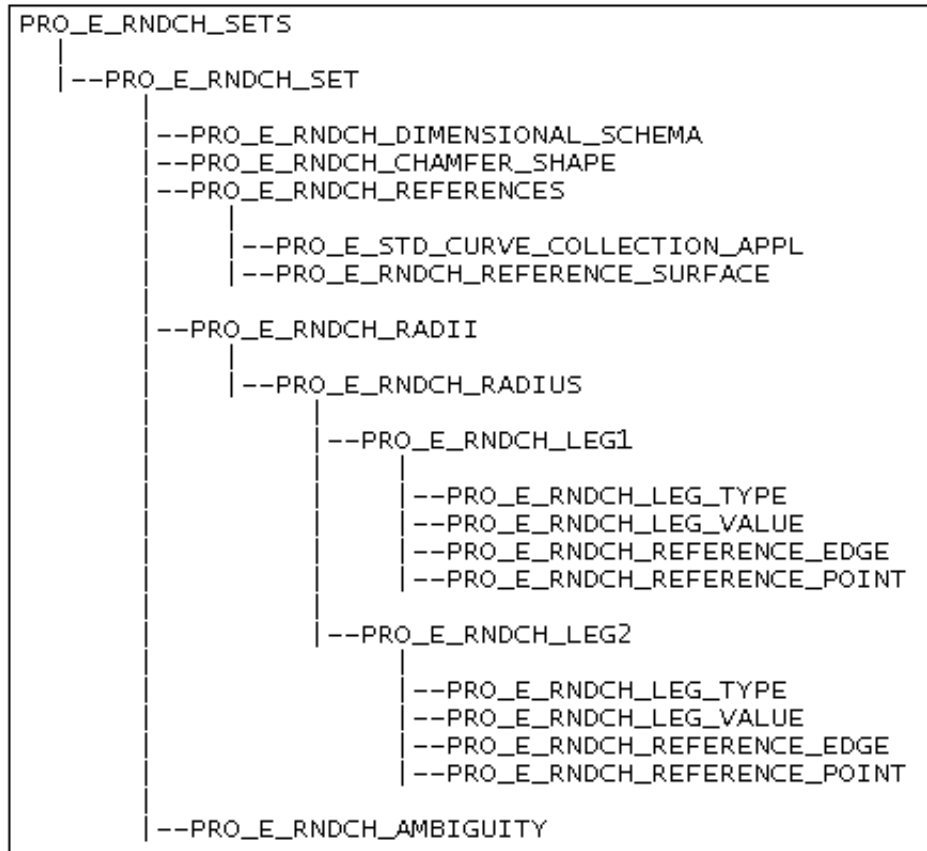
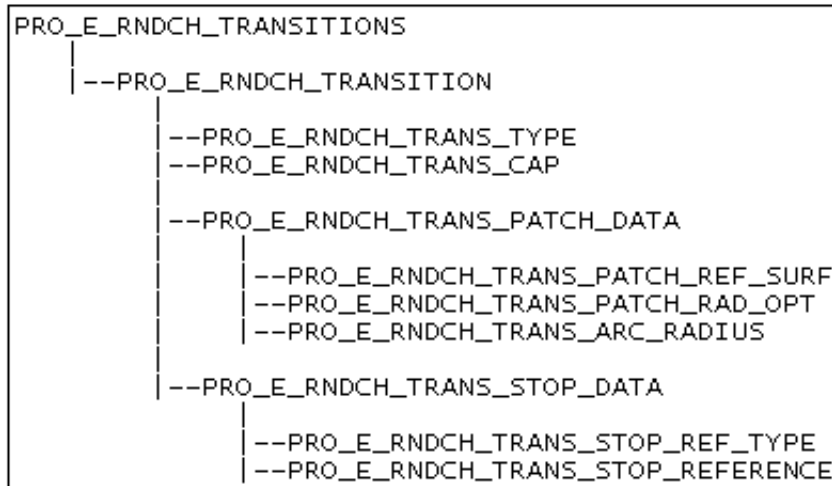


Figure 36-6: PRO_E_RNDCH_SETS



Element Trees:
Round and Chamfer

Figure 36-7: PRO_E_RNDCH_TRANSITIONS



The following list details special information about some of the elements in this tree:

- PRO_E_FEATURE_TYPE—Specifies feature type and has the value of PRO_FEAT_CHAMFER.
- PRO_E_STD_FEATURE_NAME—Specifies the name of the feature.
- PRO_E_RNDCH_SETS—Specifies an array of PRO_E_RNDCH_SET.
- PRO_E_RNDCH_ATTACH_TYPE—Specifies the attachment type and has the following values:
 - PRO_ROUND_ATTACHED
 - PRO_ROUND_UNATTACHED
 - PRO_ROUND_CAPPED_ENDS
- PRO_E_RNDCH_TRANSITIONS—Specifies a set of transition PRO_E_RNDCH_TRANSITION.

Element Details of PRO_E_RNDCH_SET for Chamfer

Each PRO_E_RNDCH_SET specifies a chamfer set for the chamfer feature and must have the following elements:

- PRO_E_RNDCH_DIMENSIONAL_SCHEMA—Specifies the type of chamfer or the dimensional schema PRO_E_RNDCH_DIMENSIONAL_SCHEMA. The different types of PRO_E_RNDCH_DIMENSIONAL_SCHEMA is as follows:
 - PRO_CHM_45_X_D—Specifies a chamfer that is at an angle of 45 degrees to both surfaces and at a distance D from the edge along each surface.
 - PRO_CHM_D_X_D—Specifies a chamfer that is at a distance D from the edge along each surface. This is the default type.
 - PRO_CHM_D1_X_D2—Specifies a chamfer at a distance D1 from the selected edge along one surface and at a distance D2 from the selected edge along the other surface.
 - PRO_CHM_ANG_X_D—Specifies a chamfer at a distance D from the selected edge along one adjacent surface, at a specified angle to that surface.
 - PRO_CHM_O_X_O—Provides direct control of the surface offset distances.
 - PRO_CHM_O1_X_O2—Provides direct control of the surface offset distances.

Note: For Surf-Surf chamfer, the available schemes are:

- Offset Surface method—PRO_CHM_O_X_O, PRO_CHM_O1_X_O2, PRO_CHM_D_X_D, PRO_CHM_D1_X_D2 are available.
- Tangent Dist method—PRO_CHM_D_X_D, PRO_CHM_D1_X_D2, PRO_CHM_45_X_D, PRO_CHM_ANG_X_D are available.

For Surface-to-Edge chamfer the available schemes are:

- Offset Surface: PRO_CHM_O_X_O, PRO_CHM_O1_X_O2.
- Tangent Distance: PRO_CHM_D_X_D, PRO_CHM_D1_X_D2.

These schemes are applicable for constant angle planes or constant 90 degree surfaces and are also available if all members of the edge chain are formed by exactly 2 planes or exactly 2 surfaces at 90 degree, as in the ends of a cylinder.

- PRO_E_RNDCH_CHAMFER_SHAPE—Specifies the shape of the chamfer feature. PRO_E_RNDCH_CHAMFER_SHAPE has the following valid values:
 - PRO_CHM_TANGENT_LEGS—Specifies the chamfer distance between vectors that are tangent to the neighboring surface of the reference edge.
 - PRO_CHM_OFFSET_SURFACE—Specifies the offset surfaces.
- PRO_E_RNDCH_REFERENCES—Specifies a set of valid references of the chamfer feature and has the following elements:
 - PRO_E_STD_CURVE_COLLECTION_APPL—Specifies reference edges and is a required element for edge chamfer.
 - PRO_E_RNDCH_REFERENCE_SURFACE—Specifies reference surfaces and is required if either PRO_E_RNDCH_DIMENSIONAL_SCHEMA is equal to PRO_CHM_D1_X_D2 or PRO_E_RNDCH_DIMENSIONAL_SCHEMA is equal to PRO_CHM_ANG_X_D.
- PRO_E_RNDCH_COMPOUND_EXT_OPTIONS—Specifies the external options. This is an optional element and has the following elements:
 - PRO_E_RNDCH_AUTO_BLEND—Specifies the auto blend.
 - PRO_E_RNDCH_TERM_SURFACE—Specifies terminating surface.
- PRO_E_RNDCH_RADII—Specifies an array of radius PRO_E_RNDCH_RADIUS.
- PRO_E_RNDCH_AMBIGUITY—Specifies the ambiguity in the chamfer set.

Note: The chamfer set can contain ambiguity if the chamfer set contains chamfer pieces that co-exist and can be placed in various locations in the selected references and in part geometry.

Element Details of PRO_E_RNDCH_RADIUS for Chamfer

Each PRO_E_RNDCH_RADIUS has the following elements:

- PRO_E_RNDCH_LEG1—Specifies leg1 of the chamfer feature.

- PRO_E_RNDCH_LEG2—Specifies leg2 of the chamfer feature and is a required element if either PRO_E_RNDCH_DIMENSIONAL_SCHEMA is equal to PRO_CHM_D1_X_D2 or PRO_E_RNDCH_DIMENSIONAL_SCHEMA is equal to PRO_CHM_ANG_X_D.

Each PRO_E_RNDCH_LEG1 or PRO_E_RNDCH_LEG2 has the following elements:

- PRO_E_RNDCH_LEG_TYPE—Specifies leg type and is a mandatory element. It is of the following types:
 - PRO_ROUND_RADIUS_TYPE_VALUE
 - PRO_ROUND_RADIUS_THROUGH_POINT

The definition of *ProRoundRadiusType* is as follows:

- PRO_E_RNDCH_LEG_VALUE—Specifies leg value and is required if PRO_E_RNDCH_LEG_TYPE is equal to PRO_ROUND_RADIUS_TYPE_VALUE.
- PRO_E_RNDCH_REFERENCE_EDGE—Specifies reference edge and is required if PRO_E_RNDCH_LEG_TYPE is equal to PRO_ROUND_RADIUS_THROUGH_POINT.
- PRO_E_RNDCH_REFERENCE_POINT—Specifies reference point and is required if PRO_E_RNDCH_LEG_TYPE is equal to PRO_ROUND_RADIUS_THROUGH_POINT.

Element Details of PRO_E_RNDCH_TRANSITION for Chamfer

Each PRO_E_RNDCH_TRANSITION represents user-defined transitions for the entire chamfer feature and consists of the following elements:

- PRO_E_RNDCH_TRANS_TYPE—Specifies the type of the transition type. Valid values are:
 - PRO_ROUND_TRANS_INTERSECT—Used to extend two or more overlapping chamfer pieces towards each other until they merge forming a sharp boundary.
 - PRO_ROUND_TRANS_BLEND—Used to create a fillet surface between the chamfer pieces using an edge reference.
 - PRO_ROUND_TRANS_STOP—Used to terminate the chamfer geometry at the specified datum point or datum plane.

- PRO_ROUND_TRANS_CONTINUE—Used to extend chamfer geometry into two chamfer pieces.
- PRO_ROUND_TRANS_PATCH—Used to create a patched surface at the location where three or four chamfer pieces overlap.
- PRO_ROUND_TRANS_BLEND_3SRF—Used to create a triangular patch as a transition of three chamfers.
- PRO_ROUND_TRANS_PLANE_CORNER—Used to chamfer the corner transition formed by three overlapping chamfer pieces with a plane.
- PRO_ROUND_TRANS_RBALL—Used to create a rolling ball transition for three or more chamfers.
- PRO_ROUND_TRANS_STOP_0_SIDE—Used to terminate the chamfer using geometry configured by Pro/ENGINEER.
- PRO_ROUND_TRANS_STOP_1_SIDE—Used to terminate the chamfer using geometry configured by Pro/ENGINEER.
- PRO_ROUND_TRANS_STOP_2_SIDE—Used to terminate the chamfer using geometry configured by Pro/ENGINEER.
- PRO_ROUND_TRANS_STOP_AT_REF—Used to terminate the chamfer geometry at the selected datum point or datum plane.

Note: Only some of the transition types listed above are available for a given context.

- PRO_E_RNDCH_TRANS_CAP—Specifies the capping surface for chamfer pieces of the chamfer feature. It has the following values:
 - PRO_ROUND_CAPPING_SURF_DISABLE = PRO_B_FALSE.
 - PRO_ROUND_CAPPING_SURF_ENABLE = PRO_B_TRUE.
- PRO_E_RNDCH_TRANS_PATCH_DATA—Specifies the patch data and is required if PRO_E_RNDCH_TRANS_TYPE is equal to PRO_ROUND_TRANS_PATCH. It has the following elements:
 - PRO_E_RNDCH_TRANS_PATCH_REF_SURF—Specifies the arc surface, which indicates that a valid surface reference has been selected to place a fillet for the active patch transition. It should have the value as PRO_SURFACE only.

- PRO_E_RNDCH_TRANS_PATCH_RAD_OPT—Specifies the arc radius options. It indicates the fillet radius for the active Patch transition and has following options:
 - PRO_ROUND_TRANS_RADIUS_ENTER_VALUE—Specifies a new radius value.
 - PRO_ROUND_TRANS_RADIUS_AUTOMATIC—Specifies the most recently used value.
- PRO_E_RNDCH_TRANS_ARC_RADIUS—Specifies the arc radius.
- PRO_E_RNDCH_TRANS_STOP_DATA—Specifies the capping surface. It has the following elements:
 - PRO_E_RNDCH_TRANS_STOP_REF_TYPE—Specifies the reference type. Valid values are:
 - PRO_ROUND_TRANS_REF_NO_REF
 - PRO_ROUND_TRANS_REF_GEOM
 - PRO_ROUND_TRANS_REF_PNTVTX
 - PRO_ROUND_TRANS_REF_DTMLN
 - PRO_ROUND_TRANS_REF_ISOLINE
 - PRO_E_RNDCH_TRANS_STOP_REFERENCE—Specifies the references for the active stop at reference transition. It can either be PRO_SURFACE or PRO_POINT.

Creating a Chamfer

Function Introduced:

- **ProFeatureCreate()**

Use the function **ProFeatureCreate()** to create a Chamfer based on element tree input. For more information about **ProFeatureCreate()**, refer to the section “Overview of Feature Creation” of chapter “Element Trees: Principles of Feature Creation”.

Note: In Pro/ENGINEER Wildfire 2.0,

- 1) Pro/TOOLKIT does not support the temporary geometry required for user-specified ambiguity and non-default transitions. Therefore, these elements cannot be used for creation of new chamfers.
- 2) If transitions are specified in the input element tree, a chamfer feature with the default transition will be created.
- 3) In case of ambiguous situation (where more than one valid solutions exist, e.g. for surface-surface chamfer - surfaces having discontinuous edges of intersection), a chamfer feature with default solution will be created.

Redefining a Chamfer

Function Introduced:

- **ProFeatureRedefine()**

Use the function **ProFeatureRedefine()** to redefine a Chamfer based on the changes made in the element tree. For more information about **ProFeatureRedefine()**, refer to the section “Feature Redefine” of chapter “Element Trees: Principles of Feature Creation”.

Note: In Pro/ENGINEER Wildfire 2.0,

- 1) A chamfer feature having default transition, can not be redefined to have any transition.
- 2) A chamfer feature having a Pro/ENGINEER user interface defined transition can be redefined to other type of transition, for example, from intersect type to corner type. The input element tree must have a valid transition of the required type.

Accessing a Chamfer

Function Introduced:

- **ProFeatureElemtreeExtract()**

Use the function **ProFeatureElemtreeExtract()** to create a feature element tree that describes the contents of a Chamfer and to retrieve the element tree description of a Chamfer. For more information about **ProFeatureElemtreeExtract()**, refer to the section “Feature Inquiry” of chapter “Element Trees: Principles of Feature Creation”.

Example 2: Sample code for creation of a Chamfer Feature

This example shows how to create a Chamfer feature. It includes all possible element assignments. By following the instructions for the feature you want to create, it should be possible to remove element settings not appropriate for your use.

```
/*----- Pro/Toolkit Includes -----*/
#include "ProToolkit.h"
#include "ProFeature.h"
#include "ProElemId.h"
#include "ProExtrude.h"
#include "ProModFeat.h"
#include "ProStdSection.h"
#include "ProElement.h"
#include "ProElempath.h"
#include "ProFeatType.h"
#include "ProFeatForm.h"
#include "ProSelection.h"
#include "ProSection.h"
#include "ProModelitem.h"

#include "ProSrfcollection.h"
#include "ProCrvcollection.h"
#include "ProRound.h"
#include "ProChamfer.h"

static ProFileName message_file;

#define C_LOG(a) ProTKPrintf ("Status for %s is = %d\n", a, status ); \
                ERROR_CHECK( a, "UgChamferCreate.c", status );

#define C_PRINT(a) ProTKPrintf ( "%s\n", a);

/*=====*\
FUNCTION : UserChamferTemplate()
PURPOSE  : Template function for creation of Chamfer features.
\*=====*/
ProError UserChamferTemplate()
{
    ProErrorlist      errors;
    ProMdl            model;
    ProSelection       model_sel;
    ProFeature         feature;
    ProFeatureCreateOptions opts[1];
    ProElempath        path;
    ProElempathItem    path_items[2];
    ProSection         section;
    ProAsmcomppath     comp_path;
    ProAsmcomppath     *p_comp_path = NULL;
```

```

ProCollection collection;
ProCrvcollinstr instr;
ProModelitem model_item;
ProReference reference;
ProName wide_string;

ProBoolean is_interactive = PRO_B_TRUE;

ProElement pro_e_feature_tree;
ProElement pro_e_feature_type;
ProElement pro_e_std_feature_name;
ProElement pro_e_rndch_sets;
ProElement pro_e_rndch_set;
ProElement pro_e_rndch_dimensional_schema;
ProElement pro_e_rndch_chamfer_shape;
ProElement pro_e_rndch_shape_options;
ProElement pro_e_rndch_variable_radius;
ProElement pro_e_rndch_compound_conic;
ProElement pro_e_rndch_conic_type;
ProElement pro_e_rndch_conic_value;
ProElement pro_e_rndch_conic_dep_opt;
ProElement pro_e_rndch_references;
ProElement pro_e_rndch_reference_type;
ProElement pro_e_std_curve_collection_appl;
ProElement pro_e_rndch_compound_spine;
ProElement pro_e_rndch_ball_spine;
ProElement pro_e_rndch_compound_ext_options;
ProElement pro_e_rndch_auto_blend;
ProElement pro_e_rndch_radii;
ProElement pro_e_rndch_radius;
ProElement pro_e_rndch_leg1;
ProElement pro_e_rndch_leg_type;
ProElement pro_e_rndch_leg_value;
ProElement pro_e_rndch_leg2;
ProElement pro_e_rndch_attach_type;
ProElement pro_e_rndch_transitions;

ProStringToWstring ( message_file, "utilities.txt" );

status = ProMdlCurrentGet (&model);
if ( status != PRO_TK_NO_ERROR ) return ( status );

/*-----*\
   Populating root element PRO_E_FEATURE_TREE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TREE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TREE, &pro_e_feature_tree );
C_LOG( " ProElementAlloc " );

/*-----*\
   Populating element PRO_E_FEATURE_TYPE

```



```

\*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TYPE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TYPE, &pro_e_feature_type );
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_feature_type, PRO_FEAT_CHAMFER );
C_LOG( " ProElementIntegerSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_feature_type );
C_LOG( " ProElemtreeElementAdd" );
\*-----*\
    Populating element PRO_E_STD_FEATURE_NAME
    - Optional
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_FEATURE_NAME *** " );
status = ProElementAlloc ( PRO_E_STD_FEATURE_NAME,
                            &pro_e_std_feature_name );
C_LOG( " ProElementAlloc " );
ProStringToWstring ( wide_string, "CHAMFER_ID_XXX" );
status = ProElementWstringSet ( pro_e_std_feature_name, wide_string );
C_LOG( " ProElementWstringSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_std_feature_name );
C_LOG( " ProElemtreeElementAdd" );

\*-----*\
    Populating element PRO_E_RNDCH_SETS
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_SETS *** " );
status = ProElementAlloc ( PRO_E_RNDCH_SETS, &pro_e_rndch_sets );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_rndch_sets );
C_LOG( " ProElemtreeElementAdd" );

\*-----*\
    Populating element PRO_E_RNDCH_SET
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_SET *** " );
status = ProElementAlloc ( PRO_E_RNDCH_SET, &pro_e_rndch_set );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_rndch_sets, NULL, pro_e_rndch_set
                                );
C_LOG( " ProElemtreeElementAdd" );

\*-----*\
    Populating element PRO_E_RNDCH_DIMENSIONAL_SCHEMA
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_DIMENSIONAL_SCHEMA *** " );
status = ProElementAlloc ( PRO_E_RNDCH_DIMENSIONAL_SCHEMA,
                            &pro_e_rndch_dimensional_schema );
C_LOG( " ProElementAlloc " );

```

```

status = ProElementIntegerSet ( pro_e_rndch_dimensional_schema,
                                PRO_CHM_D_X_D );
C_LOG( " ProElementIntegerSet" );
status = ProElemtreeElementAdd ( pro_e_rndch_set, NULL,
                                pro_e_rndch_dimensional_schema );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_RNDCH_CHAMFER_SHAPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_CHAMFER_SHAPE *** " );
status = ProElementAlloc ( PRO_E_RNDCH_CHAMFER_SHAPE,
                            &pro_e_rndch_chamfer_shape );
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_rndch_chamfer_shape,
                                PRO_CHM_OFFSET_SURFACE );
C_LOG( " ProElementIntegerSet" );
status = ProElemtreeElementAdd ( pro_e_rndch_set, NULL,
                                pro_e_rndch_chamfer_shape );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Following elements are "not required" for Chamfer , but are
    "required" for the Round Feature - but a copy of the chamfer element
    tree extracted from a chamfer feature will contain these elements
    0- PRO_E_RNDCH_SHAPE_OPTIONS
    1- PRO_E_RNDCH_VARIABLE_RADIUS :: ProRound.h mentions it "not used"
    2- PRO_E_RNDCH_COMPOUND_CONIC
    3- PRO_E_RNDCH_CONIC_TYPE
    4- PRO_E_RNDCH_CONIC_VALUE
    5- PRO_E_RNDCH_CONIC_DEP_OPT
\*-----*/

/*-----*\
    Populating element PRO_E_RNDCH_REFERENCES
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_REFERENCES *** " );
status = ProElementAlloc ( PRO_E_RNDCH_REFERENCES,
                            &pro_e_rndch_references );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_rndch_set, NULL,
                                pro_e_rndch_references );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_RNDCH_REFERENCE_TYPE
    This element is "not required" for Chamfer , but is
    "required" for the Round Feature - but a copy of the chamfer element
    tree extracted from a chamfer feature will contain these elements
\*-----*/
/*

```

```

C_PRINT( " *** Processing Element PRO_E_RNDCH_REFERENCE_TYPE *** " );
status = ProElementAlloc ( PRO_E_RNDCH_REFERENCE_TYPE,
                          &pro_e_rndch_reference_type );
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_rndch_reference_type,
                              PRO_ROUND_REF_EDGE ); [ default ]
C_LOG( " ProElementIntegerSet" );
status = ProElemtreeElementAdd ( pro_e_rndch_references, NULL,
                              pro_e_rndch_reference_type );
C_LOG( " ProElemtreeElementAdd" );
*/

/*-----*\
   Populating element PRO_E_STD_CURVE_COLLECTION_APPL
   - One by one curve collection for edges to be chamfered
   \*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_CURVE_COLLECTION_APPL *** "
        );
status = ProElementAlloc ( PRO_E_STD_CURVE_COLLECTION_APPL,
                          &pro_e_std_curve_collection_appl );
C_LOG( " ProElementAlloc " );
status = ProCrvcollectionAlloc ( &collection );
C_LOG( " ProCrvcollectionAlloc" );
status = ProCrvcollinstrAlloc ( PRO_CURV_COLL_ADD_ONE_INSTR, &instr );
C_LOG( " ProCrvcollinstrAlloc" );
if ( is_interactive == PRO_B_TRUE )
{
    ProSelection *sel_array;
    int n_sel_array;

    ProTKPrintf( " Please select edge for CHAMFER \n" );
    status = ProMessageDisplay ( message_file, "USER Select reference." );
    C_LOG( " ProMessageDisplay" );
    status = ProSelect ( "edge,curve", 1, NULL, NULL, NULL, NULL,
                      &sel_array, &n_sel_array );
    C_LOG( " ProSelect" );

    if ( n_sel_array <= 0 )
        return PRO_TK_GENERAL_ERROR;

    status = ProSelectionToReference ( sel_array[0], &reference );
    C_LOG( " ProSelectionToReference" );
}
else
{
    status = ProModelitemInit ( model, 55, PRO_EDGE, &model_item );
    C_LOG( " ProModelitemInit" );
    status = ProReferenceAlloc ( &reference );
    C_LOG( " ProReferenceAlloc" );
    status = ProReferenceSet ( reference, NULL, &model_item );
    C_LOG( " ProReferenceSet" );
}

```

```

}
status = ProCrvcollinstrReferenceAdd ( instr, reference );
C_LOG( " ProCrvcollinstrReferenceAdd" );
status = ProCrvcollectionInstructionAdd ( collection, instr );
C_LOG( " ProCrvcollectionInstructionAdd" );
status = ProCrvcollinstrAlloc ( PRO_CURVCOLL_ADD_TANGENT_INSTR, &instr
);
C_LOG( " ProCrvcollinstrAlloc" );
status = ProCrvcollectionInstructionAdd ( collection, instr );
C_LOG( " ProCrvcollectionInstructionAdd" );
status = ProElementCollectionSet( pro_e_std_curve_collection_appl,
                                collection );
C_LOG( " ProElementCollectionSet " );
status = ProElemtreeElementAdd ( pro_e_rndch_references, NULL,
                                pro_e_std_curve_collection_appl );
C_LOG( " ProElemtreeElementAdd" );
/*-----*\
    Following elements are "not required" for Chamfer , but are
    "required" for the Round Feature - but a copy of the chamfer element
    tree extracted from a chamfer feature will contain these elements
    0- PRO_E_RNDCH_COMPOUND_SPINE
    1- PRO_E_RNDCH_BALL_SPINE
    2- PRO_E_STD_CURVE_COLLECTION_APPL
\*-----*/

/*-----*\
    Following elements are "not used" for Chamfer as listed in ProRound.h
    0- PRO_E_RNDCH_COMPOUND_EXT_OPTIONS
    1- |-- PRO_E_RNDCH_AUTO_BLEND
    2- |-- PRO_E_RNDCH_TERM_SURFACE
\*-----*/

/*-----*\
    Populating element PRO_E_RNDCH_RADII
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_RADII *** " );
status = ProElementAlloc ( PRO_E_RNDCH_RADII, &pro_e_rndch_radii );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_rndch_set, NULL,
                                pro_e_rndch_radii );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_RNDCH_RADIUS
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_RADIUS *** " );
status = ProElementAlloc ( PRO_E_RNDCH_RADIUS, &pro_e_rndch_radius );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_rndch_radii, NULL,
                                pro_e_rndch_radius );
C_LOG( " ProElemtreeElementAdd" );

```

```

/*-----*\
    Populating element PRO_E_RNDCH_LEG1
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_LEG1 *** " );
status = ProElementAlloc ( PRO_E_RNDCH_LEG1, &pro_e_rndch_leg1 );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_rndch_radius, NULL,
                                pro_e_rndch_leg1 );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_RNDCH_LEG_TYPE
    ProRoundRadiusType -
    + PRO_ROUND_RADIUS_TYPE_VALUE,
    + PRO_ROUND_RADIUS_THROUGH_POINT
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_LEG_TYPE *** " );
status = ProElementAlloc ( PRO_E_RNDCH_LEG_TYPE, &pro_e_rndch_leg_type
                           );
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_rndch_leg_type,
                               PRO_ROUND_RADIUS_TYPE_VALUE );
C_LOG( " ProElementIntegerSet" );
status = ProElemtreeElementAdd ( pro_e_rndch_leg1, NULL,
                                pro_e_rndch_leg_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_RNDCH_LEG_VALUE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_LEG_VALUE *** " );
status = ProElementAlloc ( PRO_E_RNDCH_LEG_VALUE, &pro_e_rndch_leg_value
                           );
C_LOG( " ProElementAlloc " );
status = ProElementDoubleSet ( pro_e_rndch_leg_value, 26.500000 );
C_LOG( " ProElementDoubleSet" );
status = ProElemtreeElementAdd ( pro_e_rndch_leg1, NULL,
                                pro_e_rndch_leg_value );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_RNDCH_LEG2 - Optional
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_LEG2 *** " );
status = ProElementAlloc ( PRO_E_RNDCH_LEG2, &pro_e_rndch_leg2 );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_rndch_radius, NULL,
                                pro_e_rndch_leg2 );
C_LOG( " ProElemtreeElementAdd" );

```

```

/*-----*\
    Populating element PRO_E_RNDCH_LEG_TYPE - Optional
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_LEG_TYPE *** " );
status = ProElementAlloc ( PRO_E_RNDCH_LEG_TYPE,
                          &pro_e_rndch_leg_type );
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_rndch_leg_type, 0 );
C_LOG( " ProElementIntegerSet" );
status = ProElemtreeElementAdd ( pro_e_rndch_leg2, NULL,
                                pro_e_rndch_leg_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_RNDCH_LEG_VALUE - Optional
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_LEG_VALUE *** " );
status = ProElementAlloc ( PRO_E_RNDCH_LEG_VALUE,
                          &pro_e_rndch_leg_value );
C_LOG( " ProElementAlloc " );
status = ProElementDoubleSet ( pro_e_rndch_leg_value, 0.000000 );
C_LOG( " ProElementDoubleSet" );
status = ProElemtreeElementAdd ( pro_e_rndch_leg2, NULL,
                                pro_e_rndch_leg_value );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_CURVE_COLLECTION_APPL - Optional ( empty
    collection )
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_CURVE_COLLECTION_APPL *** "
        );
status = ProElementAlloc ( PRO_E_STD_CURVE_COLLECTION_APPL,
                          &pro_e_std_curve_collection_appl );
C_LOG( " ProElementAlloc " );

/*-----*\
    Populating element PRO_E_RNDCH_ATTACH_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_ATTACH_TYPE *** " );
status = ProElementAlloc ( PRO_E_RNDCH_ATTACH_TYPE,
                          &pro_e_rndch_attach_type );
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_rndch_attach_type,
                                PRO_ROUND_ATTACHED );
C_LOG( " ProElementIntegerSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_rndch_attach_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\

```

```

    Populating element PRO_E_RNDCH_TRANSITIONS - Optional for the current
    chamfer
/*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_TRANSITIONS *** " );
status = ProElementAlloc ( PRO_E_RNDCH_TRANSITIONS,
                          &pro_e_rndch_transitions );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_rndch_transitions );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Creating feature in the current model.
/*-----*/
status = ProMdlToModelitem( model, &model_item );
status = ProSelectionAlloc ( p_comp_path, &model_item, &model_sel);

opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMS;
status = ProFeatureCreate (model_sel, pro_e_feature_tree, opts, 1,
                          &feature, &errors);
C_LOG ( " ProFeatureCreate");

/*-----*\
    Free up the allocated memory.
/*-----*/

status = ProElementFree (&pro_e_feature_tree );
C_LOG ( " ProElementFree");

return (status);
}

#undef C_LOG
#undef C_PRINT

```

Example 3: Sample code for creation of a Edge Chamfer Feature

The example shows how to create a Edge Chamfer feature. The user is prompted to select an edge on which the chamfer will be created.

```

/*----- Pro/Toolkit Includes -----*/
#include "ProToolkit.h"
#include "ProFeature.h"
#include "ProElemId.h"
#include "ProExtrude.h"
#include "ProModFeat.h"
#include "ProStdSection.h"
#include "ProElement.h"
#include "ProElemPath.h"
#include "ProFeatType.h"

```

```

#include "ProFeatForm.h"
#include "ProSelection.h"
#include "ProSection.h"
#include "ProModelitem.h"
#include "ProSrfcollection.h"
#include "ProCrvcollection.h"
#include "ProRound.h"
#include "ProChamfer.h"

static ProFileName message_file;

#define C_LOG(a) ProTKPrintf ("Status for %s is = %d\n", a, status ); \
                    ERROR_CHECK( a, "UgChamferCreate.c", status );

#define C_PRINT(a) ProTKPrintf ( "%s\n", a);

/*=====*\
FUNCTION : UserEdgeChamferCreate()
PURPOSE : Function for creation of Edge Chamfer features.
\*=====*/
ProError UserEdgeChamferCreate()
{
    ProErrorlist          errors;
    ProMdl                model;
    ProSelection          model_sel;
    ProFeature            feature;
    ProFeatureCreateOptions opts[1];
    ProElempath           path;
    ProElempathItem       path_items[2];
    ProSection            section;
    ProAsmcomppath        comp_path;
    ProAsmcomppath        *p_comp_path = NULL;

    ProCollection collection;
    ProCrvcollinstr instr;
    ProModelitem model_item;
    ProReference reference;
    ProName wide_string;

    ProElement pro_e_feature_tree;
    ProElement pro_e_feature_type;
    ProElement pro_e_std_feature_name;
    ProElement pro_e_rndch_sets;
    ProElement pro_e_rndch_set;
    ProElement pro_e_rndch_dimensional_schema;
    ProElement pro_e_rndch_chamfer_shape;
    ProElement pro_e_rndch_shape_options;
    ProElement pro_e_rndch_variable_radius;
    ProElement pro_e_rndch_compound_conic;
    ProElement pro_e_rndch_conic_type;
    ProElement pro_e_rndch_conic_value;

```



```

ProElement pro_e_rndch_conic_dep_opt;
ProElement pro_e_rndch_references;
ProElement pro_e_rndch_reference_type;
ProElement pro_e_std_curve_collection_appl;
ProElement pro_e_rndch_compound_spine;
ProElement pro_e_rndch_ball_spine;
ProElement pro_e_rndch_compound_ext_options;
ProElement pro_e_rndch_auto_blend;
ProElement pro_e_rndch_radii;
ProElement pro_e_rndch_radius;
ProElement pro_e_rndch_leg1;
ProElement pro_e_rndch_leg_type;
ProElement pro_e_rndch_leg_value;
ProElement pro_e_rndch_leg2;
ProElement pro_e_rndch_attach_type;
ProElement pro_e_rndch_transitions;

ProStringToWstring ( message_file, "utilities.txt" );

status = ProMdlCurrentGet (&model);
if ( status != PRO_TK_NO_ERROR ) return ( status );

/*-----*\
   Populating root element PRO_E_FEATURE_TREE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TREE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TREE, &pro_e_feature_tree );
C_LOG( " ProElementAlloc " );

/*-----*\
   Populating element PRO_E_FEATURE_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TYPE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TYPE, &pro_e_feature_type );
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_feature_type, PRO_FEAT_CHAMFER );
C_LOG( " ProElementIntegerSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_feature_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
   Populating element PRO_E_STD_FEATURE_NAME - Optional
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_FEATURE_NAME *** " );
status = ProElementAlloc ( PRO_E_STD_FEATURE_NAME,
                          &pro_e_std_feature_name );
C_LOG( " ProElementAlloc " );
ProStringToWstring ( wide_string, "CHAMFER_ID_XXX" );
status = ProElementWstringSet ( pro_e_std_feature_name, wide_string );
C_LOG( " ProElementWstringSet" );

```

```

status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_std_feature_name );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_RNDCH_SETS
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_SETS *** " );
status = ProElementAlloc ( PRO_E_RNDCH_SETS, &pro_e_rndch_sets );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_rndch_sets );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_RNDCH_SET
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_SET *** " );
status = ProElementAlloc ( PRO_E_RNDCH_SET, &pro_e_rndch_set );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_rndch_sets, NULL, pro_e_rndch_set
                                );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_RNDCH_DIMENSIONAL_SCHEMA
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_DIMENSIONAL_SCHEMA *** " );
status = ProElementAlloc ( PRO_E_RNDCH_DIMENSIONAL_SCHEMA,
                          &pro_e_rndch_dimensional_schema );
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_rndch_dimensional_schema,
                              PRO_CHM_D_X_D );
C_LOG( " ProElementIntegerSet" );
status = ProElemtreeElementAdd ( pro_e_rndch_set, NULL,
                              pro_e_rndch_dimensional_schema );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_RNDCH_CHAMFER_SHAPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_CHAMFER_SHAPE *** " );
status = ProElementAlloc ( PRO_E_RNDCH_CHAMFER_SHAPE,
                          &pro_e_rndch_chamfer_shape );
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_rndch_chamfer_shape,
                              PRO_CHM_OFFSET_SURFACE );
C_LOG( " ProElementIntegerSet" );
status = ProElemtreeElementAdd ( pro_e_rndch_set, NULL,
                              pro_e_rndch_chamfer_shape );
C_LOG( " ProElemtreeElementAdd" );

```

```

/*-----*\
    Populating element PRO_E_RNDCH_REFERENCES
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_REFERENCES *** " );
status = ProElementAlloc ( PRO_E_RNDCH_REFERENCES,
                          &pro_e_rndch_references );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_rndch_set, NULL,
                                pro_e_rndch_references );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_CURVE_COLLECTION_APPL
    - One by one curve collection for edges to be chamfered
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_CURVE_COLLECTION_APPL *** "
        );
status = ProElementAlloc ( PRO_E_STD_CURVE_COLLECTION_APPL,
                          &pro_e_std_curve_collection_appl );
C_LOG( " ProElementAlloc " );
status = ProCrvcollectionAlloc ( &collection );
C_LOG( " ProCrvcollectionAlloc" );
status = ProCrvcollinstrAlloc ( PRO_CURV_COLL_ADD_ONE_INSTR, &instr );
C_LOG( " ProCrvcollinstrAlloc" );
{
    ProSelection *sel_array;
    int n_sel_array;

    ProTKPrintf( " Please select edge for CHAMFER \n" );
    status = ProMessageDisplay ( message_file, "USER Select reference." );
    C_LOG( " ProMessageDisplay" );
    status = ProSelect ( "edge,curve", 1, NULL, NULL, NULL, NULL,
                        &sel_array, &n_sel_array );
    C_LOG( " ProSelect" );

    if ( n_sel_array <= 0 )
        return PRO_TK_GENERAL_ERROR;

    status = ProSelectionToReference ( sel_array[0], &reference );
    C_LOG( " ProSelectionToReference" );
}
status = ProCrvcollinstrReferenceAdd ( instr, reference );
C_LOG( " ProCrvcollinstrReferenceAdd" );
status = ProCrvcollectionInstructionAdd ( collection, instr );
C_LOG( " ProCrvcollectionInstructionAdd" );
status = ProCrvcollinstrAlloc ( PRO_CURV_COLL_ADD_TANGENT_INSTR, &instr
                                );
C_LOG( " ProCrvcollinstrAlloc" );
status = ProCrvcollectionInstructionAdd ( collection, instr );
C_LOG( " ProCrvcollectionInstructionAdd" );

```

```

status = ProElementCollectionSet( pro_e_std_curve_collection_appl,
                                collection );
C_LOG( " ProElementCollectionSet " );
status = ProElemtreeElementAdd ( pro_e_rndch_references, NULL,
                                pro_e_std_curve_collection_appl );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_RNDCH_RADII
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_RADII *** " );
status = ProElementAlloc ( PRO_E_RNDCH_RADII, &pro_e_rndch_radii );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_rndch_set, NULL,
                                pro_e_rndch_radii );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_RNDCH_RADIUS
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_RADIUS *** " );
status = ProElementAlloc ( PRO_E_RNDCH_RADIUS, &pro_e_rndch_radius );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_rndch_radii, NULL,
                                pro_e_rndch_radius );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_RNDCH_LEG1
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_LEG1 *** " );
status = ProElementAlloc ( PRO_E_RNDCH_LEG1, &pro_e_rndch_leg1 );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_rndch_radius, NULL,
                                pro_e_rndch_leg1 );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_RNDCH_LEG_TYPE
    ProRoundRadiusType -
    + PRO_ROUND_RADIUS_TYPE_VALUE,
    + PRO_ROUND_RADIUS_THROUGH_POINT
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_LEG_TYPE *** " );
status = ProElementAlloc ( PRO_E_RNDCH_LEG_TYPE, &pro_e_rndch_leg_type);
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_rndch_leg_type,
                                PRO_ROUND_RADIUS_TYPE_VALUE );
C_LOG( " ProElementIntegerSet" );
status = ProElemtreeElementAdd ( pro_e_rndch_leg1, NULL,
                                pro_e_rndch_leg_type );

```

```

C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_RNDCH_LEG_VALUE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_LEG_VALUE *** " );
status = ProElementAlloc ( PRO_E_RNDCH_LEG_VALUE, &pro_e_rndch_leg_value
    );
C_LOG( " ProElementAlloc " );
status = ProElementDoubleSet ( pro_e_rndch_leg_value, 26.500000 );
C_LOG( " ProElementDoubleSet" );
status = ProElemtreeElementAdd ( pro_e_rndch_leg1, NULL,
    pro_e_rndch_leg_value );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_RNDCH_ATTACH_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_RNDCH_ATTACH_TYPE *** " );
status = ProElementAlloc ( PRO_E_RNDCH_ATTACH_TYPE,
    &pro_e_rndch_attach_type );
C_LOG( " ProElementAlloc " );
status = ProElementIntegerSet ( pro_e_rndch_attach_type,
    PRO_ROUND_ATTACHED );
/* 0 :: Ref ProRoundAttachType :: ProRound.h */
C_LOG( " ProElementIntegerSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
    pro_e_rndch_attach_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Creating feature in the current model.
\*-----*/
status = ProMdlToModelitem( model, &model_item );
status = ProSelectionAlloc (p_comp_path, &model_item,
    &model_sel);
opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMS;
status = ProFeatureCreate (model_sel, pro_e_feature_tree, opts, 1,
    &feature, &errors);
C_LOG ( " ProFeatureCreate");

/*-----*\
    Free up the allocated memory.
\*-----*/

status = ProElementFree (&pro_e_feature_tree );
C_LOG ( " ProElementFree");

return (status);
}

```

```
#undef C_LOG
#undef C_PRINT
```

Corner Chamfer Feature

Feature Element Tree for Corner Chamfers

The element tree for a chamfer is documented in the header file *ProChamfer.h*, and has a simple structure. The following figure demonstrates the feature element tree structure:

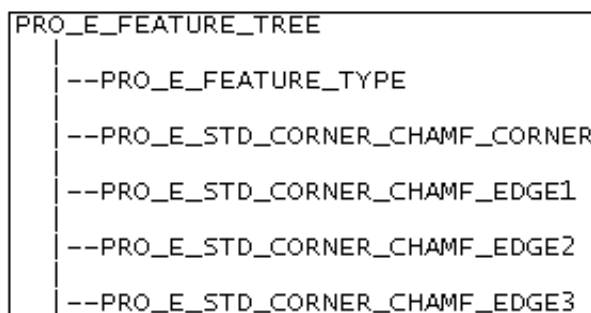


Figure 36-8: Corner Chamfer Element Tree

The feature element tree contains no non-standard element types. The following list details special information about some of the elements in this tree:

- **PRO_E_STD_FEATURE_TYPE**— Specifies the feature type and should be **PRO_FEAT_CORN_CHAMF**.
- **PRO_E_STD_CORNER_CHAMF_CORNER**—Specifies the corner reference and can be selection of **PRO_EDGE**, **PRO_EDGE_PNT**, **PRO_EDGE_START** or **PRO_EDGE_END**. When using **PRO_EDGE** and **PRO_EDGE_PNT** type of reference appropriate parameter should be set using the function **ProReferenceParamsSet()**.
- **PRO_E_STD_CORNER_CHAMF_EDGE1**— Specifies the edge placement dimension common to the specified corner and is the offset of first corner edge.
- **PRO_E_STD_CORNER_CHAMF_EDGE2**— Specifies the edge placement dimension common to the specified corner and is the offset of second corner edge.

- **PRO_E_STD_CORNER_CHAMF_EDGE3**— Specifies the edge placement dimension common to the specified corner and is the offset of third corner edge.

Creating a Corner Chamfer

Function Introduced:

- **ProFeatureCreate()**

Use the function **ProFeatureCreate()** to create a Corner Chamfer based on element tree input. For more information about **ProFeatureCreate()**, refer to the section “Overview of Feature Creation” of chapter “Element Trees: Principles of Feature Creation”.

Redefining a Corner Chamfer

Function Introduced:

- **ProFeatureRedefine()**

Use the function **ProFeatureRedefine()** to redefine a Corner Chamfer based on the changes made in the element tree. For more information about **ProFeatureRedefine()**, refer to the section “Feature Redefine” of chapter “Element Trees: Principles of Feature Creation”.

Accessing a Corner Chamfer

Function Introduced:

- **ProFeatureElemtreeExtract()**

Use the function **ProFeatureElemtreeExtract()** to create a feature element tree that describes the contents of a Corner Chamfer and to retrieve the element tree description of a Corner Chamfer. For more information about **ProFeatureElemtreeExtract()**, refer to the section “Feature Inquiry” of chapter “Element Trees: Principles of Feature Creation”.

Element Trees: Hole

This chapter describes the programmatic creation of Hole features using the Pro/TOOLKIT include file *ProHole.h*.

We recommend you read the section, Overview of Feature Creation in the chapter, Element Trees: Hole. It provides the necessary background for creating features using Pro/TOOLKIT.

Topic	Page
Overview	37 - 2
Feature Element Tree for Hole Features	37 - 3
Feature Element Data Types	37 - 5
Common Element Values	37 - 8
PRO_E_HLE_COM Values	37 - 9
Valid PRO_E_HLE_COM Sub-Elements	37 - 64
Hole Placement Types	37 - 69
Miscellaneous Information	37 - 73

Note: Pro/TOOLKIT applications written with the Release 2000i element tree are not compatible with the Release 2000i2 element tree. The element tree for Hole features in Release 2000i2 was greatly enhanced to include many options not available in Release 2000i. Applications using Release 2000i2 must use the new Hole element tree, published in ProHole.h. Such applications cannot use the old hole element tree when creating, redefining, or investigating hole features.

Overview

Pro/TOOLKIT supports four types of Holes:

- Straight
- Standard
- Sketched
- Custom

The Standard Hole type is sub-divided into two categories:

- Standard Clearance Hole
- Standard Threaded Hole

This chapter details the procedure and the sequence of the creation of the element tree for all of the above types.

All Hole types and placement types require entry of specific elements during element tree creation. Elements must be entered in the specified order.

To create a Hole feature, first add to the element tree all elements related to the hole type. Then, add the elements required for Hole placement. Creating Sketched Holes uses techniques similar to creation of the other sketched features (see Element Trees: Sketched Features).

Note: All angle elements are specified in degrees.

You can use Intent Datums such as Intent Point, Intent Axis, and Intent Plane for hole placement in parts.

Lightweight holes can be created only in parts and not in assemblies. You can toggle between a lightweight and regular hole, only in case of simple holes and not in sketched, standard, or custom holes.

Feature Element Tree for Hole Features

The element tree for the Hole feature is documented in the Pro/TOOLKIT header file *ProHole.h*.

Figure 37-1: Feature Element Tree for Hole Feature

```
PRO_E_FEATURE_TREE
|
|--PRO_E_FEATURE_TYPE
|
|--PRO_E_FEATURE_FORM
|
|--PRO_E_HLE_COM
|
|--PRO_E_HLE_PLACEMENT
|
|--PRO_E_INT_PARTS
|
|--PRO_E_PATTERN
|
|--PRO_E_STD_FEATURE_NAME
```

Figure 37-2: Common Elements for Hole Types

```

PRO_E_HLE_COM
|--PRO_E_HLE_TYPE_NEW
|--PRO_E_HLE_STAN_TYPE
|--PRO_E_HLE_THRDSERIS
|--PRO_E_HLE_FITTYPE
|--PRO_E_HLE_SCREWSIZE
|--PRO_E_HLE_ADD_THREAD
|--PRO_E_HLE_ADD_CBORE
|--PRO_E_HLE_ADD_CSINK
|--PRO_E_HLE_MAKE_LIGHTWT
|
|--PRO_E_DIAMETER
|
|--PRO_E_HOLE_STD_DEPTH
|   |--PRO_E_HOLE_DEPTH_TO
|   |   |--PRO_E_HOLE_DEPTH_TO_TYPE
|   |   |--PRO_E_EXT_DEPTH_TO_VALUE
|   |   |--PRO_E_EXT_DEPTH_TO_REF
|   |--PRO_E_HOLE_DEPTH_FROM
|   |   |--PRO_E_HOLE_DEPTH_FROM_TYPE
|   |   |--PRO_E_EXT_DEPTH_FROM_VALUE
|   |   |--PRO_E_EXT_DEPTH_FROM_REF
|
|--PRO_E_HLE_HOLEDIAM
|--PRO_E_HLE_DEPTH
|--PRO_E_HLE_CSINKANGLE
|--PRO_E_HLE_CBOREDEPTH
|--PRO_E_HLE_CBOREDIAM
|--PRO_E_HLE_CSINKDIAM
|--PRO_E_HLE_DEPTH_DIM_TYPE
|--PRO_E_HLE_THRD_DEPTH
|--PRO_E_HLE_THRDDEPTH
|--PRO_E_HLE_DRILLANGLE
|--PRO_E_HLE_DRILLDEPTH
|--PRO_E_STD_HOLE_DEPTH_REF
|--PRO_E_SKETCHER
|--PRO_E_HLE_CRDIR_FLIP
|--PRO_E_HLE_ADD_EXIT_CSINK
|--PRO_E_HLE_EXIT_CSINKANGLE
|--PRO_E_HLE_EXIT_CSINKDIAM
|--PRO_E_HLE_ADD_NOTE
|--PRO_E_HOLE_NOTE

```

Figure 37-3: Common Elements for Hole Placement

```
PRO_E_HLE_PLACEMENT
|--PRO_E_HLE_PRIM_REF
|--PRO_E_HLE_PL_TYPE
|--PRO_E_HLE_DIM_REF1
|--PRO_E_HLE_PLC_ALIGN_OPT1
|--PRO_E_HLE_DIM_DIST1
|--PRO_E_HLE_DIM_REF2
|--PRO_E_HLE_PLC_ALIGN_OPT2
|--PRO_E_HLE_DIM_DIST2
|--PRO_E_LIN_HOLE_DIR_REF
|--PRO_E_HLE_AXIS
|--PRO_E_HLE_REF_PLANE
|--PRO_E_HLE_REF_ANG
|--PRO_E_HLE_DIM_DIA
|--PRO_E_HLE_DIM_RAD
|--PRO_E_HLE_DIM_LIN
|--PRO_E_HLE_NORM_PLA
|--PRO_E_HLE_NORM_OFFST
|--PRO_E_HLE_PLCMNT_PLANE
|--PRO_E_HLE_REF_PLANE_1
|--PRO_E_HLE_REF_ANG_1
```

Element Trees: Hole

Feature Element Data Types

The following table lists data types for hole type and placement elements.

Element values must be of the specified type.

Hole Element Table

Element Id	Element Name	Data Type
PRO_E_FEATURE_TYPE	Feature Type	PRO_VALUE_TYPE_INT
PRO_E_FEATURE_FORM	Feature Form	PRO_VALUE_TYPE_INT
PRO_E_STD_FEATURE_NAME	Feature Name	PRO_VALUE_TYPE_WSTRING
PRO_E_HLE_COM	Hole	Compound
PRO_E_HLE_TYPE_NEW	Hole Type	PRO_VALUE_TYPE_INT
PRO_E_HLE_STAN_TYPE	Standard Type	PRO_VALUE_TYPE_INT
PRO_E_HLE_THRDSERIS	Thread Series	PRO_VALUE_TYPE_INT

Hole Element Table

Element Id	Element Name	Data Type
PRO_E_HLE_FITTYPE	Fit Type	PRO_VALUE_TYPE_INT
PRO_E_HLE_SCREWSIZE	Screw Size	PRO_VALUE_TYPE_INT
PRO_E_HLE_DEPTH	Depth	PRO_VALUE_TYPE_INT
PRO_E_STD_HOLE_DEPTH_REF	Reference	PRO_VALUE_TYPE_SELECTION
PRO_E_HLE_DEPTH_DIM_TYPE	Depth Dim Scheme	PRO_VALUE_TYPE_INT
PRO_E_HLE_HOLEDIAM	Diameter	PRO_VALUE_TYPE_DOUBLE
PRO_E_HLE_DRILLANGLE	Drillhead Angle	PRO_VALUE_TYPE_DOUBLE
PRO_E_HLE_ADD_THREAD	Add Thread	PRO_VALUE_TYPE_INT
PRO_E_HLE_THRD_DEPTH	Thread Depth	PRO_VALUE_TYPE_INT
PRO_E_HLE_THRDDEPTH	Thread Depth	PRO_VALUE_TYPE_DOUBLE
PRO_E_HLE_ADD_CBORE	Add Counterbore	PRO_VALUE_TYPE_INT
PRO_E_HLE_CBOREDEPTH	Counterbore Depth	PRO_VALUE_TYPE_DOUBLE
PRO_E_HLE_CBOREDIAM	Counterbore Diameter	PRO_VALUE_TYPE_DOUBLE
PRO_E_HLE_ADD_CSINK	Add Countersink	PRO_VALUE_TYPE_INT
PRO_E_HLE_CSINKANGLE	Csink Angle	PRO_VALUE_TYPE_DOUBLE
PRO_E_HLE_CSINKDIAM	Csink Diameter	PRO_VALUE_TYPE_DOUBLE
PRO_E_HLE_DRILLDEPTH	Drill Depth	PRO_VALUE_TYPE_DOUBLE
PRO_E_HLE_ADD_EXIT_CSINK	Add Exit Csink	PRO_VALUE_TYPE_INT
PRO_E_HLE_EXIT_CSINKANGLE	Exit Csink Angle	PRO_VALUE_TYPE_DOUBLE
PRO_E_HLE_EXIT_CSINKDIAM	Exit Csink Diameter	PRO_VALUE_TYPE_DOUBLE
PRO_E_HLE_ADD_NOTE	Add Hole Note	PRO_VALUE_TYPE_INT
PRO_E_HOLE_NOTE	Hole Note	The element is not accessible through Pro/TOOLKIT
PRO_E_DIAMETER	Diameter	PRO_VALUE_TYPE_DOUBLE

Hole Element Table

Element Id	Element Name	Data Type
PRO_E_HLE_MAKE_LIGHTWT	Make lightweight hole	PRO_VALUE_TYPE_INT (It is given by the enumerated type <i>ProHleLightWtFlag</i>)
PRO_E_HOLE_STD_DEPTH	Depth Element	Compound
PRO_E_HOLE_DEPTH_TO	Depth Two	Compound
PRO_E_HOLE_DEPTH_TO_TYPE	Depth Two	PRO_VALUE_TYPE_INT
PRO_E_EXT_DEPTH_TO_VALUE	Depth Value	PRO_VALUE_TYPE_DOUBLE
PRO_E_EXT_DEPTH_TO_REF	Reference	PRO_VALUE_TYPE_SELECTION
PRO_E_HOLE_DEPTH_FROM	Depth One	Compound
PRO_E_HOLE_DEPTH_FROM_TYPE	Depth One	PRO_VALUE_TYPE_INT
PRO_E_EXT_DEPTH_FROM_VALUE	Depth Value	PRO_VALUE_TYPE_DOUBLE
PRO_E_EXT_DEPTH_FROM_REF	Reference	PRO_VALUE_TYPE_SELECTION
PRO_E_HLE_SKETCHER	Sketcher	N/A
PRO_E_HLE_CRDIR_FLIP	Creation Direction	PRO_VALUE_TYPE_INT
PRO_E_HLE_PLACEMENT	Placement	N/A
PRO_E_HLE_PRIM_REF	Primary Reference	PRO_VALUE_TYPE_SELECTION
PRO_E_HLE_PL_TYPE	Placement Options	PRO_VALUE_TYPE_INT
PRO_E_HLE_DIM_REF1	DimensionRef 1	PRO_VALUE_TYPE_SELECTION
PRO_E_HLE_PLG_ALIGN_OPT1	Alignment for placement reference 1	PRO_VALUE_TYPE_INT
PRO_E_HLE_DIM_DIST1	Distance 1	PRO_VALUE_TYPE_DOUBLE
PRO_E_HLE_PLG_ALIGN_OPT2	Alignment for placement reference 2	PRO_VALUE_TYPE_INT
PRO_E_HLE_DIM_REF2	DimensionRef 2	PRO_VALUE_TYPE_SELECTION
PRO_E_LIN_HOLE_DIR_REF	Reference Direction	PRO_VALUE_TYPE_SELECT

Hole Element Table

Element Id	Element Name	Data Type
PRO_E_HLE_DIM_DIST2	Distance 2	PRO_VALUE_TYPE_DOUBLE
PRO_E_HLE_AXIS	Axis	PRO_VALUE_TYPE_SELECTION
PRO_E_HLE_REF_PLANE	Reference Plane	PRO_VALUE_TYPE_SELECTION
PRO_E_HLE_REF_ANG	Angle	PRO_VALUE_TYPE_DOUBLE
PRO_E_HLE_DIM_TYPE	Dimension Type	PRO_VALUE_TYPE_INT
PRO_E_HLE_DIM_DIA	Diameter	PRO_VALUE_TYPE_DOUBLE
PRO_E_HLE_DIM_RAD	Radius	PRO_VALUE_TYPE_DOUBLE
PRO_E_HLE_DIM_LIN	Linear Distance	PRO_VALUE_TYPE_DOUBLE
PRO_E_HLE_NORM_PLA	Normal Plane	PRO_VALUE_TYPE_SELECTION
PRO_E_HLE_NORM_OFFST	Offset	PRO_VALUE_TYPE_DOUBLE
PRO_E_HLE_PLCMNT_PLANE	Placement Plane	PRO_VALUE_TYPE_SELECTION
PRO_E_HLE_REF_PLANE_1	Reference Plane	PRO_VALUE_TYPE_SELECTION
PRO_E_HLE_REF_ANG_1	Angle	PRO_VALUE_TYPE_DOUBLE
PRO_E_INT_PARTS	Intsct Parts	N/A
PRO_E_PATTERN	Pattern	N/A

Common Element Values

All holes require definition of the feature type and feature form. The following table shows valid values for the common elements in the hole element tree.

Common Element Values

Element ID	Value
PRO_E_FEATURE_TYPE	PRO_FEAT_HOLE

Common Element Values

Element ID	Value
PRO_E_FEATURE_FORM	PRO_HLE_TYPE_STRAIGHT (for straight holes) PRO_HLE_TYPE_SKETCHED (for other hole types)
PRO_E_STD_FEATURE_NAME	Wstring (feature name)

PRO_E_HLE_COM Values

Values required for PRO_E_HLE_COM compound element vary for different hole types. The following tables show the PRO_E_HLE_COM element values required to define different hole types. Be sure to enter the elements into the element tree in the order specified by these tables.

Straight Hole

The following table shows elements for creating a straight hole.

Straight Hole Elements

Element	Status
PRO_E_HLE_TYPE_NEW	PRO_HLE_NEW_TYPE_STRAIGHT
PRO_E_DIAMETER	Mandatory
PRO_E_HOLE_STD_DEPTH	Mandatory
PRO_E_HOLE_DEPTH_TO	Mandatory
PRO_E_HOLE_DEPTH_TO_TYPE	Mandatory
PRO_E_EXT_DEPTH_TO_VALUE	Depends on PRO_E_HOLE_DEPTH_TO_TYPE
PRO_E_EXT_DEPTH_TO_REF	Depends on PRO_E_HOLE_DEPTH_TO_TYPE
PRO_E_HOLE_DEPTH_FROM	Mandatory
PRO_E_HOLE_DEPTH_FROM_TYPE	Mandatory
PRO_E_EXT_DEPTH_FROM_VALUE	Depends on PRO_E_HOLE_DEPTH_FROM_TYPE
PRO_E_EXT_DEPTH_FROM_REF	Depends on PRO_E_HOLE_DEPTH_FROM_TYPE

Figure Straight Hole with Linear Placement shows code for creating a straight hole with linear placement and through-all depth. The hole has 100 units diameter, and is placed 100 units distant from the first reference and 200 units distance from the second.

The function **ProDemoHoleCreate()** builds the complete element tree serially. First add all elements required for the straight hole under the PRO_E_HLE_COM element. Then enter the placement elements under the PRO_E_HLE_PLACEMENT element. Use element PRO_E_HOLE_DEPTH_TO_TYPE to specify the hole as 'through all'.

The function **UserElemtreeElementAdd()** is a small utility that add an element to the element tree.

Straight Hole with Linear Placement

```

/*----- Pro/Toolkit Includes -----*/

#include "ProToolkit.h"
#include "ProFeature.h"
#include "ProElemId.h"
#include "ProStdSection.h"
#include "ProElement.h"
#include "ProFeatType.h"
#include "ProFeatForm.h"
#include "ProSelection.h"
#include "ProHole.h"

/*----- MACRO -----*/
/* UG_HOLE_LOG : Macro to log the calls to functions inputs :
   a - name of the function getting called
   b - error return by the function called
*/
#define UG_HOLE_LOG( a, b ) ProTKPrintf(" Return value of function\n%s is %d\n", a, b );

/*----- Function Prototypes -----*/

ProError UserElemtreeElementAdd( ProElement parent_element, ProElement
                                child_element, ProValueData value_data );

/*----- External Data -----*/

/*----- Global Data -----*/

/*=====*\
FUNCTION : ProDemoHoleCreate
PURPOSE  : Demonstrates the creation of a straight linear hole
\*=====*/

ProError ProDemoHoleCreate()
{
    ProError status;

    ProElement feat_elemtree;

```

```

ProElement elem_feattype;
ProElement elem_featform;

ProElement elem_hle_com;
ProElement elem_hle_type_new;
ProElement elem_hle_stan_type;

ProElement elem_diameter;

ProElement elem_hole_std_depth;
ProElement elem_hole_depth_to;
ProElement elem_hole_depth_to_type;
ProElement elem_ext_depth_to_value;
ProElement elem_ext_depth_to_ref;
ProElement elem_hole_depth_from;
ProElement elem_hole_depth_from_type;
ProElement elem_ext_depth_from_value;
ProElement elem_ext_depth_from_ref;

ProElement elem_hle_placement;
ProElement elem_hle_prim_ref;
ProElement elem_hle_pl_type;
ProElement elem_hle_dim_ref1;
ProElement elem_hle_dim_dist1;
ProElement elem_hle_dim_ref2;
ProElement elem_hle_dim_dist2;

ProValue value;
ProValueData value_data;

ProSelection *p_selection;
int n_selection;

ProFeatureCreateOptions options[] = { PRO_FEAT_CR_DEFINE_MISS_ELEMS };
ProFeature created_feature;
ProErrorlist p_errors;
ProMdl model;
ProModelitem model_item;
ProSelection model_selection;

/* Start of Element Tree Creation */

/* Adding the root element */

status = ProElementAlloc ( PRO_E_FEATURE_TREE, &feat_elemtree );

/* Adding the element for feature type */

ProTKPrintf(" PRO_E_FEATURE_TYPE \n");
ProTKPrintf(" ***** \n");
status = ProElementAlloc ( PRO_E_FEATURE_TYPE, &elem_feattype );

```

```

UG_HOLE_LOG (" ProElementAlloc ", status);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_FEAT_HOLE ;

status = UserElemtreeElementAdd( feat_elemtree, elem_feattype,
                                value_data );
UG_HOLE_LOG (" ProElementAlloc ", status );

/* Adding the element for feature form */

ProTKPrintf(" PRO_E_FEATURE_FORM \n");
ProTKPrintf(" ***** \n");

status = ProElementAlloc ( PRO_E_FEATURE_FORM , &elem_featform );

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_HLE_TYPE_STRAIGHT ;

status = UserElemtreeElementAdd( feat_elemtree, elem_featform,
                                value_data );
UG_HOLE_LOG (" ProElementAlloc ", status );

/* Adding the common element for hole information */

ProTKPrintf(" PRO_E_HLE_COM \n");
ProTKPrintf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_COM, &elem_hle_com );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_COM", status);

status = ProElemtreeElementAdd ( feat_elemtree, NULL,
                                elem_hle_com );

UG_HOLE_LOG (" ProElemtreeElementAdd ", status);

/* Adding the element for hole type : Straight, Standard, Sketched */

ProTKPrintf(" PRO_E_HLE_TYPE_NEW \n");
ProTKPrintf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_TYPE_NEW, &elem_hle_type_new );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_TYPE_NEW ", status);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_HLE_NEW_TYPE_STRAIGHT ;

status = UserElemtreeElementAdd ( elem_hle_com, elem_hle_type_new,
                                value_data );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status);

```

```

/* Specifying the Diameter of the Hole */

ProTKPrintf(" PRO_E_DIAMETER \n");
ProTKPrintf(" ***** \n");

status = ProElementAlloc ( PRO_E_DIAMETER, &elem_diameter );
UG_HOLE_LOG (" ProElementAlloc PRO_E_DIAMETER ", status);

value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 100.0 ;

status = UserElemtreeElementAdd ( elem_hle_com, elem_diameter,
                                value_data );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status);

/* Adding an element for the Standard Depth

    |--PRO_E_HOLE_STD_DEPTH
    |
    | |--PRO_E_HOLE_DEPTH_TO
    | |
    | | |--PRO_E_HOLE_DEPTH_TO_TYPE
    | | |--PRO_E_EXT_DEPTH_TO_VALUE
    | | |--PRO_E_EXT_DEPTH_TO_REF
    | |
    | |--PRO_E_HOLE_DEPTH_FROM
    | |
    | | |--PRO_E_HOLE_DEPTH_FROM_TYPE
    | | |--PRO_E_EXT_DEPTH_FROM_VALUE
    | | |--PRO_E_EXT_DEPTH_FROM_REF
    |

*/

ProTKPrintf(" PRO_E_HOLE_STD_DEPTH \n");
ProTKPrintf(" ***** \n");

status = ProElementAlloc ( PRO_E_HOLE_STD_DEPTH,
                          &elem_hole_std_depth );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HOLE_STD_DEPTH ", status);

status = ProElemtreeElementAdd ( elem_hle_com, NULL,
                                elem_hole_std_depth );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status);

ProTKPrintf(" PRO_E_HOLE_DEPTH_TO \n", status);
ProTKPrintf(" ***** \n");

status = ProElementAlloc ( PRO_E_HOLE_DEPTH_TO,
                          &elem_hole_depth_to );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HOLE_DEPTH_TO ", status);

status = ProElemtreeElementAdd ( elem_hole_std_depth, NULL,
                                elem_hole_depth_to );
UG_HOLE_LOG (" ProElemtreeElementAdd ", status);

```

```

ProTKPrintf(" PRO_E_HOLE_DEPTH_TO_TYPE \n");
ProTKPrintf(" ***** \n");

status = ProElementAlloc ( PRO_E_HOLE_DEPTH_TO_TYPE,
                           &elem_hole_depth_to_type );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HOLE_DEPTH_TO_TYPE ", status);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_HLE_STRGHT_THRU_ALL_DEPTH ;

status = UserElemtreeElementAdd ( elem_hole_depth_to,
                                  elem_hole_depth_to_type, value_data );
UG_HOLE_LOG (" ProElemtreeElementAdd ", status);

ProTKPrintf(" PRO_E_HOLE_DEPTH_FROM \n");
ProTKPrintf(" ***** \n");

status = ProElementAlloc ( PRO_E_HOLE_DEPTH_FROM,
                           &elem_hole_depth_from );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HOLE_DEPTH_FROM ", status);

status = ProElemtreeElementAdd ( elem_hole_std_depth, NULL,
                                  elem_hole_depth_from );
UG_HOLE_LOG (" ProElemtreeElementAdd ", status);

ProTKPrintf(" PRO_E_HOLE_DEPTH_FROM_TYPE \n");
ProTKPrintf(" ***** \n");

status = ProElementAlloc ( PRO_E_HOLE_DEPTH_FROM_TYPE,
                           &elem_hole_depth_from_type );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HOLE_DEPTH_FROM_TYPE ",
              status);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_HLE_STRGHT_NONE_DEPTH ;

status = UserElemtreeElementAdd ( elem_hole_depth_from,
                                  elem_hole_depth_from_type, value_data);
UG_HOLE_LOG (" ProElemtreeElementAdd ", status);

/* Adding elements related with the placement details

| --PRO_E_HLE_PLACEMENT
|   | --PRO_E_HLE_PRIM_REF
|   | --PRO_E_HLE_PL_TYPE
|   | --PRO_E_HLE_DIM_REF1
|   | --PRO_E_HLE_DIM_DIST1
|   | --PRO_E_HLE_DIM_REF2
|   | --PRO_E_HLE_DIM_DIST2

*/

```

```

ProTKPrintf(" PRO_E_HLE_PLACEMENT \n");
ProTKPrintf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_PLACEMENT,
                           &elem_hle_placement );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_PLACEMENT", status);

status = ProElemtreeElementAdd ( feat_elemtree, NULL,
                                elem_hle_placement );
UG_HOLE_LOG (" ProElemtreeElementAdd ", status);

ProTKPrintf(" PRO_E_HLE_PRIM_REF \n");
ProTKPrintf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_PRIM_REF, &elem_hle_prim_ref );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_PRIM_REF ", status);

ProTKPrintf("Select a surface for Hole Placement \n");
status = ProSelect ( "datum,surface", 1, NULL, NULL, NULL, NULL,
                    &p_selection, &n_selection);
if ( n_selection <= 0 ) return (0);
UG_HOLE_LOG (" ProSelect ", status);

value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_selection[0] ;

status = UserElemtreeElementAdd ( elem_hle_placement,
                                elem_hle_prim_ref,
                                value_data );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status);

ProTKPrintf(" PRO_E_HLE_PL_TYPE \n");
ProTKPrintf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_PL_TYPE, &elem_hle_pl_type );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_PL_TYPE ", status);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_HLE_PL_TYPE_LIN;

status = UserElemtreeElementAdd ( elem_hle_placement,
                                elem_hle_pl_type,
                                value_data );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status);

ProTKPrintf(" PRO_E_HLE_DIM_REF1 \n");
ProTKPrintf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_DIM_REF1, &elem_hle_dim_ref1 );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_DIM_REF1 ", status);

```

```

ProTKPrintf("Select a 1st Reference for Hole Placement \n");
status = ProSelect ( "datum,surface,edge", 1, NULL, NULL, NULL, NULL,
                    &p_selection, &n_selection);
if ( n_selection <= 0 ) return (0);
UG_HOLE_LOG ( " ProSelect ", status);

value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_selection[0] ;

status = UserElemtreeElementAdd ( elem_hle_placement,
                                  elem_hle_dim_ref1,
                                  value_data );
UG_HOLE_LOG ( " UserElemtreeElementAdd ", status);

ProTKPrintf(" PRO_E_HLE_DIM_DIST1 \n");
ProTKPrintf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_DIM_DIST1,
                           &elem_hle_dim_dist1 );
UG_HOLE_LOG ( " ProElementAlloc PRO_E_HLE_DIM_DIST1 ", status);

value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 100.0;

status = UserElemtreeElementAdd ( elem_hle_placement,
                                  elem_hle_dim_dist1,
                                  value_data );
UG_HOLE_LOG ( " UserElemtreeElementAdd ", status);

ProTKPrintf(" PRO_E_HLE_DIM_REF2 \n");
ProTKPrintf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_DIM_REF2,
                           &elem_hle_dim_ref2 );
UG_HOLE_LOG ( " ProElementAlloc PRO_E_HLE_DIM_REF2 ", status);

ProTKPrintf("Select a 2st Reference for Hole Placement \n");
status = ProSelect ( "datum,surface,edge", 1, NULL,
                    NULL, NULL, NULL,
                    &p_selection, &n_selection);
if ( n_selection <= 0 ) return (0);
UG_HOLE_LOG ( " ProSelect ", status);

value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_selection[0] ;

status = UserElemtreeElementAdd ( elem_hle_placement,
                                  elem_hle_dim_ref2,
                                  value_data );
UG_HOLE_LOG ( " UserElemtreeElementAdd ", status);

```



```

ProTKPrintf(" PRO_E_HLE_DIM_DIST2 \n");
ProTKPrintf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_DIM_DIST2,
                           &elem_hle_dim_dist2 );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_DIM_DIST2 ", status);

value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 200.0;

status = UserElemtreeElementAdd ( elem_hle_placement,
                                  elem_hle_dim_dist2,
                                  value_data );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status);

/* End of Element Tree Creation */

/* Start of Feature Creation */

status = ProMdlCurrentGet ( &model );
UG_HOLE_LOG (" ProMdlCurrentGet ", status);

status = ProMdlToModelitem( model, &model_item );
UG_HOLE_LOG (" ProMdlToModelitem ", status);

status = ProSelectionAlloc( NULL, &model_item, &model_selection );
UG_HOLE_LOG (" ProSelectionAlloc ", status);

status = ProFeatureCreate ( model_selection, feat_elemtree,
                           options, 1,
                           &created_feature, &p_errors );
UG_HOLE_LOG (" ProFeatureCreate ", status);

/* End of Feature Creation */

/* Freeing the resources */

status = ProElementFree( &feat_elemtree );
UG_HOLE_LOG (" ProElementFree ", status);

status = ProSelectionFree( &model_selection );
UG_HOLE_LOG (" ProSelectionFree ", status);

/* End of Freeing */
return ( status );
}
/*****\
FUNCTION : UserElemtreeElementAdd
PURPOSE  : Adding an element to the parent element / tree
input    : parent_element, child_element, value_data

```

```

output : none
returns :
PRO_TK_NO_ERROR - Successfully added the element
PRO_TK_GENERAL_ERROR - Failed to add the element
any other return as returned by ProElemtreeElementAdd()
\*=====*/

ProError UserElemtreeElementAdd( ProElement parent_element,
                                ProElement child_element,
                                ProValueData value_data )
{
    ProValue value;
    ProError status;

    status = ProValueAlloc ( &value );
    UG_HOLE_LOG ( " ProValueAlloc ", status);

    status = ProValueDataSet ( value, &value_data );
    UG_HOLE_LOG ( " ProValueDataSet ", status);

    status = ProElementValueSet ( child_element, value );
    UG_HOLE_LOG ( " ProElementValueSet ", status);

    status = ProElemtreeElementAdd ( parent_element, NULL,
                                    child_element );
    UG_HOLE_LOG ( " ProElemtreeElementAdd ", status);

    return ( status );
}

```

Sketched Hole

Pro/TOOLKIT supports two methods for creating sketched holes. The first is as described in ‘Element Trees: Sketched Features’, the second uses the function **ProFeatureCreate()** more directly.

The following table describes the required elements for sketched hole features.

Sketched Hole Elements

Element	Status
PRO_E_HLE_TYPE_NEW	PRO_HLE_NEW_TYPE_SKETCH
PRO_E_SKETCHER	Mandatory
PRO_E_HLE_CRDIR_FLIP	Mandatory

Procedure Using Techniques from Creating Sketched Features

1. Add the required elements for the sketched feature as outlined in the table in section Sketched Hole.
2. Add all the placement elements.
3. Set the argument *ProFeatureCreateOptions* for **ProFeatureCreate()** to **PRO_FEAT_CR_INCOMPLETE_FEAT** and call **ProFeatureCreate()** with the created element tree.
4. Fetch the section handle for the section of the incomplete feature, using the sequence of calls **ProElemPathAlloc()**, **ProElemPathDataSet()**, and **ProFeatureElemValueGet()**.
5. Create a 2D revolved section with the retrieved section handle. Add the center-line for the axis of revolution as required for the section for revolved feature in Pro/ENGINEER user interface.
6. Attach the new section to the element tree, then call **ProFeatureRedefine()** with the element tree created in these steps.

Refer to Sketched Hole with Conventional Approach for a code example of this technique of hole creation.

Sketched Hole with Conventional Approach

Example 1: Creating a Standard Sketched Hole with Linear Placement

In the conventional approach:

- Use **ProFeatureCreate()** to create incomplete feature
- Use the feature handle to get the section handle
- Build the section
- Give a call to **ProFeatureRedefine()** to redefine
- Complete the feature.

```
#include "ProToolkit.h"
#include "ProFeature.h"
#include "ProElemId.h"
#include "ProStdSection.h"
#include "ProElement.h"
#include "ProFeatType.h"
#include "ProFeatForm.h"
#include "ProSelection.h"
#include "ProHole.h"
/*-----*\
MACRO
```

```

\*-----*/

/*-----*\
UG_HOLE_LOG : Macro to log the calls to functions inputs :
  a - name of the function getting called
  b - error return by the function called
\*-----*/

#define UG_HOLE_LOG( a, b )
printf(" Return value of function %s is %d\n", a, b );

/*----- Function Prototypes -----*/

ProError UserElemtreeElementAdd( ProElement parent_element,
ProElement child_element,
ProValueData value_data );

ProError UserHoleSketchRedefine ( ProFeature feature_here,
ProElement feat_elemtree );

ProError UserHoleSectionAdd ( ProSection section_here );

/*-----*\
External Data
\*-----*/

/*-----*\
Global Data
\*-----*/

/*=====*\
FUNCTION : ProDemoSketchedConvHoleCreate
PURPOSE : Demonstrates the creation of a Standard Threaded hole
with linear placement
\*=====*/

ProError ProDemoSketchedConvHoleCreate()
{
ProError status;

ProElement feat_elemtree;
ProElement elem_feattype;
ProElement elem_featform;

ProElement elem_hle_com;
ProElement elem_hle_type_new;
ProElement elem_hle_sketcher;
ProElement elem_hle_crdir_flip;

ProElement elem_hle_placement;
ProElement elem_hle_prim_ref;
ProElement elem_hle_pl_type;

```

```

ProElement elem_hle_dim_ref1;
ProElement elem_hle_dim_dist1;
ProElement elem_hle_dim_ref2;
ProElement elem_hle_dim_dist2;

ProValue value;
ProValueData value_data;

ProSelection *p_selection;
int n_selection;

ProFeatureCreateOptions options[] = { PRO_FEAT_CR_INCOMPLETE_FEAT };
ProFeatureCreateOptions redefine_options[] = {
PRO_FEAT_CR_DEFINE_MISS_ELEMS };
ProFeature created_feature;
ProErrorlist p_errors;
ProMdl model;
ProModelitem model_item;
ProSelection model_selection;

ProAsmcomppath comp_path;
/*-----*\
Start of Element Tree Creation
\*-----*/

/*-----*\
Adding the root element
\*-----*/

status = ProElementAlloc ( PRO_E_FEATURE_TREE, &feat_elemtree );

/*-----*\
Adding the element for feature type
\*-----*/

printf(" PRO_E_FEATURE_TYPE \n");
printf(" ***** \n");
status = ProElementAlloc ( PRO_E_FEATURE_TYPE, &elem_feattype );
UG_HOLE_LOG (" ProElementAlloc ", status);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_FEAT_HOLE ;

status = UserElemtreeElementAdd( feat_elemtree,
                                elem_feattype, value_data );
UG_HOLE_LOG (" ProElementAlloc ", status );

/*-----*\
Adding the element for feature form
\*-----*/

printf(" PRO_E_FEATURE_FORM \n");

```

```

printf(" ***** \n");

status = ProElementAlloc ( PRO_E_FEATURE_FORM , &elem_featform );
UG_HOLE_LOG (" ProElementAlloc ", status );

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_HLE_TYPE_STRAIGHT ;

status = UserElemtreeElementAdd( feat_elemtree, elem_featform,
                                value_data );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status );

/*-----*\
Adding the common element for hole information
\*-----*/

printf(" PRO_E_HLE_COM \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_COM, &elem_hle_com );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_COM", status);

status = ProElemtreeElementAdd ( feat_elemtree, NULL, elem_hle_com );

UG_HOLE_LOG (" ProElemtreeElementAdd ", status);
/*-----*\
Adding the element for hole type: Straight, Standard, Sketched
\*-----*/

printf(" PRO_E_HLE_TYPE_NEW \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_TYPE_NEW, &elem_hle_type_new );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_TYPE_NEW ", status);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_HLE_NEW_TYPE_SKETCH ;

status = UserElemtreeElementAdd ( elem_hle_com, elem_hle_type_new,
                                value_data );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status);

/*-----*\
Elements for Sketched Hole
\*-----*/

printf(" PRO_E_HLE_CRDIR_FLIP \n");
printf(" ***** \n");

status = ProElementAlloc (PRO_E_HLE_CRDIR_FLIP, &elem_hle_crdir_flip );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_CRDIR_FLIP ", status);

```

```

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_HLE_CR_IN_SIDE_ONE;

status = UserElemtreeElementAdd ( elem_hle_com, elem_hle_crdir_flip,
                                value_data );
UG_HOLE_LOG ( " UserElemtreeElementAdd ", status);

/*-----*\
Elements related to the placement details

|--PRO_E_HLE_PLACEMENT
| |--PRO_E_HLE_PRIM_REF
| |--PRO_E_HLE_PL_TYPE
| |--PRO_E_HLE_DIM_REF1
| |--PRO_E_HLE_DIM_DIST1
| |--PRO_E_HLE_DIM_REF2
| |--PRO_E_HLE_DIM_DIST2

\*-----*/

printf(" PRO_E_HLE_PLACEMENT \n");
printf(" ***** \n");

status = ProElementAlloc (PRO_E_HLE_PLACEMENT,&elem_hle_placement );
UG_HOLE_LOG ( " ProElementAlloc PRO_E_HLE_PLACEMENT", status);

status = ProElemtreeElementAdd (feat_elemtree, NULL, elem_hle_placement);
UG_HOLE_LOG ( " ProElemtreeElementAdd ", status);

printf(" PRO_E_HLE_PRIM_REF \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_PRIM_REF, &elem_hle_prim_ref );
UG_HOLE_LOG ( " ProElementAlloc PRO_E_HLE_PRIM_REF ", status);

printf("Select a surface for Hole Placement \n");
status = ProSelect ( "datum,surface", 1, NULL, NULL, NULL, NULL,
                    &p_selection, &n_selection);
if ( n_selection <= 0 ) return (0);
UG_HOLE_LOG ( " ProSelect ", status);

value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_selection[0] ;

status = UserElemtreeElementAdd ( elem_hle_placement,
                                elem_hle_prim_ref, value_data );
UG_HOLE_LOG ( " UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_PL_TYPE \n");
printf(" ***** \n");

```

```

status = ProElementAlloc ( PRO_E_HLE_PL_TYPE, &elem_hle_pl_type );
UG_HOLE_LOG ( " ProElementAlloc PRO_E_HLE_PL_TYPE ", status);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_HLE_PL_TYPE_LIN;

status = UserElemtreeElementAdd ( elem_hle_placement,
                                elem_hle_pl_type, value_data );
UG_HOLE_LOG ( " UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_DIM_REF1 \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_DIM_REF1, &elem_hle_dim_ref1 );
UG_HOLE_LOG ( " ProElementAlloc PRO_E_HLE_DIM_REF1 ", status);

printf("Select a 1st Reference for Hole Placement \n");
status = ProSelect ( "datum,surface,edge", 1, NULL, NULL,
                    NULL, NULL,
                    &p_selection, &n_selection);
if ( n_selection <= 0 ) return (0);
UG_HOLE_LOG ( " ProSelect ", status);

value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_selection[0] ;

status = UserElemtreeElementAdd ( elem_hle_placement,
                                elem_hle_dim_ref1, value_data );
UG_HOLE_LOG ( " UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_DIM_DIST1 \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_DIM_DIST1, &elem_hle_dim_dist1 );
UG_HOLE_LOG ( " ProElementAlloc PRO_E_HLE_DIM_DIST1 ", status);

value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 0.1;

status = UserElemtreeElementAdd ( elem_hle_placement,
                                elem_hle_dim_dist1, value_data );
UG_HOLE_LOG ( " UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_DIM_REF2 \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_DIM_REF2, &elem_hle_dim_ref2 );
UG_HOLE_LOG ( " ProElementAlloc PRO_E_HLE_DIM_REF2 ", status);

printf("Select a 2st Reference for Hole Placement \n");

```



```

status = ProSelect ( "datum,surface,edge", 1, NULL, NULL, NULL, NULL,
                    &p_selection, &n_selection);
if ( n_selection <= 0 ) return (0);
UG_HOLE_LOG ( " ProSelect ", status);

value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_selection[0] ;

status = UserElemtreeElementAdd ( elem_hle_placement,
                                elem_hle_dim_ref2, value_data );
UG_HOLE_LOG ( " UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_DIM_DIST2 \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_DIM_DIST2,
                          &elem_hle_dim_dist2 );
UG_HOLE_LOG ( " ProElementAlloc PRO_E_HLE_DIM_DIST2 ", status);

value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 0.2;

status = UserElemtreeElementAdd ( elem_hle_placement,
                                elem_hle_dim_dist2, value_data );
UG_HOLE_LOG ( " UserElemtreeElementAdd ", status);

/*-----*\
End of Element Tree Creation
\*-----*/
/*-----*\
Start of Feature Creation
\*-----*/

status = ProMdlCurrentGet ( &model );
UG_HOLE_LOG ( " ProMdlCurrentGet ", status);

status = ProMdlToModelitem( model, &model_item );
UG_HOLE_LOG ( " ProMdlToModelitem ", status);

status = ProSelectionAlloc( NULL, &model_item, &model_selection );
UG_HOLE_LOG ( " ProSelectionAlloc ", status);

status = ProFeatureCreate ( model_selection, feat_elemtree,
                          options, 1,
                          &created_feature, &p_errors );
UG_HOLE_LOG ( " ProFeatureCreate ", status);

/*-----*\
Redefining the feature
\*-----*/

```

```

status = UserHoleSketchRedefine ( created_feature, feat_elemtree );
UG_HOLE_LOG ( " UserHoleSketchRedefine ", status);

status = ProSelectionAsmcomppathGet ( model_selection, &comp_path );
UG_HOLE_LOG ( " ProSelectionAsmcomppathGet ", status);

status = ProFeatureRedefine ( &comp_path, &created_feature,
                             feat_elemtree,
                             redefine_options, 1,
                             &p_errors );
UG_HOLE_LOG ( " ProFeatureRedefine ", status);

/*-----*\
Freeing the resources
\*-----*/
status = ProElementFree( &feat_elemtree );
UG_HOLE_LOG ( " ProElementFree ", status);

status = ProSelectionFree( &model_selection );
UG_HOLE_LOG ( " ProSelectionFree ", status);

/*-----*\
End of Freeing
\*-----*/
return ( status );
}

/*=====*\
FUNCTION : UserElemtreeElementAdd
PURPOSE  : Adding an element to the parent element / tree
input: parent_element, child_element, value_data
output  : none
returns :
PRO_TK_NO_ERROR - Successfully added the element
PRO_TK_GENERAL_ERROR - Failed to add the element
any other return as returned by ProElemtreeElementAdd()
\*=====*/
ProError UserElemtreeElementAdd( ProElement parent_element, ProElement
                                child_element,
                                ProValueData value_data )
{
ProValue value;
ProError status;

status = ProValueAlloc ( &value );
UG_HOLE_LOG ( " ProValueAlloc ", status);

status = ProValueDataSet ( value, &value_data );
UG_HOLE_LOG ( " ProValueDataSet ", status);

status = ProElementValueSet ( child_element, value );
UG_HOLE_LOG ( " ProElementValueSet ", status);

```

```

status = ProElemtreeElementAdd ( parent_element, NULL,
                                child_element );
UG_HOLE_LOG (" ProElemtreeElementAdd ", status);

return ( status );
}
/*=====*\
FUNCTION : UserHoleSketchRedefine
PURPOSE  : Redefining the hole element tree to add PRO_E_HLE_SKETCHER
input    : feature_here, feat_elemtree
output   : none
returns  :
PRO_TK_NO_ERROR - Successfully added the element
PRO_TK_GENERAL_ERROR - Failed to add the element
any other return as returned by ProElemtreeElementAdd()
\*=====*/
ProError UserHoleSketchRedefine ( ProFeature feature_here, ProElement
                                feat_elemtree )
{
    ProError status;
    ProElemPath elempath_hole_sketch, elempath_hole_com;
    ProElemPathItem elempathitem_hole_sketch[2],
                    elempathitem_hole_com[1];

    ProValue value;
    ProValueData value_data;
    ProElement elem_sketcher;
    ProElement elem_hle_com_local;
    ProSection section;
/*-----*\
    Extracting the section handle from the feature
\*-----*/
    status = ProElemPathAlloc (&elempath_hole_sketch);
    UG_HOLE_LOG (" ProElemPathAlloc PRO_E_SKETCHER ", status);

    elempathitem_hole_sketch[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
    elempathitem_hole_sketch[0].path_item.elem_id = PRO_E_HLE_COM ;
    elempathitem_hole_sketch[1].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
    elempathitem_hole_sketch[1].path_item.elem_id = PRO_E_SKETCHER;

    status = ProElemPathDataSet( elempath_hole_sketch,
                                elempathitem_hole_sketch, 2);
    UG_HOLE_LOG (" ProElemPathDataSet ", status);

    status = ProFeatureElemValueGet ( &feature_here,
                                    elempath_hole_sketch, &value);
    UG_HOLE_LOG (" ProFeatureElemValueGet ", status);

    status = ProValueDataGet ( value, &value_data);
    UG_HOLE_LOG (" ProValueDataGet ", status);

```

```

    section = ( ProSection ) value_data.v.p;

/*-----*\
    Adding the 2D section to the element tree
\*-----*/

    status = UserHoleSectionAdd ( section );
    UG_HOLE_LOG ( " UserHoleSectionAdd ", status);

/*-----*\
    Getting the handle for PRO_E_HLE_COM Element
\*-----*/

    status = ProElempathAlloc (&elempath_hole_com);
    UG_HOLE_LOG ( " ProElempathAlloc PRO_E_SKETCHER ", status);

    elempathitem_hole_com[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
    elempathitem_hole_com[0].path_item.elem_id = PRO_E_HLE_COM ;

    status = ProElempathDataSet ( elempath_hole_com,
                                elempathitem_hole_com, 1);
    UG_HOLE_LOG ( " ProElempathDataSet ", status);

    status = ProElemtreeElementGet ( feat_elemtree, elempath_hole_com,
                                &elem_hle_com_local );
    UG_HOLE_LOG ( " ProElemtreeElementGet ", status);

/*-----*\
    Adding the Sketch to PRO_E_HLE_COM Element
\*-----*/

    printf(" PRO_E_SKETCHER \n");
    printf(" ***** \n");

    status = ProElementAlloc ( PRO_E_SKETCHER, &elem_sketcher );
    UG_HOLE_LOG ( " ProElementAlloc PRO_E_SKETCHER ", status);

    status = ProElementValueSet ( elem_sketcher, value );
    UG_HOLE_LOG ( " ProElementValueSet", status);

    status = ProElemtreeElementAdd( elem_hle_com_local, NULL,
                                elem_sketcher );
    UG_HOLE_LOG ( " ProElemtreeElementAdd", status);

    return ( status );
}
/*=====*\
FUNCTION : UserHoleSectionAdd
PURPOSE  : Adding a 2D Section
input    : section_here
output   : none
returns  :

```

```

PRO_TK_NO_ERROR - Successfully added the element
PRO_TK_GENERAL_ERROR - Failed to add the element
any other return as returned by ProSectionRegenerate()
\*=====*/
ProError UserHoleSectionAdd ( ProSection section_here )

{
Pro2dLinedef line;
ProError status;
int line_id[4];
int cline_id;

ProName section_name;

int i, error_id;

ProWSecerror section_errors1;
ProWSecerror section_errors2;

status = ProSecerrorAlloc ( &section_errors1 );
UG_HOLE_LOG ( " ProSecerrorAlloc ", status);

status = ProSecerrorAlloc ( &section_errors2 );
UG_HOLE_LOG ( " ProSecerrorAlloc ", status);

line.type = PRO_2D_CENTER_LINE;
line.end1[0] = 0;
line.end1[1] = 0;
line.end2[0] = 0;
line.end2[1] = 100;

status = ProSectionEntityAdd ( section_here,
                               (Pro2dEntdef*)&line, &cline_id );
UG_HOLE_LOG ( " ProSectionEntityAdd ", status);

line.type = PRO_2D_LINE;
line.end1[0] = 0;
line.end1[1] = 0;
line.end2[0] = 100;
line.end2[1] = 0;

status = ProSectionEntityAdd ( section_here, (Pro2dEntdef*)&line,
                               &line_id[0] );
UG_HOLE_LOG ( " ProSectionEntityAdd ", status);

line.type = PRO_2D_LINE;
line.end1[0] = 100;
line.end1[1] = 0;
line.end2[0] = 100;
line.end2[1] = 100;

status = ProSectionEntityAdd(section_here,
                               (Pro2dEntdef*)&line, &line_id[1] );

```

```

UG_HOLE_LOG (" ProSectionEntityAdd ", status);

line.type = PRO_2D_LINE;
line.end1[0] = 100;
line.end1[1] = 100;
line.end2[0] = 0;
line.end2[1] = 100;

status = ProSectionEntityAdd (section_here,
    (Pro2dEntdef*)&line, &line_id[2]);
UG_HOLE_LOG (" ProSectionEntityAdd ", status);

line.type = PRO_2D_LINE;
line.end1[0] = 0;
line.end1[1] = 100;
line.end2[0] = 0;
line.end2[1] = 0;

status = ProSectionEntityAdd(section_here,
    (Pro2dEntdef*)&line, &line_id[3] );
UG_HOLE_LOG (" ProSectionEntityAdd ", status);

status = ProSectionEpsilonSet( section_here, 0.1 );
UG_HOLE_LOG (" ProSectionEpsilonSet ", status);

status = ProSectionAutodim ( section_here, &section_errors1 );
UG_HOLE_LOG (" ProSectionAutodim ", status);

status = ProSectionRegenerate ( section_here, &section_errors2 );
UG_HOLE_LOG (" ProSectionRegenerate ", status);
return ( status );
}

```

Procedure Using ProFeatureCreate()

In this approach to sketched hole creation, populate the required elements in the element tree (as shown in the table in section **Sketched Hole**), and then call **ProFeatureCreate()**.

Refer to **Standard Threaded Hole** for a code example for creation of this type of hole.

Refer to **Sketched Hole with ProFeatureCreate()** for a code example of this technique of hole creation.

Sketched Hole with ProFeatureCreate()

Example: Creating a Standard Sketched Hole with Linear Placement

Using new approach:

- Create the complete element tree with the sketcher element
- Call **ProFeatureCreate()** to create the hole feature

```
#include "ProToolkit.h"
#include "ProFeature.h"
#include "ProElemId.h"
#include "ProStdSection.h"
#include "ProElement.h"
#include "ProFeatType.h"
#include "ProFeatForm.h"
#include "ProSelection.h"
#include "ProHole.h"

/*-----*\
                                MACRO
\*-----*/

/*-----*\
UG_HOLE_LOG : Macro to log the calls to functions
inputs :
    a - name of the function getting called
    b - error return by the function called
\*-----*/

#define UG_HOLE_LOG( a, b )  printf(" Return value of function %s is
%d\n", a, b );

/*----- Function Prototypes -----*/

ProError UserElemtreeElementAdd( ProElement parent_element, ProElement
child_element,
                                ProValueData value_data );

ProError UserHoleSectionAdd ( ProSection section_here );

/*-----*\
                                External Data
\*-----*/

/*-----*\
                                Global Data
\*-----*/
```

```

/*=====*\
FUNCTION : ProDemoSketchedModiHoleCreate
PURPOSE  : Demonstrates the creation of a Standard Threaded hole
           with linear placement
\*=====*/

ProError ProDemoSketchedModiHoleCreate()
{
ProError status;

    ProElement feat_elemtree;
        ProElement elem_feattype;
        ProElement elem_featform;

        ProElement elem_hle_com;
        ProElement elem_hle_type_new;
        ProElement elem_hle_sketcher;
        ProElement elem_hle_crdire_flip;

        ProElement elem_hle_placement;
        ProElement elem_hle_prim_ref;
        ProElement elem_hle_pl_type;
        ProElement elem_hle_dim_ref1;
        ProElement elem_hle_dim_dist1;
        ProElement elem_hle_dim_ref2;
        ProElement elem_hle_dim_dist2;

    ProValue value;
    ProValueData value_data;

    ProSelection *p_selection;
    int n_selection;

    ProFeatureCreateOptions options[] = { PRO_FEAT_CR_INCOMPLETE_FEAT };
    ProFeatureCreateOptions redefine_options[] = {
        PRO_FEAT_CR_DEFINE_MISS_ELEMS };

    ProFeature created_feature;
    ProErrorlist p_errors;
    ProMdl model;
    ProModelitem model_item;
    ProSelection model_selection;

    ProAsmcomppath comp_path;

    ProSection section;

/*-----*\
                                Start of Element Tree Creation
\*-----*/

/*-----*\

```



```

                                Adding the root element
/*-----*/

    status = ProElementAlloc ( PRO_E_FEATURE_TREE, &feat_elemtree );

/*-----*\
                                Adding the element for feature type
\*-----*/

    printf(" PRO_E_FEATURE_TYPE \n");
    printf(" ***** \n");
    status = ProElementAlloc ( PRO_E_FEATURE_TYPE, &elem_feattype );
    UG_HOLE_LOG (" ProElementAlloc ", status);

    value_data.type = PRO_VALUE_TYPE_INT;
    value_data.v.i = PRO_FEAT_HOLE ;

    status = UserElemtreeElementAdd( feat_elemtree, elem_feattype,
                                    value_data );
    UG_HOLE_LOG (" ProElementAlloc ", status );

/*-----*\
                                Adding the element for feature form
\*-----*/

    printf(" PRO_E_FEATURE_FORM \n");
    printf(" ***** \n");

    status = ProElementAlloc ( PRO_E_FEATURE_FORM , &elem_featform );
    UG_HOLE_LOG (" ProElementAlloc ", status );

    value_data.type = PRO_VALUE_TYPE_INT;
    value_data.v.i = PRO_HLE_TYPE_STRAIGHT ;

    status = UserElemtreeElementAdd( feat_elemtree, elem_featform,
                                    value_data );
    UG_HOLE_LOG (" UserElemtreeElementAdd ", status );
/*-----*\
                                Adding the common element for hole information
\*-----*/

    printf(" PRO_E_HLE_COM \n");
    printf(" ***** \n");

    status = ProElementAlloc ( PRO_E_HLE_COM, &elem_hle_com );
    UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_COM", status);

    status = ProElemtreeElementAdd ( feat_elemtree, NULL, elem_hle_com );

    UG_HOLE_LOG (" ProElemtreeElementAdd ", status);

```

```

/*-----*\
    Adding the element for hole type: Straight, Standard, Sketched
\*-----*/

printf(" PRO_E_HLE_TYPE_NEW \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_TYPE_NEW, &elem_hle_type_new );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_TYPE_NEW ", status);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_HLE_NEW_TYPE_SKETCH ;

status = UserElemtreeElementAdd ( elem_hle_com, elem_hle_type_new,
value_data );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status);
/*-----*\
    Adding the sketcher element
\*-----*/

printf(" PRO_E_HLE_SKETCHER \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_SKETCHER, &elem_hle_sketcher );

UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_SKETCHER ", status);

value_data.type = PRO_VALUE_TYPE_POINTER;

status = ProSection2DAlloc ( &section );
UG_HOLE_LOG (" ProSection2DAlloc ", status);

status = UserHoleSectionAdd ( section );
UG_HOLE_LOG (" UserHoleSectionAdd ", status);

value_data.v.r = ( void * ) section ;

status = UserElemtreeElementAdd ( elem_hle_com, elem_hle_sketcher,
                                value_data );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_CRDIR_FLIP \n");
printf(" ***** \n");

status = ProElementAlloc (PRO_E_HLE_CRDIR_FLIP, &elem_hle_crdir_flip );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_CRDIR_FLIP ", status);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_HLE_CR_IN_SIDE_ONE;

```

```

        status = UserElemtreeElementAdd ( elem_hle_com, elem_hle_crdir_flip,
                                           value_data );
        UG_HOLE_LOG ( " UserElemtreeElementAdd ", status);

/*-----*\
        Elements related to the placement details

        |--PRO_E_HLE_PLACEMENT
        |   |--PRO_E_HLE_PRIM_REF
        |   |--PRO_E_HLE_PL_TYPE
        |   |--PRO_E_HLE_DIM_REF1
        |   |--PRO_E_HLE_DIM_DIST1
        |   |--PRO_E_HLE_DIM_REF2
        |   |--PRO_E_HLE_DIM_DIST2

\*-----*/

        printf(" PRO_E_HLE_PLACEMENT \n");
        printf(" ***** \n");

        status = ProElementAlloc ( PRO_E_HLE_PLACEMENT,
                                   &elem_hle_placement );
        UG_HOLE_LOG ( " ProElementAlloc PRO_E_HLE_PLACEMENT", status);

        status = ProElemtreeElementAdd ( feat_elemtree, NULL,
                                           elem_hle_placement );
        UG_HOLE_LOG ( " ProElemtreeElementAdd ", status);

        printf(" PRO_E_HLE_PRIM_REF \n");
        printf(" ***** \n");

        status = ProElementAlloc ( PRO_E_HLE_PRIM_REF, &elem_hle_prim_ref );
        UG_HOLE_LOG ( " ProElementAlloc PRO_E_HLE_PRIM_REF ", status);

        printf("Select a surface for Hole Placement \n");
        status = ProSelect ( "datum,surface", 1, NULL, NULL, NULL, NULL,
                             &p_selection, &n_selection);
        if ( n_selection <= 0 ) return (0);
        UG_HOLE_LOG ( " ProSelect ", status);

        value_data.type = PRO_VALUE_TYPE_SELECTION;
        value_data.v.r = p_selection[0] ;

        status = UserElemtreeElementAdd ( elem_hle_placement,
                                           elem_hle_prim_ref, value_data );
        UG_HOLE_LOG ( " UserElemtreeElementAdd ", status);

        printf(" PRO_E_HLE_PL_TYPE \n");
        printf(" ***** \n");

```

```

status = ProElementAlloc ( PRO_E_HLE_PL_TYPE, &elem_hle_pl_type );
UG_HOLE_LOG ( " ProElementAlloc PRO_E_HLE_PL_TYPE ", status);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_HLE_PL_TYPE_LIN;

status = UserElemtreeElementAdd ( elem_hle_placement, elem_hle_pl_type,
                                value_data );
UG_HOLE_LOG ( " UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_DIM_REF1 \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_DIM_REF1, &elem_hle_dim_ref1 );
UG_HOLE_LOG ( " ProElementAlloc PRO_E_HLE_DIM_REF1 ", status);

printf("Select a 1st Reference for Hole Placement \n");
status = ProSelect ( "datum,surface,edge", 1, NULL, NULL, NULL, NULL,
                    &p_selection, &n_selection);
if ( n_selection <= 0 ) return (0);
UG_HOLE_LOG ( " ProSelect ", status);

value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_selection[0] ;

status = UserElemtreeElementAdd ( elem_hle_placement,
                                elem_hle_dim_ref1, value_data );
UG_HOLE_LOG ( " UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_DIM_DIST1 \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_DIM_DIST1, &elem_hle_dim_dist1 );
UG_HOLE_LOG ( " ProElementAlloc PRO_E_HLE_DIM_DIST1 ", status);

value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 0.1;

status = UserElemtreeElementAdd ( elem_hle_placement,
                                elem_hle_dim_dist1, value_data );
UG_HOLE_LOG ( " UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_DIM_REF2 \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_DIM_REF2, &elem_hle_dim_ref2 );
UG_HOLE_LOG ( " ProElementAlloc PRO_E_HLE_DIM_REF2 ", status);

```

```

printf("Select a 2st Reference for Hole Placement \n");
status = ProSelect ( "datum,surface,edge", 1, NULL, NULL, NULL, NULL,
                    &p_selection, &n_selection);
if ( n_selection <= 0 ) return (0);
UG_HOLE_LOG ( " ProSelect ", status);

value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_selection[0] ;

status = UserElemtreeElementAdd ( elem_hle_placement,
elem_hle_dim_ref2, value_data );
UG_HOLE_LOG ( " UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_DIM_DIST2 \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_DIM_DIST2, &elem_hle_dim_dist2 );
UG_HOLE_LOG ( " ProElementAlloc PRO_E_HLE_DIM_DIST2 ", status);

value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 0.2;

status = UserElemtreeElementAdd ( elem_hle_placement,
                                elem_hle_dim_dist2, value_data );
UG_HOLE_LOG ( " UserElemtreeElementAdd ", status);

/*-----*\
                        End of Element Tree Creation
\*-----*/

/*-----*\
                        Start of Feature Creation
\*-----*/

status = ProMdlCurrentGet ( &model );
UG_HOLE_LOG ( " ProMdlCurrentGet ", status);

status = ProMdlToModelitem( model, &model_item );
UG_HOLE_LOG ( " ProMdlToModelitem ", status);

status = ProSelectionAlloc( NULL, &model_item, &model_selection );
UG_HOLE_LOG ( " ProSelectionAlloc ", status);

/*-----*\
                        Calling ProFeatureCreate()
\*-----*/

status = ProFeatureCreate ( model_selection, feat_elemtree,
                          options, 1,
                          &created_feature, &p_errors );
UG_HOLE_LOG ( " ProFeatureCreate ", status);

```

```

/*-----*\
                Freeing the resources
\*-----*/

    status = ProElementFree( &feat_elemtree );
    UG_HOLE_LOG ( " ProElementFree ", status);

    status = ProSelectionFree( &model_selection );
    UG_HOLE_LOG ( " ProSelectionFree ", status);

/*-----*\
                End of Freeing
\*-----*/

    return ( status );
}
/*=====*\
FUNCTION : UserElemtreeElementAdd
PURPOSE  : Adding an element to the parent element / tree
input    : parent_element, child_element, value_data
output   : none
returns  :
    PRO_TK_NO_ERROR - Successfully added the element
    PRO_TK_GENERAL_ERROR - Failed to add the element
    any other return as returned by ProElemtreeElementAdd()
\*=====*/

ProError UserElemtreeElementAdd( ProElement parent_element,
                                ProElement child_element,
                                ProValueData value_data )
{
    ProValue value;
    ProError status;

    status = ProValueAlloc ( &value );
    UG_HOLE_LOG ( " ProValueAlloc ", status);

    status = ProValueDataSet ( value, &value_data );
    UG_HOLE_LOG ( " ProValueDataSet ", status);

    status = ProElementValueSet ( child_element, value );
    UG_HOLE_LOG ( " ProElementValueSet ", status);

    status = ProElemtreeElementAdd ( parent_element, NULL,
                                    child_element );
    UG_HOLE_LOG ( " ProElemtreeElementAdd ", status);

    return ( status );
}
/*=====*\
FUNCTION : UserHoleSectionAdd

```

```

PURPOSE   : Adding a 2D Section
input     : section_here
output    : none
returns   :
    PRO_TK_NO_ERROR - Successfully added the element
    PRO_TK_GENERAL_ERROR - Failed to add the element
    any other return as returned by ProSectionRegenerate()
\*=====*/

ProError UserHoleSectionAdd ( ProSection section_here )
{
    Pro2dLinedef line;
    ProError status;
    int line_id[4];
    int cline_id;

    ProName section_name;

    int i, error_id;

    ProWSecerror section_errors1;
    ProWSecerror section_errors2;

    status = ProSecerrorAlloc ( &section_errors1 );
    UG_HOLE_LOG ( " ProSecerrorAlloc ", status);

    status = ProSecerrorAlloc ( &section_errors2 );
    UG_HOLE_LOG ( " ProSecerrorAlloc ", status);

    line.type = PRO_2D_CENTER_LINE;
    line.end1[0] = 0;
    line.end1[1] = 0;
    line.end2[0] = 0;
    line.end2[1] = 100;

    status = ProSectionEntityAdd ( section_here,
                                   (Pro2dEntdef*)&line, &cline_id );
    UG_HOLE_LOG ( " ProSectionEntityAdd ", status);

    line.type = PRO_2D_LINE;
    line.end1[0] = 0;
    line.end1[1] = 0;
    line.end2[0] = 100;
    line.end2[1] = 0;

    status = ProSectionEntityAdd (section_here,
                                   (Pro2dEntdef*)&line, &line_id[0] );
    UG_HOLE_LOG ( " ProSectionEntityAdd ", status);

    line.type = PRO_2D_LINE;
    line.end1[0] = 100;

```

```

        line.end1[1] = 0;
        line.end2[0] = 100;
        line.end2[1] = 100;

        status = ProSectionEntityAdd ( section_here,
        (Pro2dEntdef*)&line, &line_id[1] );
        UG_HOLE_LOG ( " ProSectionEntityAdd ", status);

        line.type = PRO_2D_LINE;
        line.end1[0] = 100;
        line.end1[1] = 100;
        line.end2[0] = 0;
        line.end2[1] = 100;

        status = ProSectionEntityAdd (section_here,
                                     (Pro2dEntdef*)&line, &line_id[2] );
        UG_HOLE_LOG ( " ProSectionEntityAdd ", status);

        line.type = PRO_2D_LINE;
        line.end1[0] = 0;
        line.end1[1] = 100;
        line.end2[0] = 0;
        line.end2[1] = 0;

        status = ProSectionEntityAdd(section_here,
                                     (Pro2dEntdef*)&line, &line_id[3] );
        UG_HOLE_LOG ( " ProSectionEntityAdd ", status);

        status = ProSectionEpsilonSet( section_here, 0.1 );
        UG_HOLE_LOG ( " ProSectionEpsilonSet ", status);

        status = ProSectionAutodim ( section_here, &section_errors1 );
        UG_HOLE_LOG ( " ProSectionAutodim ", status);

        status = ProSectionRegenerate ( section_here, &section_errors2 );
        UG_HOLE_LOG ( " ProSectionRegenerate ", status);

        return ( status );
}

```

Standard Threaded Hole

The following table shows elements for creating a standard threaded hole.

Refer to Example 2: Creating a Standard Threaded Hole with Linear Placement for a code example on creating this type of hole.

Standard Hole Elements

Element	Status
PRO_E_HLE_TYPE_NEW	Mandatory
PRO_E_HLE_STAN_TYPE	Mandatory
PRO_E_HLE_THRDSERIS	Mandatory
PRO_E_HLE_FITTYPE	Mandatory: set to PRO_HLE_CLOSE_FIT
PRO_E_HLE_SCREWSIZE	Mandatory
PRO_E_HLE_DEPTH	Mandatory
PRO_E_STD_HOLE_DEPTH_REF	Depends on PRO_E_HLE_DEPTH
PRO_E_HLE_CRDIR_FLIP	Mandatory
PRO_E_HLE_HOLEDIAM	Mandatory
PRO_E_HLE_DRILLANGLE	Required for variable depth hole
PRO_E_HLE_DRILLDEPTH	Mandatory, even for a through-all hole. This element is required. If not added, hole creation will succeed but the feature cannot be redefined in the Pro/ENGINEER user interface.
PRO_E_HLE_ADD_THREAD	Mandatory
PRO_E_HLE_THRD_DEPTH	Mandatory
PRO_E_HLE_THRDDEPTH	Mandatory, even for a non-threaded hole or a thru-threaded hole. This element is required. If not added, hole creation will succeed but the feature cannot be redefined in the Pro/ENGINEER user interface.
PRO_E_HLE_ADD_CBORE	Mandatory
PRO_E_HLE_CBOREDEPTH	Required for counterbore option
PRO_E_HLE_CBOREDIAM	Required for counterbore option
PRO_E_HLE_ADD_CSINK	Mandatory
PRO_E_HLE_CSINKANGLE	Required for countersink option
PRO_E_HLE_CSINKDIAM	Required for countersink option
PRO_E_HLE_ADD_EXIT_CSINK	Required for Thru all hole
PRO_E_HLE_EXIT_CSINKANGLE	Required for exit countersink option
PRO_E_HLE_EXIT_CSINKDIAM	Required for exit countersink option
PRO_E_HLE_ADD_NOTE	Required for Hole note
PRO_E_HOLE_NOTE	Depends on PRO_E_HLE_ADD_NOTE

Element Trees: Hole

Example 2: Creating a Standard Threaded Hole with Linear Placement

```
#include "ProToolkit.h"
#include "ProFeature.h"
#include "ProElemId.h"
#include "ProStdSection.h"
#include "ProElement.h"
#include "ProFeatType.h"
#include "ProFeatForm.h"
#include "ProSelection.h"
#include "ProHole.h"

/*-----*\
                                     MACRO
\*-----*/

/*-----*\
UG_HOLE_LOG : Macro to log the calls to functions
inputs :
    a - name of the function getting called
    b - error return by the function called
\*-----*/

#define UG_HOLE_LOG( a, b )  printf(" Return value of function %s is
                                %d\n", a, b );

/*----- Function Prototypes -----*/

ProError UserElemtreeElementAdd( ProElement parent_element,
                                ProElement child_element,
                                ProValueData value_data );

/*-----*\
                                     External Data
\*-----*/

/*-----*\
                                     Global Data
\*-----*/

/*=====*\
FUNCTION : ProDemoStandardThreadedHoleCreate
PURPOSE  : Demonstrates the creation of a Standard Threaded hole
           with linear placement
\*=====*/

ProError ProDemoStandardThreadedHoleCreate()
{
ProError status;
```

```

ProElement feat_elmtree;
  ProElement elem_feattype;
  ProElement elem_featform;

  ProElement elem_hle_com;
  ProElement elem_hle_type_new;
  ProElement elem_hle_stan_type;

  ProElement elem_diameter;

  ProElement elem_hole_std_depth;
  ProElement elem_hole_depth_to;
  ProElement elem_hole_depth_to_type;
  ProElement elem_ext_depth_to_value;
  ProElement elem_ext_depth_to_ref;
  ProElement elem_hole_depth_from;
  ProElement elem_hole_depth_from_type;
  ProElement elem_ext_depth_from_value;
  ProElement elem_ext_depth_from_ref;

  ProElement elem_hle_thrdseris;
  ProElement elem_hle_fittype;
  ProElement elem_hle_screwsize;
  ProElement elem_hle_add_thread;
  ProElement elem_hle_add_bore;
  ProElement elem_hle_add_csink;

  ProElement elem_hle_holediam;
  ProElement elem_hle_drillangle;
  ProElement elem_hle_csinkangle;
  ProElement elem_hle_cboreangle;
  ProElement elem_hle_cborediam;
  ProElement elem_hle_csinkdiam;
  ProElement elem_hle_thrddepth;
  ProElement elem_hle_drilldepth;
  ProElement elem_hle_thrd_depth;
  ProElement elem_hle_depth;
  ProElement elem_hle_sketcher;
  ProElement elem_hle_crdir_flip;
  ProElement elem_hle_placement;
  ProElement elem_hle_prim_ref;
  ProElement elem_hle_pl_type;
  ProElement elem_hle_dim_ref1;
  ProElement elem_hle_dim_dist1;
  ProElement elem_hle_dim_ref2;
  ProElement elem_hle_dim_dist2;

  ProValue value;
  ProValueData value_data;

```

```

ProSelection *p_selection;
int n_selection;

ProFeatureCreateOptions options[] = { PRO_FEAT_CR_DEFINE_MISS_ELEMS };
ProFeature created_feature;
ProErrorlist p_errors;
ProMdl model;
ProModelitem model_item;
ProSelection model_selection;

/*-----*\
                                Start of Element Tree Creation
\*-----*/

/*-----*\
                                Adding the root element
\*-----*/

    status = ProElementAlloc ( PRO_E_FEATURE_TREE, &feat_elemtree );

/*-----*\
                                Adding the element for feature type
\*-----*/

    printf(" PRO_E_FEATURE_TYPE \n");
    printf(" ***** \n");
    status = ProElementAlloc ( PRO_E_FEATURE_TYPE, &elem_feattype );
    UG_HOLE_LOG (" ProElementAlloc ", status);

    value_data.type = PRO_VALUE_TYPE_INT;
    value_data.v.i = PRO_FEAT_HOLE ;

    status = UserElemtreeElementAdd( feat_elemtree,
                                    elem_feattype, value_data );
    UG_HOLE_LOG (" ProElementAlloc ", status );

/*-----*\
                                Adding the element for feature form
\*-----*/

    printf(" PRO_E_FEATURE_FORM \n");
    printf(" ***** \n");

    status = ProElementAlloc ( PRO_E_FEATURE_FORM , &elem_featform );

    value_data.type = PRO_VALUE_TYPE_INT;
    value_data.v.i = PRO_HLE_TYPE_STRAIGHT ;

    status = UserElemtreeElementAdd (feat_elemtree,
                                    elem_featform, value_data );
    UG_HOLE_LOG (" ProElementAlloc ", status );

```

```

/*-----*\
                Adding the common element for hole information
\*-----*/

printf(" PRO_E_HLE_COM \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_COM, &elem_hle_com );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_COM", status);

status = ProElemtreeElementAdd ( feat_elemtree, NULL, elem_hle_com );

UG_HOLE_LOG (" ProElemtreeElementAdd ", status);

/*-----*\
                Adding the element for hole type: Straight, Standard, Sketched
\*-----*/

printf(" PRO_E_HLE_TYPE_NEW \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_TYPE_NEW, &elem_hle_type_new );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_TYPE_NEW ", status);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_HLE_NEW_TYPE_STANDARD ;

status = UserElemtreeElementAdd ( elem_hle_com, elem_hle_type_new,
                                value_data );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status);

/*-----*\
                Elements for Standard Threaded Hole
\*-----*/

printf(" PRO_E_HLE_STAN_TYPE \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_STAN_TYPE, &elem_hle_stan_type );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_STAN_TYPE ", status);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_HLE_TAPPED_TYPE ;
/* PRO_HLE_TAPPED_TYPE PRO_HLE_CLEARANCE_TYPE */

status = UserElemtreeElementAdd ( elem_hle_com, elem_hle_stan_type,
                                value_data );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_THRDSERIS \n");

```

```

printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_THRDSERIS, &elem_hle_thrdseris );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_THRDSERIS ", status);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = 0 ;    /* 2: ISO */

status = UserElemtreeElementAdd ( elem_hle_com, elem_hle_thrdseris,
                                value_data );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_FITTYPE \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_FITTYPE, &elem_hle_fitttype );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_FITTYPE ", status);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_HLE_CLOSE_FIT ;

status = UserElemtreeElementAdd ( elem_hle_com, elem_hle_fitttype,
                                value_data );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_SCREWSIZE \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_SCREWSIZE, &elem_hle_screwsiz );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_SCREWSIZE ", status);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = 0 ;

status = UserElemtreeElementAdd ( elem_hle_com, elem_hle_screwsiz,
                                value_data );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_ADD_THREAD \n");
printf(" ***** \n");

status = ProElementAlloc (PRO_E_HLE_ADD_THREAD, &elem_hle_add_thread );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_ADD_THREAD ", status);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_HLE_ADD_THREAD ;

status = UserElemtreeElementAdd ( elem_hle_com, elem_hle_add_thread,
                                value_data );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status);

```

```

printf(" PRO_E_HLE_ADD_CBORE  \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_ADD_CBORE, &elem_hle_add_bore );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_ADD_CBORE ", status);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_HLE_NO_CBORE;

status = UserElemtreeElementAdd ( elem_hle_com, elem_hle_add_bore,
                                value_data );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_ADD_CSINK  \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_ADD_CSINK, &elem_hle_add_csink );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_ADD_CSINK ", status);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_HLE_NO_CSINK;

status = UserElemtreeElementAdd ( elem_hle_com, elem_hle_add_csink,
                                value_data );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_HOLEDIAM  \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_HOLEDIAM, &elem_hle_holediam );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_HOLEDIAM ", status);

value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 0.05 ;

status = UserElemtreeElementAdd ( elem_hle_com, elem_hle_holediam,
                                value_data );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_THRDDEPTH  \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_THRDDEPTH, &elem_hle_thrddepth );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_THRDDEPTH ", status);

value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 0 ;

status = UserElemtreeElementAdd ( elem_hle_com, elem_hle_thrddepth,
                                value_data );

```

```

UG_HOLE_LOG (" UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_DRILLDEPTH  \n");
printf(" ***** \n");

status = ProElementAlloc (PRO_E_HLE_DRILLDEPTH, &elem_hle_drilldepth );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_DRILLDEPTH ", status);

value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 0 ;

status = UserElemtreeElementAdd ( elem_hle_com, elem_hle_drilldepth,
                                value_data );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_THRD_DEPTH  \n");
printf(" ***** \n");

status = ProElementAlloc (PRO_E_HLE_THRD_DEPTH, &elem_hle_thrd_depth );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_THRD_DEPTH ", status);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_HLE_THRU_THREAD; /* PRO_HLE_VARIABLE_THREAD */

status = UserElemtreeElementAdd ( elem_hle_com, elem_hle_thrd_depth,
                                value_data );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_DEPTH  \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_DEPTH, &elem_hle_depth );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_DEPTH ", status);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_HLE_STD_THRU_ALL_DEPTH;

status = UserElemtreeElementAdd ( elem_hle_com, elem_hle_depth,
                                value_data );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_CRDIR_FLIP  \n");
printf(" ***** \n");

status = ProElementAlloc (PRO_E_HLE_CRDIR_FLIP, &elem_hle_crdir_flip );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_CRDIR_FLIP ", status);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_HLE_CR_IN_SIDE_ONE;

```



```

        status = UserElemtreeElementAdd ( elem_hle_com, elem_hle_crdir_flip,
                                           value_data );
        UG_HOLE_LOG ( " UserElemtreeElementAdd ", status);

/*-----*\
        Elements related to the placement details

        |--PRO_E_HLE_PLACEMENT
        |   |--PRO_E_HLE_PRIM_REF
        |   |--PRO_E_HLE_PL_TYPE
        |   |--PRO_E_HLE_DIM_REF1
        |   |--PRO_E_HLE_DIM_DIST1
        |   |--PRO_E_HLE_DIM_REF2
        |   |--PRO_E_HLE_DIM_DIST2

\*-----*/

        printf(" PRO_E_HLE_PLACEMENT \n");
        printf(" ***** \n");

        status = ProElementAlloc ( PRO_E_HLE_PLACEMENT,
                                   &elem_hle_placement );
        UG_HOLE_LOG ( " ProElementAlloc PRO_E_HLE_PLACEMENT", status);

        status = ProElemtreeElementAdd ( feat_elemtree, NULL,
                                           elem_hle_placement );
        UG_HOLE_LOG ( " ProElemtreeElementAdd ", status);

        printf(" PRO_E_HLE_PRIM_REF \n");
        printf(" ***** \n");

        status = ProElementAlloc ( PRO_E_HLE_PRIM_REF, &elem_hle_prim_ref );
        UG_HOLE_LOG ( " ProElementAlloc PRO_E_HLE_PRIM_REF ", status);

        printf("Select a surface for Hole Placement \n");
        status = ProSelect ( "datum,surface", 1, NULL, NULL, NULL, NULL,
                             &p_selection, &n_selection);
        if ( n_selection <= 0 ) return (0);
        UG_HOLE_LOG ( " ProSelect ", status);

        value_data.type = PRO_VALUE_TYPE_SELECTION;
        value_data.v.r = p_selection[0] ;

        status = UserElemtreeElementAdd ( elem_hle_placement,
                                           elem_hle_prim_ref, value_data );
        UG_HOLE_LOG ( " UserElemtreeElementAdd ", status);

        printf(" PRO_E_HLE_PL_TYPE \n");
        printf(" ***** \n");

        status = ProElementAlloc ( PRO_E_HLE_PL_TYPE, &elem_hle_pl_type );

```

```

UG_HOLE_LOG ( " ProElementAlloc PRO_E_HLE_PL_TYPE ", status);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_HLE_PL_TYPE_LIN;

status = UserElemtreeElementAdd ( elem_hle_placement, elem_hle_pl_type,
                                value_data );
UG_HOLE_LOG ( " UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_DIM_REF1 \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_DIM_REF1, &elem_hle_dim_ref1 );
UG_HOLE_LOG ( " ProElementAlloc PRO_E_HLE_DIM_REF1 ", status);

printf("Select a 1st Reference for Hole Placement \n");
status = ProSelect ( "datum,surface,edge", 1, NULL, NULL, NULL, NULL,
                    &p_selection, &n_selection);
if ( n_selection <= 0 ) return (0);
UG_HOLE_LOG ( " ProSelect ", status);

value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_selection[0] ;

status = UserElemtreeElementAdd ( elem_hle_placement,
elem_hle_dim_ref1, value_data );
UG_HOLE_LOG ( " UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_DIM_DIST1 \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_DIM_DIST1, &elem_hle_dim_dist1 );
UG_HOLE_LOG ( " ProElementAlloc PRO_E_HLE_DIM_DIST1 ", status);

value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 0.1;

status = UserElemtreeElementAdd ( elem_hle_placement,
                                elem_hle_dim_dist1, value_data );
UG_HOLE_LOG ( " UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_DIM_REF2 \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_DIM_REF2, &elem_hle_dim_ref2 );
UG_HOLE_LOG ( " ProElementAlloc PRO_E_HLE_DIM_REF2 ", status);

printf("Select a 2st Reference for Hole Placement \n");
status = ProSelect ( "datum,surface,edge", 1, NULL, NULL, NULL, NULL,
                    &p_selection, &n_selection);
if ( n_selection <= 0 ) return (0);

```

```

UG_HOLE_LOG (" ProSelect ", status);

value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_selection[0] ;

status = UserElemtreeElementAdd ( elem_hle_placement,
                                elem_hle_dim_ref2, value_data );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_DIM_DIST2 \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_DIM_DIST2, &elem_hle_dim_dist2 );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_DIM_DIST2 ", status);

value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 0.2;

status = UserElemtreeElementAdd ( elem_hle_placement,
                                elem_hle_dim_dist2, value_data );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status);

/*-----*\
                End of Element Tree Creation
\*-----*/

/*-----*\
                Start of Feature Creation
\*-----*/

status = ProMdlCurrentGet ( &model );
UG_HOLE_LOG (" ProMdlCurrentGet ", status);

status = ProMdlToModelitem( model, &model_item );
UG_HOLE_LOG (" ProMdlToModelitem ", status);

status = ProSelectionAlloc( NULL, &model_item, &model_selection );
UG_HOLE_LOG (" ProSelectionAlloc ", status);

status = ProFeatureCreate ( model_selection, feat_elemtree,
                           options, 1,
                           &created_feature, &p_errors );
UG_HOLE_LOG (" ProFeatureCreate ", status);

/*-----*\
                End of Feature Creation
\*-----*/

/*-----*\
                Freeing the resources
\*-----*/

```

```

    status = ProElementFree( &feat_elemtree );
    UG_HOLE_LOG ( " ProElementFree ", status);

    status = ProSelectionFree( &model_selection );
    UG_HOLE_LOG ( " ProSelectionFree ", status);

/*-----*\
                End of Freeing
\*-----*/

    return ( status );
}
/*=====*\
FUNCTION : UserElemtreeElementAdd
PURPOSE  : Adding an element to the parent element / tree
input    : parent_element, child_element, value_data
output   : none
returns  :
    PRO_TK_NO_ERROR - Successfully added the element
    PRO_TK_GENERAL_ERROR - Failed to add the element
    any other return as returned by ProElemtreeElementAdd()
\*=====*/

ProError UserElemtreeElementAdd( ProElement parent_element,
                                ProElement child_element,
                                ProValueData value_data )
{
    ProValue value;
    ProError status;

    status = ProValueAlloc ( &value );
    UG_HOLE_LOG ( " ProValueAlloc ", status);

    status = ProValueDataSet ( value, &value_data );
    UG_HOLE_LOG ( " ProValueDataSet ", status);

    status = ProElementValueSet ( child_element, value );
    UG_HOLE_LOG ( " ProElementValueSet ", status);

    status = ProElemtreeElementAdd ( parent_element, NULL,
                                    child_element );
    UG_HOLE_LOG ( " ProElemtreeElementAdd ", status);

    return ( status );
}

```

Standard Clearance Hole

The following table outlines elements for standard clearance holes.

Refer to Example 3: Creating a Standard Clearance Hole with Linear Placement for a code example on creating this type of hole.

Standard Clearance Hole Elements

Element	Status
PRO_E_HLE_TYPE_NEW	Mandatory
PRO_E_HLE_STAN_TYPE	Mandatory
PRO_E_HLE_THRDSERIS	Mandatory
PRO_E_HLE_FITTYPE	Mandatory
PRO_E_HLE_SCREWSIZE	Mandatory
PRO_E_HLE_DEPTH	Mandatory
PRO_E_HLE_HOLEDIAM	Mandatory
PRO_E_HLE_DRILLANGLE	Required for variable depth hole
PRO_E_HLE_ADD_THREAD	Mandatory
PRO_E_HLE_ADD_CBORE	Mandatory
PRO_E_HLE_CBOREDEPTH	Required for counterbore option
PRO_E_HLE_CBOREDIAM	Required for counterbore option
PRO_E_HLE_ADD_CSINK	Mandatory
PRO_E_HLE_CSINKANGLE	Required for countersink option
PRO_E_HLE_CSINKDIAM	Required for countersink option
PRO_E_HLE_ADD_EXIT_CSINK	Required for Thru all hole
PRO_E_HLE_EXIT_CSINKANGLE	Required for exit countersink option
PRO_E_HLE_EXIT_CSINKDIAM	Required for exit countersink option
PRO_E_HLE_ADD_NOTE	Required for Hole note
PRO_E_HOLE_NOTE	Depends on PRO_E_HLE_ADD_NOTE

Element Trees: Hole

Example 3: Creating a Standard Clearance Hole with Linear Placement

```
#include "ProToolkit.h"
#include "ProFeature.h"
#include "ProElemId.h"
#include "ProStdSection.h"
#include "ProElement.h"
#include "ProFeatType.h"
#include "ProFeatForm.h"
#include "ProSelection.h"
#include "ProHole.h"
```

```

/*-----*
MACRO
\*-----*/

/*-----*
UG_HOLE_LOG : Macro to log the calls to functions
inputs :
    a - name of the function getting called
    b - error return by the function called
\*-----*/

#define UG_HOLE_LOG( a, b )  printf(" Return value of function %s
                                is %d\n", a, b );

/*----- Function Prototypes -----*/

ProError UserElemtreeElementAdd( ProElement parent_element,
                                ProElement child_element,
                                ProValueData value_data );

/*-----*
External Data
\*-----*/

/*-----*
Global Data
\*-----*/

/*=====*\
FUNCTION : ProDemoStandardClearanceHoleCreate
PURPOSE  : Demonstrates the creation of a Standard Clearance hole
            with linear placement
\*=====*/

ProError ProDemoStandardClearanceHoleCreate()
{
ProError status;

    ProElement feat_elemtree;
    ProElement elem_feattype;
    ProElement elem_featform;

    ProElement elem_hle_com;
    ProElement elem_hle_type_new;
    ProElement elem_hle_stan_type;

    ProElement elem_diameter;

    ProElement elem_hole_std_depth;
    ProElement elem_hole_depth_to;
    ProElement elem_hole_depth_to_type;
    ProElement elem_ext_depth_to_value;

```

```

ProElement elem_ext_depth_to_ref;
ProElement elem_hole_depth_from;
ProElement elem_hole_depth_from_type;
ProElement elem_ext_depth_from_value;
ProElement elem_ext_depth_from_ref;

ProElement elem_hle_thrdseris;
ProElement elem_hle_fitttype;
ProElement elem_hle_screwsize;
ProElement elem_hle_add_thread;
ProElement elem_hle_add_bore;
ProElement elem_hle_add_csink;

ProElement elem_hle_holediam;
ProElement elem_hle_drillangle;
ProElement elem_hle_csinkangle;
ProElement elem_hle_cboreangle;
ProElement elem_hle_cborediam;
ProElement elem_hle_csinkdiam;
ProElement elem_hle_thrddepth;
ProElement elem_hle_drilldepth;
ProElement elem_hle_thrd_depth;
ProElement elem_hle_depth;
ProElement elem_hle_sketcher;
ProElement elem_hle_crdir_flip;

ProElement elem_hle_placement;
ProElement elem_hle_prim_ref;
ProElement elem_hle_pl_type;
ProElement elem_hle_dim_ref1;
ProElement elem_hle_dim_dist1;
ProElement elem_hle_dim_ref2;
ProElement elem_hle_dim_dist2;

ProValue value;
ProValueData value_data;

ProSelection *p_selection;
int n_selection;

ProFeatureCreateOptions options[] = { PRO_FEAT_CR_DEFINE_MISS_ELEMS };
ProFeature created_feature;
ProErrorlist p_errors;
ProMdl model;
ProModelitem model_item;
ProSelection model_selection;

/*-----*\
                                Start of Element Tree Creation
\*-----*/

```

```

/*-----*\
                        Adding the root element
\*-----*/

    status = ProElementAlloc ( PRO_E_FEATURE_TREE, &feat_elemtree );

/*-----*\
                        Adding the element for feature type
\*-----*/

    printf(" PRO_E_FEATURE_TYPE \n");
    printf(" ***** \n");
    status = ProElementAlloc ( PRO_E_FEATURE_TYPE, &elem_feattype );
    UG_HOLE_LOG (" ProElementAlloc ", status);

    value_data.type = PRO_VALUE_TYPE_INT;
    value_data.v.i = PRO_FEAT_HOLE ;

    status = UserElemtreeElementAdd( feat_elemtree, elem_feattype,
                                    value_data );
    UG_HOLE_LOG (" ProElementAlloc ", status );

/*-----*\
                        Adding the element for feature form
\*-----*/

    printf(" PRO_E_FEATURE_FORM \n");
    printf(" ***** \n");

    status = ProElementAlloc ( PRO_E_FEATURE_FORM , &elem_featform );

value_data.type = PRO_VALUE_TYPE_INT;
    value_data.v.i = PRO_HLE_TYPE_STRAIGHT ;

    status = UserElemtreeElementAdd( feat_elemtree, elem_featform,
                                    value_data );
    UG_HOLE_LOG (" ProElementAlloc ", status );

/*-----*\
                        Adding the common element for hole information
\*-----*/

    printf(" PRO_E_HLE_COM \n");
    printf(" ***** \n");

    status = ProElementAlloc ( PRO_E_HLE_COM, &elem_hle_com );
    UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_COM", status);

    status = ProElemtreeElementAdd ( feat_elemtree, NULL, elem_hle_com );

    UG_HOLE_LOG (" ProElemtreeElementAdd ", status);

```



```

/*-----*\
    Adding the element for hole type: Straight, Standard, Sketched
\*-----*/

printf(" PRO_E_HLE_TYPE_NEW \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_TYPE_NEW, &elem_hle_type_new );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_TYPE_NEW ", status);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_HLE_NEW_TYPE_STANDARD ;

status = UserElemtreeElementAdd ( elem_hle_com, elem_hle_type_new,
                                value_data );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status);

/*-----*\
    Elements for Standard Clearance Hole
\*-----*/

printf(" PRO_E_HLE_STAN_TYPE \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_STAN_TYPE, &elem_hle_stan_type );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_STAN_TYPE ", status);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_HLE_CLEARANCE_TYPE ;

status = UserElemtreeElementAdd ( elem_hle_com, elem_hle_stan_type,
                                value_data );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_THRDSERIS \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_THRDSERIS, &elem_hle_thrdseris );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_THRDSERIS ", status);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = 0 ;    /* 2: ISO */

status = UserElemtreeElementAdd ( elem_hle_com, elem_hle_thrdseris,
                                value_data );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_FITTYPE \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_FITTYPE, &elem_hle_fittype );

```

```

UG_HOLE_LOG ( " ProElementAlloc PRO_E_HLE_FITTYPE ", status);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_HLE_CLOSE_FIT ;

status = UserElemtreeElementAdd ( elem_hle_com, elem_hle_fittype,
                                value_data );
UG_HOLE_LOG ( " UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_SCREWSIZE \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_SCREWSIZE, &elem_hle_screwsizesize );
UG_HOLE_LOG ( " ProElementAlloc PRO_E_HLE_SCREWSIZE ", status);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = 0 ;

status = UserElemtreeElementAdd ( elem_hle_com, elem_hle_screwsizesize,
                                value_data );
UG_HOLE_LOG ( " UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_ADD_THREAD \n");
printf(" ***** \n");

status = ProElementAlloc (PRO_E_HLE_ADD_THREAD, &elem_hle_add_thread );
UG_HOLE_LOG ( " ProElementAlloc PRO_E_HLE_ADD_THREAD ", status);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_HLE_NO_THREAD ;

status = UserElemtreeElementAdd (elem_hle_com, elem_hle_add_thread,
                                value_data );
UG_HOLE_LOG ( " UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_ADD_CBORE \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_ADD_CBORE, &elem_hle_add_bore );
UG_HOLE_LOG ( " ProElementAlloc PRO_E_HLE_ADD_CBORE ", status);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_HLE_NO_CBORE;

status = UserElemtreeElementAdd ( elem_hle_com, elem_hle_add_bore,
                                value_data );
UG_HOLE_LOG ( " UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_ADD_CSINK \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_ADD_CSINK, &elem_hle_add_csink );
UG_HOLE_LOG ( " ProElementAlloc PRO_E_HLE_ADD_CSINK ", status);

```

```

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_HLE_NO_CSINK;

status = UserElemtreeElementAdd ( elem_hle_com, elem_hle_add_csink,
                                value_data );
UG_HOLE_LOG ( " UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_HOLEDIAM  \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_HOLEDIAM, &elem_hle_holediam );
UG_HOLE_LOG ( " ProElementAlloc PRO_E_HLE_HOLEDIAM ", status);

value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 0.05 ;

status = UserElemtreeElementAdd ( elem_hle_com, elem_hle_holediam,
                                value_data );
UG_HOLE_LOG ( " UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_DEPTH  \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_DEPTH, &elem_hle_depth );
UG_HOLE_LOG ( " ProElementAlloc PRO_E_HLE_DEPTH ", status);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_HLE_STD_THRU_ALL_DEPTH; /* PRO_HLE_STD_VAR_DEPTH
*/

status = UserElemtreeElementAdd ( elem_hle_com, elem_hle_depth,
                                value_data );
UG_HOLE_LOG ( " UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_CRDIR_FLIP  \n");
printf(" ***** \n");

status = ProElementAlloc (PRO_E_HLE_CRDIR_FLIP, &elem_hle_crdir_flip );
UG_HOLE_LOG ( " ProElementAlloc PRO_E_HLE_CRDIR_FLIP ", status);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_HLE_CR_IN_SIDE_ONE; /* PRO_HLE_CR_IN_SIDE_TWO */

status = UserElemtreeElementAdd ( elem_hle_com, elem_hle_crdir_flip,
                                value_data );
UG_HOLE_LOG ( " UserElemtreeElementAdd ", status);

/*-----*\
    Elements related to the placement details

```

```

| --PRO_E_HLE_PLACEMENT
| | --PRO_E_HLE_PRIM_REF
| | --PRO_E_HLE_PL_TYPE
| | --PRO_E_HLE_DIM_REF1
| | --PRO_E_HLE_DIM_DIST1
| | --PRO_E_HLE_DIM_REF2
| | --PRO_E_HLE_DIM_DIST2
\*-----*/

printf(" PRO_E_HLE_PLACEMENT \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_PLACEMENT,
                           &elem_hle_placement );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_PLACEMENT", status);

status = ProElemtreeElementAdd ( feat_elemtree, NULL,
                                elem_hle_placement );
UG_HOLE_LOG (" ProElemtreeElementAdd ", status);

printf(" PRO_E_HLE_PRIM_REF \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_PRIM_REF, &elem_hle_prim_ref );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_PRIM_REF ", status);

printf("Select a surface for Hole Placement \n");
status = ProSelect ( "datum,surface", 1, NULL, NULL, NULL, NULL,
                    &p_selection, &n_selection);
if ( n_selection <= 0 ) return (0);
UG_HOLE_LOG (" ProSelect ", status);

value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_selection[0] ;

status = UserElemtreeElementAdd ( elem_hle_placement,
                                elem_hle_prim_ref, value_data );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_PL_TYPE \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_PL_TYPE, &elem_hle_pl_type );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_PL_TYPE ", status);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_HLE_PL_TYPE_LIN;

status = UserElemtreeElementAdd ( elem_hle_placement, elem_hle_pl_type,
                                value_data );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status);

```

```

printf(" PRO_E_HLE_DIM_REF1 \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_DIM_REF1, &elem_hle_dim_ref1 );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_DIM_REF1 ", status);
printf("Select a 1st Reference for Hole Placement \n");
status = ProSelect ( "datum,surface,edge", 1, NULL, NULL, NULL, NULL,
                    &p_selection, &n_selection);

if ( n_selection <= 0 ) return (0);
UG_HOLE_LOG (" ProSelect ", status);

value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_selection[0] ;

status = UserElemtreeElementAdd ( elem_hle_placement,
                                elem_hle_dim_ref1, value_data );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_DIM_DIST1 \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_DIM_DIST1, &elem_hle_dim_dist1 );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_DIM_DIST1 ", status);

value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 0.1;

status = UserElemtreeElementAdd ( elem_hle_placement,
                                elem_hle_dim_dist1, value_data );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status);

printf(" PRO_E_HLE_DIM_REF2 \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_DIM_REF2, &elem_hle_dim_ref2 );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_DIM_REF2 ", status);

printf("Select a 2st Reference for Hole Placement \n");
status = ProSelect ( "datum,surface,edge", 1, NULL, NULL, NULL, NULL,
                    &p_selection, &n_selection);
if ( n_selection <= 0 ) return (0);
UG_HOLE_LOG (" ProSelect ", status);

value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_selection[0] ;

status = UserElemtreeElementAdd ( elem_hle_placement,
                                elem_hle_dim_ref2, value_data );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status);

```

```

printf(" PRO_E_HLE_DIM_DIST2 \n");
printf(" ***** \n");

status = ProElementAlloc ( PRO_E_HLE_DIM_DIST2, &elem_hle_dim_dist2 );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_DIM_DIST2 ", status);

value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 0.2;

status = UserElemtreeElementAdd ( elem_hle_placement,
                                elem_hle_dim_dist2, value_data );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status);

/*-----*\
                                End of Element Tree Creation
\*-----*/

/*-----*\
                                Start of Feature Creation
\*-----*/

status = ProMdlCurrentGet ( &model );
UG_HOLE_LOG (" ProMdlCurrentGet ", status);

status = ProMdlToModelitem( model, &model_item );
UG_HOLE_LOG (" ProMdlToModelitem ", status);

status = ProSelectionAlloc( NULL, &model_item, &model_selection );
UG_HOLE_LOG (" ProSelectionAlloc ", status);

status = ProFeatureCreate ( model_selection, feat_elemtree,
                           options, 1,
                           &created_feature, &p_errors );
UG_HOLE_LOG (" ProFeatureCreate ", status);

/*-----*\
                                End of Feature Creation
\*-----*/

/*-----*\
                                Freeing the resources
\*-----*/

status = ProElementFree( &feat_elemtree );
UG_HOLE_LOG (" ProElementFree ", status);

status = ProSelectionFree( &model_selection );
UG_HOLE_LOG (" ProSelectionFree ", status);

/*-----*\
                                End of Freeing
\*-----*/

```

```

        return ( status );
    }
/*=====*\
FUNCTION : UserElemtreeElementAdd
PURPOSE  : Adding an element to the parent element / tree
input    : parent_element, child_element, value_data
output   : none
returns  :
    PRO_TK_NO_ERROR - Successfully added the element
    PRO_TK_GENERAL_ERROR - Failed to add the element
    any other return as returned by ProElemtreeElementAdd()
\*=====*/

ProError UserElemtreeElementAdd( ProElement parent_element,
                                ProElement child_element,
                                ProValueData value_data )
{
    ProValue value;
    ProError status;

    status = ProValueAlloc ( &value );
    UG_HOLE_LOG ( " ProValueAlloc ", status);

    status = ProValueDataSet ( value, &value_data );
    UG_HOLE_LOG ( " ProValueDataSet ", status);

    status = ProElementValueSet ( child_element, value );
    UG_HOLE_LOG ( " ProElementValueSet ", status);

    status = ProElemtreeElementAdd ( parent_element, NULL,
                                    child_element );
    UG_HOLE_LOG ( " ProElemtreeElementAdd ", status);

    return ( status );
}

```

Custom Hole

The following table outlines the elements for creating a custom hole.

Custom Hole Elements

Element	Status
PRO_E_HLE_TYPE_NEW	Mandatory
PRO_E_HLE_ADD_CBORE	Mandatory
PRO_E_HLE_ADD_CSINK	Mandatory
PRO_E_HLE_HOLEDIAM	Mandatory

Custom Hole Elements

PRO_E_HLE_DRILLANGLE	Required for variable depth hole
PRO_E_HLE_CSINKANGLE	Required for countersink option
PRO_E_HLE_CBOREDEPTH	Required for counterbore option
PRO_E_HLE_CBOREDIAM	Required for counterbore option
PRO_E_HLE_CSINKDIAM	Required for countersink option
PRO_E_HLE_DEPTH	Mandatory
PRO_E_HLE_DEPTH_DIM_TYPE	Depends on PRO_E_HLE_DEPTH
PRO_E_HLE_CRDIR_FLIP	Mandatory
PRO_E_HLE_ADD_EXIT_CSINK	Required for Thru all hole
PRO_E_HLE_EXIT_CSINKANGLE	Required for exit countersink option
PRO_E_HLE_EXIT_CSINKDIAM	Required for exit countersink option

Valid PRO_E_HLE_COM Sub-Elements

The following table gives the description of all the elements for all the hole types.

Hole Elements Description

Element ID	Comment/Description
Straight Hole	
PRO_E_HLE_TYPE_NEW = PRO_HLE_NEW_TYPE_STRAIGHT	
PRO_E_DIAMETER	Stores the diameter double value
PRO_E_HOLE_STD_DEPTH	Depth (compound element)
PRO_E_HOLE_DEPTH_TO	First Side depth info (compound element)
PRO_E_HOLE_DEPTH_TO_TYPE	type ProHleStraightDepType /* Blind*/ PRO_HLE_STRGHT_BLIND_DEPTH /*Thru Next*/ PRO_HLE_STRGHT_THRU_NEXT_DEPTH /* Thru All*/ PRO_HLE_STRGHT_THRU_ALL_DEPTH /*Thru Until*/ PRO_HLE_STRGHT_THRU_UNTIL_DEPTH /*Upto Ref*/ PRO_HLE_STRGHT_UPTO_REF_DEPTH /*None */ PRO_HLE_STRGHT_NONE_DEPTH /*Symmetric*/ PRO_HLE_STRGHT_SYM_DEPTH

Hole Elements Description

PRO_E_EXT_DEPTH_TO_VALUE	Stores variable depth double value when PRO_E_HOLE_DEPTH_TO_TYPE equals PRO_HLE_STRGHT_BLIND_DEPTH
PRO_E_EXT_DEPTH_TO_REF	Stores the upto reference when PRO_E_HOLE_DEPTH_TO_TYPE is not PRO_HLE_STRGHT_BLIND_DEPTH and not PRO_HLE_STRGHT_NONE_DEPTH.
PRO_E_HOLE_DEPTH_FROM	Second Side depth info (compound element)
PRO_E_HOLE_DEPTH_FROM_TYPE	type ProHleStraightDep /* Blind*/ PRO_HLE_STRGHT_BLIND_DEPTH /*Thru Next*/ PRO_HLE_STRGHT_THRU_NEXT_DEPTH /* Thru All*/ PRO_HLE_STRGHT_THRU_ALL_DEPTH /*Thru Until*/ PRO_HLE_STRGHT_THRU_UNTIL_DEPTH /*Upto Ref*/ PRO_HLE_STRGHT_UPTO_REF_DEPTH /*None */ PRO_HLE_STRGHT_NONE_DEPTH /*Symmetric*/ PRO_HLE_STRGHT_SYM_DEPT
PRO_E_EXT_DEPTH_FROM_VALUE	Stores variable depth double value when PRO_E_HOLE_DEPTH_FROM_TYPE equals PRO_HLE_STRGHT_BLIND_DEPTH
PRO_E_EXT_DEPTH_FROM_REF	Stores the upto reference when PRO_E_HOLE_DEPTH_FROM_TYPE is not PRO_HLE_STRGHT_BLIND_DEPTH and is not PRO_HLE_STRGHT_NONE_DEPTH and not PRO_HLE_STRGHT_SYM_DEPTH
Sketch Hole	
PRO_E_HLE_TYPE_NEW set to PRO_HLE_NEW_TYPE_SKETCH	
PRO_E_HLE_SKETCHER	2D Sketcher Element
PRO_E_HLE_CRDIR_FLIP	Direction of creation, type ProHleCrDir
Standard Hole	
PRO_E_HLE_TYPE_NEW set to PRO_HLE_NEW_TYPE_STANDARD	
PRO_E_STAN_TYPE = PRO_HLE_TAPPED_TYPE /* Tapped hole */ = PRO_HLE_CLEARANCE_TYPE /* Clearance hole */	
PRO_E_HLE_STAN_TYPE	type ProHleStandType

Hole Elements Description

PRO_E_HLE_THRDSERIS	Integer. The *.hol files get loaded as specified in Hole Parameter Files. From the *.hol files, different THREAD_SERIES information are gathered and a list is formed. This element stores the current index to the list.
PRO_E_HLE_FITTYPE	type ProHleFittype. Available for clearance hole (when PRO_E_HLE_STAN_TYPE is PRO_HLE_CLEARANCE_TYPE) /* Close Fit */ PRO_HLE_CLOSE_FIT /* Free Fit */ PRO_HLE_FREE_FIT /* Medium Fit */ PRO_HLE_MEDIUM_FIT
PRO_E_HLE_SCREWSIZE	Integer Stores an index to the screw_size list. Selecting a thread series, choose one of the.hol files. From that file screw-size list is extracted.
PRO_E_HLE_DEPTH	It is an option for different type drill depth, that is, of type ProHleStdDepType. Note: PRO_HLE_STD_VAR_DEPTH is not available for clearance hole (not for PRO_E_HLE_STAN_TYPE == PRO_HLE_CLEARANCE_TYPE).
PRO_E_STD_HOLE_DEPTH_REF	Stores reference, when PRO_E_HLE_DEPTH equals to PRO_HLE_STD_THRU_UNTIL_DEPTH or PRO_HLE_STD_TO_SEL_DEPTH
PRO_E_HLE_HOLEDIAM	Stores Drill Diameter double value. See Hole Diameter.
PRO_E_HLE_DRILLANGLE	Stores Drill Angle. Double value. Available for tapped hole with variable depth (when PRO_E_HLE_STAN_TYPE equals PRO_HLE_TAPPED_TYPE and PRO_E_HLE_DEPTH is PRO_HLE_STD_VAR_DEPTH).
PRO_E_HLE_ADD_THREAD	Option for adding thread. Available for tapped hole (when PRO_E_HLE_STAN_TYPE equals PRO_HLE_TAPPED_TYPE). Type ProHleAddThrdFlag. For add thread option it's value is PRO_HLE_ADD_THREAD. For no thread option, the value is PRO_HLE_NO_THREAD.

Hole Elements Description

PRO_E_HLE_THRD_DEPTH	Option for different type of thread depth. Type ProHleThrdDepType. Available for tapped hole with thread option, when PRO_E_HLE_STAN_TYPE equals PRO_HLE_TAPPED_TYPE and PRO_E_HLE_ADD_THREAD equals PRO_HLE_ADD_THREAD. Note: For part level, variable drill depth, and assembly level tapped threaded hole, the PRO_HLE_THRU_THREAD value for this element is invalid.
PRO_E_HLE_THRDDEPTH	Stores thread depth. Double value. Available for tapped hole, with variable thread option. That is, when PRO_E_HLE_STAN_TYPE equals PRO_HLE_TAPPED_TYPE, PRO_E_HLE_ADD_THREAD equals PRO_HLE_ADD_THREAD, and PRO_E_HLE_THRD_DEPTH equals PRO_HLE_VARIABLE_THREAD.
PRO_E_HLE_ADD_CBORE	Option for Counter Bore. Type ProHleAddCboresFlag. For counter bore it's value is PRO_HLE_ADD_CBORE. For the no counterbore option, set to PRO_HLE_NO_CBORE.
PRO_E_HLE_CBOREDEPTH	Stores counterbore depth. Double value. Available for counterbore option, when PRO_E_HLE_ADD_CBORE equals PRO_HLE_ADD_CBORE.
PRO_E_HLE_CBOREDIAM	Stores counterbore diameter. Double value. Available for counterbore option, when PRO_E_HLE_ADD_CBORE is PRO_HLE_ADD_CBORE.
PRO_E_HLE_ADD_CSINK	It is an option for Counter Sink. Type ProHleAddCsinkFlag. For counter sink it's value is PRO_HLE_ADD_CSINK. For no countersink, set to PRO_HLE_NO_CSINK.
PRO_E_HLE_CSINKANGLE	Stores counter sink angle. Double value. Available for countersink option, when PRO_E_HLE_ADD_CSINK equals PRO_HLE_ADD_CSINK.
PRO_E_HLE_CSINKDIAM	Stores countersink diameter. Double value. Available for countersink option, when PRO_E_HLE_ADD_CSINK equals PRO_HLE_ADD_CSINK.

Hole Elements Description

PRO_E_HLE_DRILLDEPTH	Stores drill depth double value. Available for tapped hole, with variable depth option. That is, when PRO_E_HLE_STAN_TYPE equals PRO_HLE_TAPPED_TYPE, and PRO_E_HLE_DEPTH equals PRO_HLE_STD_VAR_DEPTH.
PRO_E_HLE_ADD_EXIT_CSINK	An option for Exit Counter Sink of type ProHleAddExitCsinkFlag. For exit counter sink it's value is PRO_HLE_ADD_EXIT_CSINK. For no countersink, value is PRO_HLE_NO_EXIT_CSINK It is not available for assembly mode. In part mode will fail if entry and exit surfaces of hole are non-planar and non-parallel.
PRO_E_HLE_EXIT_CSINKANGLE	Stores exit countersink angle double value. Available for exit countersink option, that is, PRO_E_HLE_ADD_EXIT_CSINK == PRO_HLE_ADD_EXIT_CSINK.
PRO_E_HLE_EXIT_CSINKDIAM	Stores exit countersink diameter double value. Available for exit countersink option, that is, PRO_E_HLE_ADD_EXIT_CSINK == PRO_HLE_ADD_EXIT_CSINK.
PRO_E_HLE_ADD_NOTE	It is an option for add note. Of type ProHleAddNoteFlag. The default value is add note, i.e. PRO_HOLE_ADD_NOTE_FLAG. For no note, the value is PRO_HOLE_NO_NOTE_FLAG.
PRO_E_HOLE_NOTE	This element is not accessible through Pro/TOOLKIT. Default note will be created when PRO_E_HLE_ADD_NOTE is set to PRO_HOLE_ADD_NOTE_FLAG.
Custom Hole	
PRO_E_HLE_TYPE_NEW	PRO_HLE_CUSTOM_TYPE

Hole Elements Description

PRO_E_HLE_ADD_CBORE	The description of these items are same as described in Standard Hole section.
PRO_E_HLE_ADD_CSINK	
PRO_E_HLE_HOLEDIAM	
PRO_E_HLE_DRILLANGLE	
PRO_E_HLE_CSINKANGLE	
PRO_E_HLE_CBOREDEPTH	
PRO_E_HLE_CBOREDEPTH	
PRO_E_HLE_CSINKDIAM	
PRO_E_HLE_DRILLDEPTH	
PRO_E_HLE_DEPTH	
PRO_E_STD_HOLE_DEPTH_REF	
PRO_E_HLE_DEPTH_DIM_TYPE	
PRO_E_HLE_CRDIR_FLIP	
PRO_E_HLE_ADD_EXIT_CSINK	
PRO_E_HLE_EXIT_CSINKANGLE	
PRO_E_HLE_EXIT_CSINKDIAM	

Hole Placement Types

Pro/TOOLKIT supports several placement types for holes.

Hole Placement

The elements discussed in the following sections specify how to place a hole in relation to the model geometry. The reference entity elements are carried as selection objects, and the other elements carrying actual values of distances, offsets or angles.

Pro/TOOLKIT supports the following types of hole placement:

- Linear Hole on a Plane
- Radial Hole on Plane with Radial Dimensioning
- Radial Hole on Plane with Diameter Dimensioning
- Radial Hole on Plane with Linear Dimensioning
- Radial Hole on Cone or Cylinder
- Coaxial Hole with Axis as Primary Reference
- Coaxial Hole with Primary Reference not Axis

- Onpoint Hole

Linear Hole on a Plane

Linear placement requires as references either

- Two linear non-parallel edges in the plane of placement
- or
- Two planar non-parallel surfaces, both normal to the plane of placement.

PRO_E_HLE_PRIM_REF	Primary selection, that is, planar surface or datum plane.
PRO_E_HLE_PL_TYPE	Set to PRO_HLE_PL_TYPE_LIN
PRO_E_HLE_DIM_REF1	First secondary selection, that is, plane, edge, or axis. If edge or axis is normal to placement plane, another selection is required for dimensioning the hole. So this may require two selections.
PRO_E_HLE_PLC_ALIGN_OPT1	- Set to PRO_HLE_PLC_ALIGN to align the hole to the reference. - Set to PRO_HLE_PLC_NOT_ALIGN to use the DIST1 reference.
PRO_E_HLE_DIM_DIST1	Distance with regard to PRO_E_HLE_DIM_REF1.
PRO_E_HLE_DIM_REF2	Second secondary selection, that is, plane, edge, or axis. - If edge or axis is normal to placement plane another selection is required for dimensioning the hole. So this may require two selections.
PRO_E_HLE_PLC_ALIGN_OPT2	- Set to PRO_HLE_PLC_ALIGN to align the hole to the reference. - Set to PRO_HLE_PLC_NOT_ALIGN to use the DIST2 reference.
PRO_E_HLE_DIM_DIST2	Distance with regard to PRO_E_HLE_DIM_REF2.
PRO_E_LIN_HOLE_DIR_REF	Uses this reference to define the direction of the placement dimension scheme. This element is available if the secondary element PRO_E_HLE_DIM_REF1 contains an axis reference normal to the current hole's primary reference.

Radial Hole on Plane with Radial Dimensioning

Locating radial holes requires the first reference to be the axis of placement. This axis is a polar placement (r-theta), where r is the radial distance from a plane, and theta is the angle with respect to a plane.

PRO_E_HLE_PRIM_REF	Primary Selection, planar surface/datum plane
PRO_E_HLE_PL_TYPE set to PRO_HLE_PL_TYPE_RAD	
PRO_E_HLE_AXIS	Axis for radial hole
PRO_E_HLE_DIM_RAD	Radial distance with regard to PRO_E_HLE_AXIS
PRO_E_HLE_REF_PLANE_1	Reference plane against which angular distance will be measured
PRO_E_HLE_REF_ANG_1	Angular distance with regard to PRO_E_HLE_REF_PLANE_1

Radial Hole on Plane with Diameter Dimensioning

Locating these holes is similar to radial holes with radial dimensioning. The difference is the distance specified is in the form of diametrical distance.

PRO_E_HLE_PRIM_REF	Primary Selection, planar surface/datum plane
PRO_E_HLE_PL_TYPE set to PRO_HLE_PL_TYPE_RAD_DIA_DIM	
PRO_E_HLE_AXIS	Axis for radial hole
PRO_E_HLE_DIM_DIA	Diameter distance with regard to PRO_E_HLE_AXIS
PRO_E_HLE_REF_PLANE_1	Reference plane against which angular distance will be measured
PRO_E_HLE_REF_ANG_1	Angular distance with regard to PRO_E_HLE_REF_PLANE_1

Radial Hole on Plane with Linear Dimensioning

This type of hole placement uses an angle with respect to a plane and a linear distance from the axis of placement.

This placement type is available when you set the configuration option radial_hole_linear_dim to "YES"

PRO_E_HLE_PRIM_REF	Primary Selection, planar surface/datum plane
PRO_E_HLE_PL_TYPE is set to PRO_HLE_PL_TYPE_RAD_LIN_DIM	

PRO_E_HLE_PRIM_REF	Primary Selection, planar surface/datum plane
PRO_E_HLE_AXIS	Axis for radial hole
PRO_E_HLE_DIM_LIN	Linear distance with regard to PRO_E_HLE_AXIS
PRO_E_HLE_REF_PLANE_1	Reference plane against which angular distance will be measured
PRO_E_HLE_REF_ANG_1	Angular distance with regard to PRO_E_HLE_REF_PLANE_1

Radial Hole on Cone or Cylinder

This hole placement type requires the selection of a cone or cylinder for primary placement.

PRO_E_HLE_PRIM_REF	Primary Selection, Cone or Cylinder
PRO_E_HLE_PL_TYPE is set to PRO_HLE_PL_TYPE_RAD	
PRO_E_HLE_REF_PLANE	Reference plane against which angular distance will be measured
PRO_E_HLE_REF_ANG	Angular distance with regard to PRO_E_HLE_REF_PLANE
PRO_E_HLE_NORM_PLA	Reference plane for linear measurement
PRO_E_HLE_NORM_OFFST	Distance with regard to PRO_E_HLE_NORM_PLA

Coaxial Hole with Axis as Primary Reference

Coaxial hole placement requires an axis and a placement plane to complete the placement.

PRO_E_HLE_PRIM_REF	Primary Selection, Axis
PRO_E_HLE_PL_TYPE is set to PRO_HLE_PL_TYPE_COAX	
PRO_E_HLE_PLCMNT_PLANE	Placement surface

Coaxial Hole with Primary Reference not Axis

This is a special case of coaxial hole in which the primary selection is an axis and either plane, cone, or cylinder. The axis must be normal to the selected surface.

PRO_E_HLE_PRIM_REF	Primary Selection, Plane, Cone or Cylinder
PRO_E_HLE_PL_TYPE is set to PRO_HLE_PL_TYPE_COAX	
PRO_E_HLE_AXIS	Axis

Onpoint Hole

This placement type requires a point of type 'On Surface Point'. The hole is placed normal to the surface on which the point was created. The hole passes through the selected point.

PRO_E_HLE_PRIM_REF	Primary Selection, on Surface Created Datum Point
PRO_E_HLE_PL_TYPE is set to PRO_HLE_PL_TYPE_ON_PNT	

Miscellaneous Information

The following sections discuss important issues relating to hole feature creation.

Hole Parameter Files

Hole parameter files are setup files used to build the user interface for the hole. Programmatic hole creation uses the same files. New sets of customized files can be added as required. The values assigned to the elements PRO_E_HLE_THRDSERIS, PRO_E_HLE_SCREWSIZE, and therefore PRO_E_HLE_HOLEDIAM depend on these files.

Pro/ENGINEER and Pro/TOOLKIT load the hole parameter file (*.hol) in following order:

1. Directory specified in configuration option
hole_parameter_file_path
2. Current Directory
3. System hole parameter directory, that is,
[PROE DIR]/text/hole

Find the hole diameter from the values of PRO_E_HLE_THRDSERIS and PRO_E_HLE_SCREWSIZE specified in the *.hol files. In the Pro/ENGINEER User Interface, element PRO_E_HLE_THRDSERIS is represented as the selection between UNC, UNF or ISO.

Hole Diameter

The drill diameter `PRO_E_HLE_HOLEDIAM`, as required for the Standard Type of holes, must be smaller than the thread diameter calculated from the *.hol* file for the threaded hole. As specified in the **.hol* files, the thread diameter is the element corresponding to `BASIC_DIAM` column and the selected screw size row in the table, as specified in the selected *.hol* file. If the `PRO_E_HLE_HOLEDIAM` is not smaller than the thread diameter, the **ProFeatureCreate()** function fails and returns a `PRO_TK_GENERAL_ERROR`.

Follow these steps to enter the proper value for `PRO_E_HLE_HOLEDIAM`:

1. Determine the values to pass from the Pro/ENGINEER user interface to the following elements:
 - `PRO_E_HLE_THRDSERIS`. Note that UNC corresponds to 0, UNF to 1, and ISO to 2. These values change if you create a local *.hol* file.
 - `PRO_E_HLE_SCREWSIZE` (the values start with zero).
2. From the Pro/ENGINEER User Interface, set the options to be passed to the elements `PRO_E_HLE_THRDSERIS` and `PRO_E_HLE_SCREWSIZE`. For example, ISO with M1X25 or UNC with 1-64.
3. Observe the value hole diameter in the dialog box. The dialog box appears grayed out unless you set the configuration option `hole_diameter_override` to *yes*.
4. The value thus obtained for the hole diameter should be greater than the value defined for element `PRO_E_HLE_HOLEDIAM`

Order of Element Specification

Be sure to enter the elements into the element tree in the order specified by the tables in `PRO_E_HLE_COM` Values. Failure to follow these sequences may result in either **ProFeatureCreate()** failing with a `PRO_TK_GENERAL_ERROR` error return, or in creation of a feature which fails to get redefined.

Hole-specific Functions

Functions Introduced:

- **ProHolePropertyGet()**

You can use the function **ProHolePropertyGet()** to retrieve the value of the indicated hole value property. Only properties listed in the enum *ProHoleProperty* are supported.

Element Trees: Shell

This chapter introduces and shows how to create, redefine and access Shell features in Pro/TOOLKIT.

Topic	Page
Introduction to Shell Feature	38 - 2
Feature Element Tree for the Shell Feature	38 - 3
Creating a Shell Feature	38 - 5
Redefining a Shell Feature	38 - 5
Accessing a Shell Feature	38 - 5

Introduction to Shell Feature

When Pro/ENGINEER makes a shell, all features that were added to the solid before creating the Shell feature are hollowed. Therefore, the order of feature creation is very important when you use the Shell feature.

The Shell feature hollows the inside of the solid, leaving a shell of a specified wall thickness. It allows you to remove a surface or surfaces from the shell. If you do not select a surface to remove, a “closed shell” is created, with the inside of the part completely hollowed out and no access to the hollow. When defining a shell, you can also select surfaces with different thickness values. On flipping the thickness side, the shell thickness is added to the outside of the part.

You can also shell surfaces that are tangent to their neighbors at one or more boundaries. At the tangent edge where the separation of the shell offset occurs, a normal capping surface is constructed to close the gap.

You can also exclude one or more surfaces from being shelled. This process is called partial shelling. There are two different algorithms for partial shelling – one for concave corner surfaces and the other for convex corner surfaces. These algorithms prevent the shell subtraction volume from penetrating through the solid. In case of a part where both concave and convex corner surfaces are to be excluded, the exclusion can be achieved in multiple partial shells, each using different algorithms.

The following are the restrictions on Shell feature creation:

- You cannot add shells to any part that has a surface that moves from tangency to a point.
- You cannot select a surface to be removed that has a vertex created by the intersection of three curved surfaces.
- The surface to be removed must be surrounded by edges (a fully revolved surface of revolution is not valid) and the surfaces that intersect the edge must form an angle of less than 180 degrees through the solid geometry. As long as this condition is met, you can select any sculpted surface as the surface to be removed.
- When you select surfaces that have other surfaces tangent to them for independent thickness, all surfaces that are tangent must have the same thickness. Otherwise, the Shell feature fails.

For example, if you shell a part that contains a hole and you want the thickness of the hole wall to be different from the overall thickness, you must select both surfaces (cylinders) that make up the hole and offset them at the same distance.

- By default, a shell creates geometry with a constant wall thickness. If the system cannot create a constant thickness, the Shell feature fails.

Feature Element Tree for the Shell Feature

The element tree for the Shell feature is documented in the header file *ProShell.h*, and has a simple structure. The following figure demonstrates the element tree structure:

Figure 38-1: Feature Element Tree for Shell Feature

```

PRO_E_FEATURE_TREE
|
|-- PRO_E_FEATURE_TYPE
|
|-- PRO_E_STD_FEATURE_NAME
|
|-- PRO_E_SHELL_SRF
|
|-- PRO_E_SHELL_THICK
|
|-- PRO_E_SHELL_FLIP
|
|-- PRO_E_STD_SURF_COLLECTION_APPL
|
|-- PRO_E_SHELL_LACE_BNDRY
|
|-- PRO_E_SHELL_ALT_CUT_METHOD
|
|-- PRO_E_ST_SHELL_LOCL_LIST
|   |
|   |-- PRO_E_ST_SHELL_LOCL_CMPD
|       |
|       |-- PRO_E_ST_SHELL_SPEC_SRF
|       |
|       |-- PRO_E_ST_SHELL_SPEC_THCK

```

The shell element tree contains standard element types. The following list details special information about the elements in this tree:

- PRO_E_FEATURE_TYPE— Specifies the feature type and should have the value PRO_FEAT_SHELL.
- PRO_E_STD_FEATURE_NAME—Specifies the name of the Shell feature.
- PRO_E_SHELL_SRF— Specifies the selected surfaces to be removed from the part to create the Shell feature. This element is optional.
- PRO_E_SHELL_THICK— Specifies the default thickness for the shell. It must be positive and greater than zero.
- PRO_E_SHELL_FLIP— Specifies the side of the shell to be flipped and has the following values:
 - PRO_SHELL_OUTSIDE
 - PRO_SHELL_INSIDE
- PRO_E_STD_SURF_COLLECTION_APPL—Specifies the surfaces to be excluded during the Shell feature creation.
- PRO_E_SHELL_LACE_BNDRY—Specifies the lace boundary that forms the closure of excluded surfaces or inner surfaces. It has the following values:
 - PRO_SHELL_LACE
 - PRO_SHELL_DONT_LACE
- PRO_E_SHELL_ALT_CUT_METHOD—Specifies the alternate cut method used for partial shell volume subtraction. It has the following values:
 - PRO_SHELL_ALT_CUT_METHOD_NO—Specifies the algorithm for concave corner surfaces.
 - PRO_SHELL_ALT_CUT_METHOD_YES—Specifies the algorithm for convex corner surfaces.
- PRO_E_ST_SHELL_LOCL_LIST—Specifies an array of local thickness of the type PRO_E_ST_SHELL_LOCL_CMPD which consists of the following elements:
 - PRO_E_ST_SHELL_SPEC_SRF—Specifies the surface selected for specifying the local thickness value. This surface cannot be one of the surfaces selected to be removed.

- **PRO_E_ST_SHELL_SPEC_THICK**—Specifies the local thickness value of the selected surface (initially equal to default shell thickness). It must be positive and greater than zero.

Creating a Shell Feature

Function Introduced

- **ProFeatureCreate()**

Use the function **ProFeatureCreate()** to create a Shell feature based on the element tree input. For more information about **ProFeatureCreate()**, refer to the section “Overview of Feature Creation” of chapter “Element Trees: Principles of Feature Creation”.

Redefining a Shell Feature

Function Introduced

- **ProFeatureRedefine()**

Use the function **ProFeatureRedefine()** to redefine a Shell feature based on the changes made in the element tree. For more information about **ProFeatureRedefine()**, refer to the section “Feature Redefine” of chapter “Element Trees: Principles of Feature Creation”.

Accessing a Shell Feature

Function Introduced

- **ProFeatureElemtreeExtract()**

Use the function **ProFeatureElemtreeExtract()** to create a feature element tree that describes the contents of a Shell feature and to retrieve the element tree description of a Shell feature. For more information about **ProFeatureElemtreeExtract()**, refer to the section “Feature Inquiry” of chapter “Element Trees: Principles of Feature Creation”.

Element Trees: Patterns

This chapter describes the element tree structure required to create patterns of features. The chapter on 'Element Trees: Principles of Feature Creation' is a necessary background for this topic. Read that chapter before this one.

Topic	Page
Introduction	39 - 2
The Element Tree for Pattern Creation	39 - 2
Obtaining the Element Tree for a Pattern	39 - 22
Visiting and Creating a Pattern	39 - 23

Introduction

Using Pro/TOOLKIT, you can create patterns of features, including those not supported by Pro/TOOLKIT feature creation. Consequently, you can programmatically create patterns of any feature that can be patterned in Pro/ENGINEER.

The creation and manipulation of patterns use the following Pro/TOOLKIT objects:

- *ProPattern*—A structure that contains the type and owner of the pattern, and an opaque pattern handle
- *ProPatternClass*—An enumerated type that contains the pattern class, which specifies either a feature pattern (PRO_FEAT_PATTERN) or a group pattern (PRO_GROUP_PATTERN)

The procedure for creating a pattern is similar to creating features, in that you construct an element tree and pass this element tree to Pro/ENGINEER. When you pass the tree to Pro/ENGINEER, you also specify the feature to be patterned.

The Element Tree for Pattern Creation

Unlike the element tree for features, the element tree for a pattern does not contain information about the construction of new features. Rather, the element tree contains information needed to make copies of existing features at specified locations on the model. For example, the element tree for a pattern of holes does not contain the geometry (such as edges) used to place the holes, but contains the dimensions and dimension variations used to pattern the specified hole.

You construct the element tree for a pattern by following the procedure described in chapter 'Element Trees: Principles of Feature Creation':

1. Allocate tree elements using the function **ProElementAlloc()**.
2. Set values of the elements using the function **ProElementValueSet()**.
3. Add elements to the tree using **ProElemtreeElementAdd()**.

As with feature creation, the system cannot create your pattern unless the element tree is correct.

The element tree for a pattern is documented in the header file *ProPattern.h*. This tree contains the same information required when you create a pattern in an interactive session of Pro/ENGINEER. Therefore, you should be familiar with how to create a pattern interactively before you try to understand the element tree.

Note: It is highly recommended that you use the new element tree from *ProPattern.h*. The old element tree is only for your reference.

The following figure shows a part of the element tree for patterns. All elements are described in detail in the following sections.

Figure 39-1: A Part of the Element Tree for Patterns

Pattern element tree

```

PRO_E_PATTERN_ROOT
|
|--- PRO_E_GENPAT_TYPE
|
|--- PRO_E_GENPAT_REGEN_METHOD
|
|--- PRO_E_STD_FEATURE_NAME
|
|      * Dim Pattern *
|--- PRO_E_GENPAT_DIM
|
|      * Table Pattern *
|--- PRO_E_GENPAT_TABLE
|
|      * Ref Pattern *
|--- PRO_E_GENPAT_REF
|
|      * Fill Pattern *
|--- PRO_E_GENPAT_FILL
|
|      * Curve Pattern *
|--- PRO_E_GENPAT_CURVE
|
|      * Dir Pattern *
|--- PRO_E_GENPAT_DIR
|
|      * Axis Pattern *
|--- PRO_E_GENPAT_AXIS
|
|      * Point Pattern *
|--- PRO_E_GENPAT_POINT
|
|--- PRO_E_STD_SECTION
|
|--- PRO_E_PAT_APPLICATIONS
|
|      |-- PRO_E_PAT_MFG_TOOL_PATH
|      |
|      |      |-- PRO_E_PAT_MFG_ORD_LEADER

```

The element with the identifier `PRO_E_GENPAT_TYPE` sets the type of the pattern to be created. The structure of the rest of the element tree depends strongly on the value of this element. Valid values for the `PRO_E_GENPAT_TYPE` element are as follows:

- `PRO_GENPAT_REF_DRIVEN`—Reference pattern
- `PRO_GENPAT_DIM_DRIVEN`—Dimension pattern
- `PRO_GENPAT_TABLE_DRIVEN`—Table pattern
- `PRO_GENPAT_FILL_DRIVEN`—Fill pattern
- `PRO_GENPAT_DIR_DRIVEN`—Direction pattern
- `PRO_GENPAT_AXIS_DRIVEN`—Axis pattern
- `PRO_GENPAT_POINT_DRIVEN`—Point pattern
- `PRO_GENPAT_CRV_DRIVEN`—Curve pattern

The element with the identifier `PRO_E_GENPAT_REGEN_METHOD` sets the regeneration method for the pattern. The regeneration method varies with the complexity of the pattern. Valid values for the `PRO_E_GENPAT_REGEN_METHOD` element are as follows:

- `PRO_PAT_GENERAL`—General pattern. This is the most complex type of pattern.
- `PRO_PAT_VARYING`—Varying pattern.
- `PRO_PAT_IDENTICAL`—Identical pattern. This is the least complex type of pattern.

For more information on regeneration methods, see the *Part Modeling User's Guide*.

You must populate the element `PRO_E_STD_SECTION` with a valid reference sketch and other related elements for the following pattern types:

- a point pattern that uses an element of type `PRO_GENPAT_REF_SKETCH`
- a fill pattern
- a curve pattern

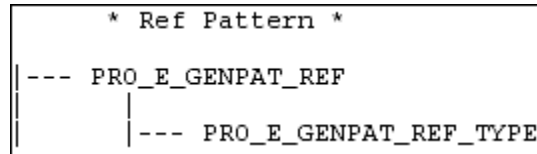
For more information on how to populate the section elements with the valid sketched reference, refer to the 'Element Trees: Sketched Features' chapter.

The following sections describe the types of pattern in more detail.

Reference Patterns

A reference pattern uses an existing pattern as a guide for the placement of the new pattern members. Consequently, if the pattern type is `PRO_GENPAT_REF_DRIVEN`, the element tree requires only that you specify the type of the reference pattern.

Figure 39-2: Element Tree for a Reference Pattern



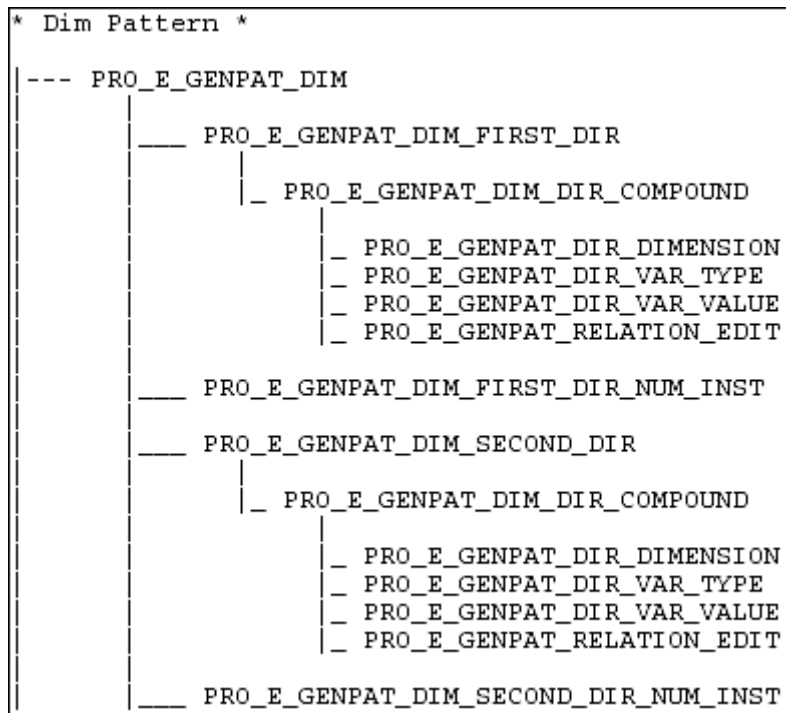
The element with identifier `PRO_E_GENPAT_REF` specifies the type of pattern to be created. The valid values are as follows:

- `PRO_PAT_FEATURE`—Use feature pattern references.
- `PRO_PAT_GROUP`—Use group pattern references.
- `PRO_PAT_BOTH`—Use feature and group pattern references.

Dimension Patterns

If the pattern type is `PRO_GENPAT_DIM_DRIVEN`, the element tree must include information about the dimensions used to drive the pattern. You must specify this information for each direction in which the feature is to be patterned.

Figure 39-3: Element Tree for a Dimension Pattern



Element Trees:
Patterns

The elements with identifiers PRO_E_GENPAT_DIM_FIRST_DIR and PRO_E_GENPAT_DIM_SECOND_DIR contain information about the pattern dimensions.

These elements are array elements that contain as many PRO_E_GENPAT_DIM_DIR_COMPOUND elements as are required to complete the pattern. The following table describes the contents of the PRO_E_GENPAT_DIM_DIR_COMPOUND element.

Element ID Values	Element Name	Data Type	Valid Values
PRO_E_GENPAT_DIR_DIMENSION	Dimension	PRO_VALUE_TYPE_SELECTION	
PRO_E_GENPAT_DIR_VAR_TYPE	Variation type	PRO_VALUE_TYPE_INT	PRO_PAT_RELATION_DRIVEN, PRO_PAT_VALUE_DRIVEN

Element ID Values	Element Name	Data Type	Valid Values
PRO_E_GENPAT_DIR_VAR_VALUE	Variation value (increment)	PRO_VALUE_TYPE_DOUBLE	
PRO_E_GENPAT_RELATION_EDIT	Relation	Application (PRO_VALUE_TYPE_POINTER)	

The element PRO_E_GENPAT_DIR_VAR_TYPE specifies whether the pattern increment is relation-driven or value-driven. If the increment is relation-driven, the element PRO_E_GENPAT_RELATION_EDIT contains an array of wide strings whose members are individual relations.

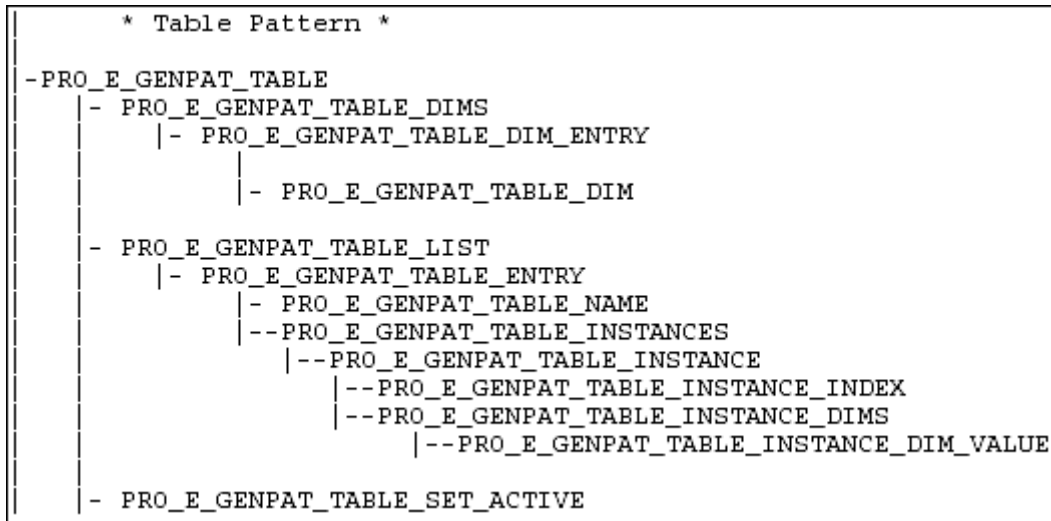
The elements PRO_E_GENPAT_DIM_FIRST_DIR_NUM_INST and PRO_E_PAT_SECOND_DIR_NUM_INST specify the number of instances in each of the pattern dimensions.

Even if the pattern extends in only one direction, you must specify elements for the second direction. In this case, add an empty PRO_E_PAT_SECOND_DIR element and set the value of the PRO_E_PAT_SECOND_DIR_NUM_INST element to 1 (not 0).

Table Patterns

If the pattern type is PRO_GENPAT_TABLE_DRIVEN, your element tree must contain the table-driven dimensions and table information (variation in dimensions for each instance). The following figure shows the elements of the Table pattern:

Figure 39-4: Element Tree for a Table Pattern



Element Trees:
Patterns

The PRO_E_GENPAT_TABLE_DIMS element is an array that contains one table dimension (PRO_E_GENPAT_TABLE_DIM) element for each dimension to be varied in the tables. The value of each PRO_E_GENPAT_TABLE_DIM element is a *ProSelection* object for the corresponding dimension.

The PRO_E_GENPAT_TABLE_LIST element is an array element that contains all the tables that control the pattern. This element should contain one PRO_E_GENPAT_TABLE_ENTRY element for each table.

Each PRO_E_GENPAT_TABLE_ENTRY element contains the name of the table (PRO_E_GENPAT_TABLE_NAME) and table instances (PRO_E_GENPAT_TABLE_INSTANCES).

Each PRO_E_GENPAT_TABLE_INSTANCE element contains an index number (PRO_E_GENPAT_TABLE_INSTANCE_INDEX) element and a dimensions (PRO_E_GENPAT_TABLE_INSTANCE_DIMS) element. The PRO_E_GENPAT_TABLE_INSTANCE_DIMS element is an array element that must contain one dimension value (PRO_E_GENPAT_TABLE_INSTANCE_DIM_VALUE) element for each of the selected dimensions in the PRO_E_PAT_MULT_TABLE_DIMS element. Note that the dimension value specifies the value of the selected dimension, not the dimension increment.

The following table lists the contents of each PRO_E_GENPAT_TABLE_ENTRY element.

Element ID Values	Element Name	Data Type
PRO_E_GENPAT_TABLE_NAME	Table name	PRO_VALUE_TYPE_WSTRING
PRO_E_GENPAT_TABLE_INSTANCES	Table instances	Array
PRO_E_GENPAT_TABLE_INSTANCE	Table instance	Compound
PRO_E_GENPAT_TABLE_INSTANCE_INDEX	Instance index	PRO_VALUE_TYPE_INT
PRO_E_GENPAT_TABLE_INSTANCE_DIMS	Dimension variations	Compound
PRO_E_GENPAT_TABLE_INSTANCE_DIM_VALUE	Dimension value	PRO_VALUE_TYPE_DOUBLE

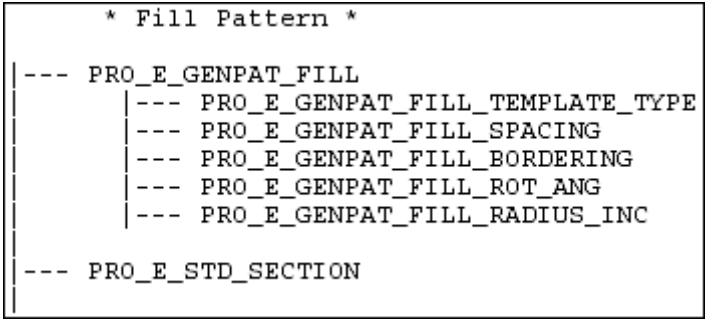
The element PRO_E_GENPAT_TABLE_SET_ACTIVE sets the active table for the pattern. Valid values are 0 (for the first table) through $(num\_tables - 1)$, where *num_tables* is the number of tables in the element tree.

Fill Patterns

A fill pattern controls the pattern by filling an area with pattern members. You can select a grid to define this area.

If the pattern type in your element tree is PRO_GENPAT_FILL_DRIVEN, the element tree must contain information about the grid and pattern members. The following figure shows the elements of the Fill pattern:

Figure 39-5: Element Tree for a Fill Pattern



Element Trees:
Patterns

The element PRO_E_GENPAT_FILL_TEMPLATE_TYPE specifies the type of grid template that you want to use to create a fill pattern.

The element PRO_E_GENPAT_FILL_SPACING specifies the spacing between the pattern members.

The element PRO_E_GENPAT_FILL_BORDERING specifies the minimum distance between the centers of the pattern members and the area boundary.

The element PRO_E_GENPAT_FILL_ROT_ANG specifies the rotation angle of the grid about the Csys origin.

The element PRO_E_GENPAT_FILL_RADIUS_INC specifies the radial spacing for circular and spiral grids.

The following table lists the contents of each PRO_E_GENPAT_FILL element.

Element ID Values	Element Name	Data Type	Valid Values
PRO_E_GENPAT_FILL_TEMPLATE_TYPE	Fill template	PRO_VALUE_TYPE_INT	PRO_GENPAT_SQUARE_FILL, PRO_GENPAT_DIAMOND_FILL, PRO_GENPAT_TRIANGLE_FILL, PRO_GENPAT_CIRCLE_FILL, PRO_GENPAT_CURVE_FILL, PRO_GENPAT_SPIRAL_FILL

Element ID Values	Element Name	Data Type	Valid Values
PRO_E_GENPAT_FILL_SPACING	Fill spacing	PRO_VALUE_TYPE_DOUBLE	
PRO_E_GENPAT_FILL_BORDERING	Fill bordering	PRO_VALUE_TYPE_DOUBLE	
PRO_E_GENPAT_FILL_ROT_ANG	Fill angle	PRO_VALUE_TYPE_DOUBLE	
PRO_E_GENPAT_FILL_RADIUS_INCREMENT	Fill radius increment	PRO_VALUE_TYPE_DOUBLE	

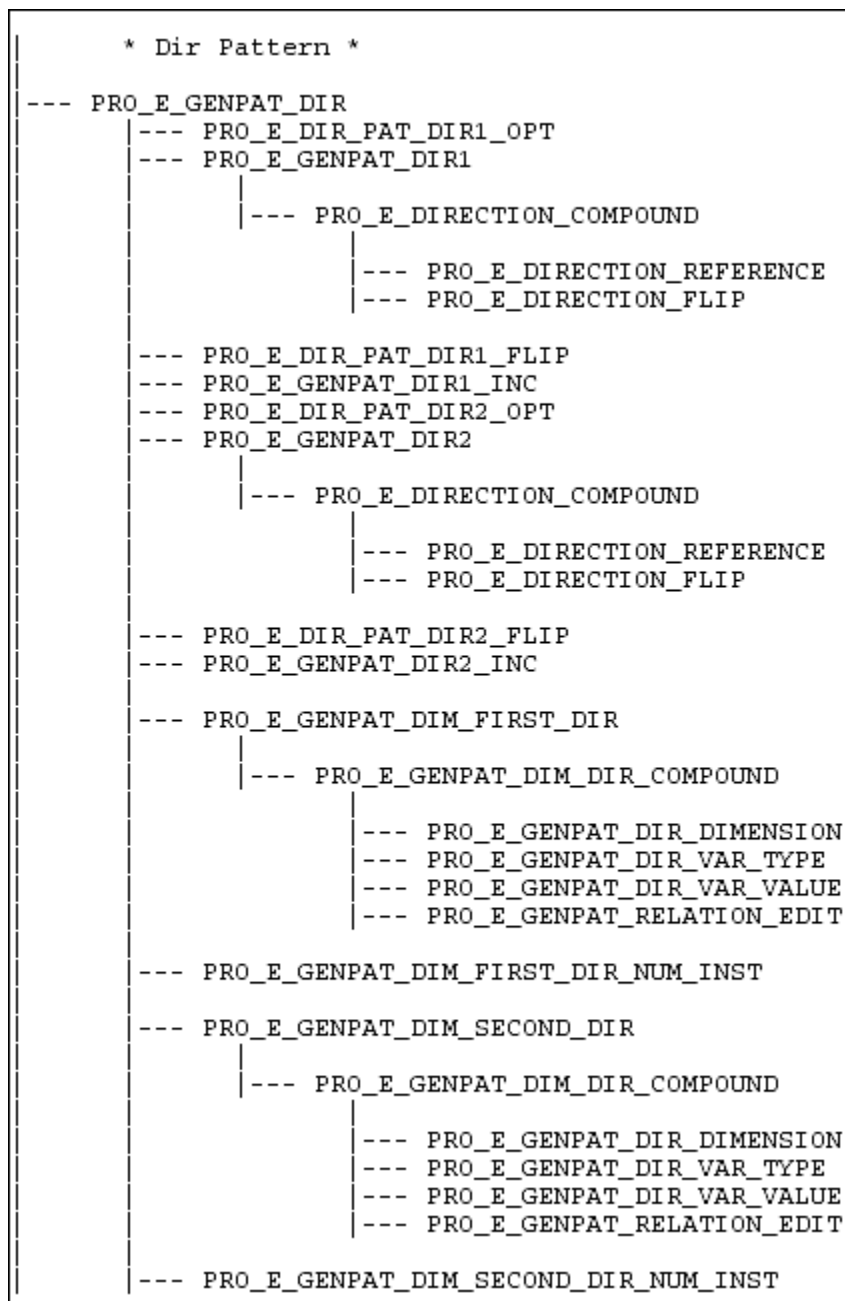
Direction Patterns

A direction pattern adds pattern members in one or two selected directions.

If the pattern type in the element tree is PRO_GENPAT_DIR_DRIVEN, the element tree must contain the information about the two directions and the pattern members.

The following figure shows the elements of a direction pattern:

Figure 39-6: Element Tree for a Direction Pattern



For a direction patter, use the elements PRO_E_DIR_PAT_DIR1_OPT and PRO_E_DIR_PAT_DIR2_OPT to specify the pattern orientation in the first and second direction, respectively. The orientation options are translation, rotation, and coordinate system.

Depending on the selected orientation, choose references for the two directions using the element PRO_E_DIRECTION_REFERENCE. To flip the selected directions, use the elements PRO_E_DIR_PAT_DIR1_FLIP and PRO_E_DIR_PAT_DIR2_FLIP. The values of the references and the choice to flip the direction are stored in the elements PRO_E_GENPAT_DIR1 and PRO_E_GENPAT_DIR2.

The elements PRO_E_GENPAT_DIR1_INC and PRO_E_GENPAT_DIR2_INC specify the spacing between the pattern members in the first and second directions, respectively.

The elements with identifiers PRO_E_GENPAT_DIM_FIRST_DIR and PRO_E_GENPAT_DIM_SECOND_DIR contain dimension information for the pattern members in the first and second direction, respectively. These elements are array elements that contain as many PRO_E_GENPAT_DIM_DIR_COMPOUND elements as required to complete the pattern. For more information on the elements PRO_E_GENPAT_DIR_DIM_COMPOUND, PRO_E_GENPAT_FIRST_DIR_NUM_INST, and PRO_E_GENPAT_SECOND_DIR_NUM_INST, refer to the section on Dimension Patterns.

The following table lists the contents of each PRO_E_GENPAT_DIR element.

Element ID Values	Element Name	Data Type	Valid Values
PRO_E_DIR_PAT_DIR1_OPT	Direction 1st option	PRO_VALUE_TYPE_INT	PRO_GENPAT_TRANSLATIONAL, PRO_GENPAT_DIR1_ROTATIONAL
PRO_E_DIR_PAT_DIR2_OPT	Direction 2nd option	PRO_VALUE_TYPE_INT	PRO_GENPAT_TRANSLATIONAL, PRO_GENPAT_DIR2_ROTATIONAL
PRO_E_GENPAT_DIR1 and PRO_E_GENPAT_DIR2	1st direction and 2nd direction	Compound	

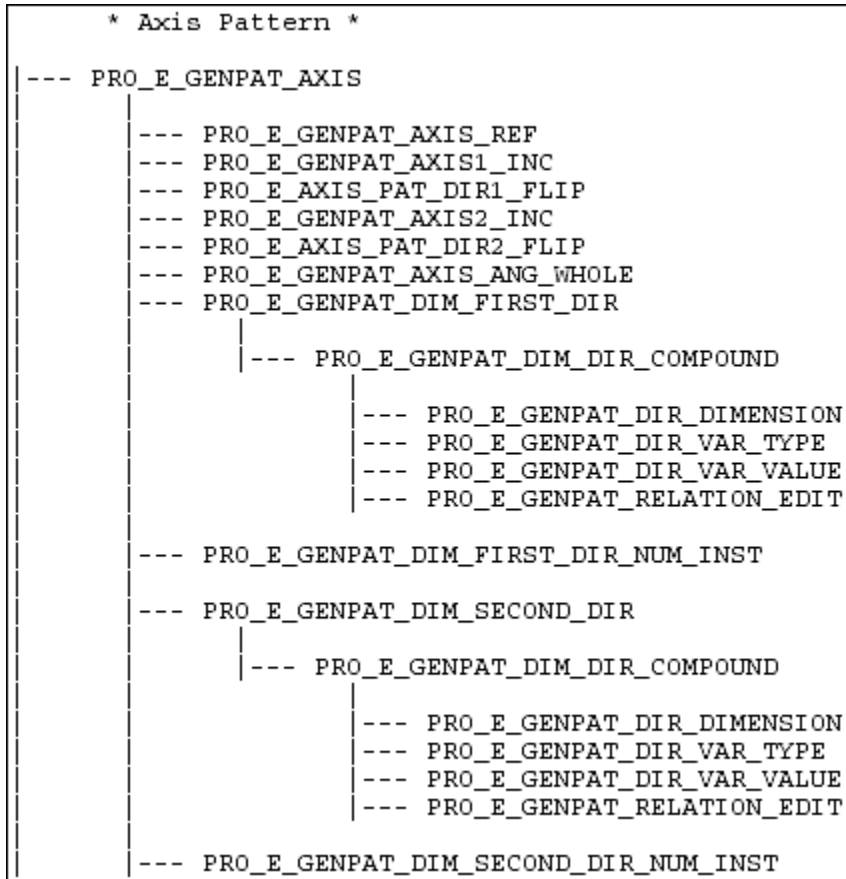
Element ID Values	Element Name	Data Type	Valid Values
PRO_E_DIRECTION_COMPOUND	PRO_E_DIRECTION_COMPOUND	Compound	
PRO_E_DIRECTION_REFERENCE	Direction reference	PRO_VALUE_TYPE_SELECTION	
PRO_E_DIRECTION_FLIP	Direction flip	PRO_VALUE_TYPE_INT	Value ignored
PRO_E_DIR_PAT_DIR1_FLIP and PRO_E_DIR_PAT_DIR2_FLIP	1st direction flip and 2nd direction flip	PRO_VALUE_TYPE_INT	0 or 1
PRO_E_GENPAT_DIR1_INC and PRO_E_GENPAT_DIR2_INC	1st direction increment and 2nd direction increment	PRO_VALUE_TYPE_DOUBLE	
PRO_E_GENPAT_DIM_FIRST_DIR and PRO_E_GENPAT_DIM_SECOND_DIR	1st direction dimensions	Array	

Axis Patterns

Use the axis pattern to create a pattern by revolving a feature around a selected axis. An axis pattern allows you to place members in the first direction or angular direction and in the second direction or radial direction.

If the pattern type in your element tree is PRO_GENPAT_AXIS_DRIVEN, the element tree must contain information about the pattern members in the two directions and the axis around which you want to create a pattern.

Figure 39-7: Element Tree for an Axis Pattern



The element PRO_E_GENPAT_AXIS_REF specifies the axis around which you want to create the pattern. The value of this element is PRO_VALUE_TYPE_SELECTION, which is of type ProSelection.

The element PRO_E_GENPAT_AXIS1_INC specifies the spacing between the pattern members in the first direction and is of type PRO_VALUE_TYPE_DOUBLE. The first direction being angular, this distance is the angular distance and the range is -360 through +360.

The element PRO_E_GENPAT_AXIS2_INC specifies the spacing between the pattern members in the second direction and is of type PRO_VALUE_TYPE_DOUBLE. The second direction being radial, this distance is the linear distance and the range is -999999999.9999 through +999999999.9999.

The elements PRO_E_AXIS_PAT_DIR1_FLIP and PRO_E_AXIS_PAT_DIR2_FLIP flip the pattern members in the first and second direction, respectively, around the axis. These elements are of type PRO_VALUE_TYPE_INT.

The element PRO_E_GENPAT_AXIS_ANG_WHOLE specifies the angular extent of the pattern members and is of type PRO_VALUE_TYPE_DOUBLE. The range for this element is 0.0000 through 999999999.9999.

The value of each PRO_E_GENPAT_DIR_DIMENSION element is a ProSelection object for the corresponding dimensions.

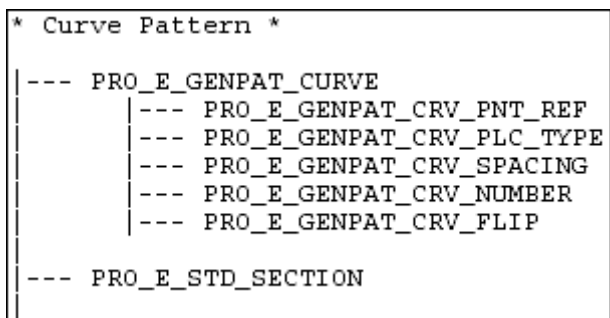
The elements with identifiers PRO_E_GENPAT_DIM_FIRST_DIR and PRO_E_GENPAT_DIM_SECOND_DIR contain dimension information for the pattern members in the first and second direction, respectively. For more information on these elements, refer to the section on Direction Patterns.

Curve Patterns

A curve pattern creates instances of a feature along a sketched curve or a datum curve.

If the pattern type in your element tree is PRO_GENPAT_CRV_DRIVEN, the element tree must contain information about the curve and the pattern members.

Figure 39-8: Element Tree for a Curve Pattern



The element PRO_E_GENPAT_CRV_PNT_REF specifies the curve to be used as a reference.

After you select the reference, you can either select a sketched curve or draw a curve using Sketcher. The element PRO_E_GENPAT_CRV_PLC_TYPE specifies the curve types.

The element PRO_E_GENPAT_CRV_SPACING specifies the separation between the pattern members.

The element PRO_E_GENPAT_CRV_NUMBER specifies the number of pattern members to be created.

The element PRO_E_GENPAT_CRV_FLIP specifies flipping the curve used in patterning.

The following table lists the contents of each PRO_E_GENPAT_CURVE element.

Element ID Values	Element Name	Data Type	Valid Values
PRO_E_GENPAT_CRV_PNT_REF	Curve reference	PRO_VALUE_TYPE_SELECTION	
PRO_E_GENPAT_CRV_PLC_TYPE	Curve type	PRO_VALUE_TYPE_INT	
PRO_E_GENPAT_CRV_SPACING	Curve spacing	PRO_VALUE_TYPE_DOUBLE	0.0000 to 1000000.0000
PRO_E_GENPAT_CRV_NUMBER	Curve number	PRO_VALUE_TYPE_INT	
PRO_E_GENPAT_CRV_FLIP	Curve flip	PRO_VALUE_TYPE_INT	0 or 1

Point Patterns

A point pattern creates a pattern by placing a pattern member at a particular point.

The following figure shows the elements of a point pattern:

Figure 39-9: Element Tree for a Point Pattern

```

* Point Pattern *
|
|--- PRO_E_GENPAT_POINT
|   |--- PRO_E_GENPAT_POINT_REF_TYPE
|   |--- PRO_E_GENPAT_POINT_REF
|
|--- PRO_E_STD_SECTION

```

If the pattern type in the element tree is PRO_GENPAT_POINT_DRIVEN, the element tree must contain the information about the reference point and the actual point at which you want to draw the pattern.

Use the element PRO_E_GENPAT_POINT_REF_TYPE to select the type of the point at which you want to repeat the selected feature. This selected point is the reference point for creating the pattern. You can select a point from the following options:

- Internal or External sketch
- Datum point feature

After the type of the reference point is set, use the element PRO_E_GENPAT_POINT_REF to select the actual point.

The following table lists the contents of each PRO_E_GENPAT_POINT element.

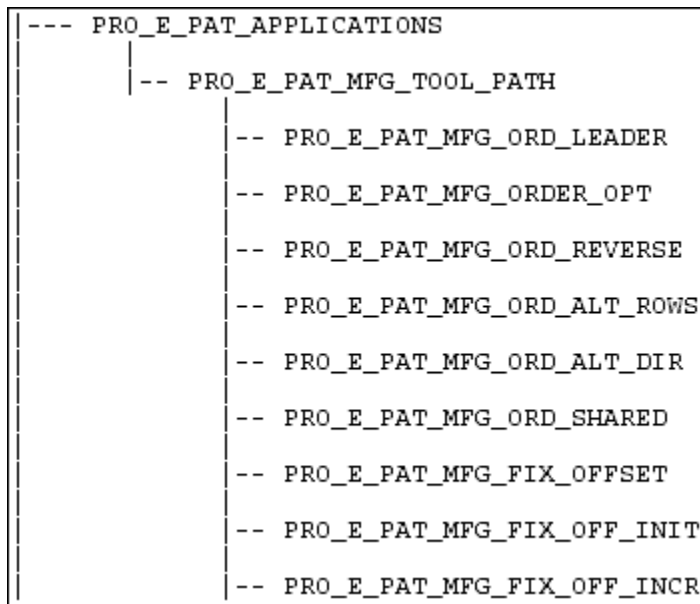
Element ID Values	Element Name	Data Type	Valid Values
PRO_E_GENPAT_POINT_REF_TYPE	Point type	PRO_VALUE_TYPE_INT	PRO_GENPAT_REF_SKETCH, PRO_GENPAT_REF_POINT
PRO_E_GENPAT_POINT_REF	Point reference	PRO_VALUE_TYPE_SELECTION	

NC Sequence Pattern

You can create an NC sequence by using the element tree for pattern applications. The element tree PRO_E_PAT_APPLICATIONS described in this section allows you to set various options related to manufacturing order and fixtures used to pattern an NC sequence.

The following figure shows the element tree for pattern applications.

Figure 39-10: Pattern Element Tree for NC Sequence



The compound element PRO_E_PAT_MFG_TOOL_PATH contains the following elements of type integer:

- PRO_E_PAT_MFG_ORD_LEADER—Specifies the number of the pattern member that you want to use as the manufacturing leader. The default value of this element is zero, which indicates that the pattern leader itself is the manufacturing leader.
- PRO_E_PAT_MFG_ORDER_OPT—Specifies the criteria for selecting the manufacturing order. The valid values are:
 - 1— Specifies the manufacturing order based on the pattern order. Set the values of the following elements as:
 - PRO_E_PAT_MFG_ORD_REVERSE—Specify 1 to reverse the order of the pattern for the CL output.
 - PRO_E_PAT_MFG_ORD_ALT_ROWS—Specify 1 to set the alternate rows of the pattern in the same direction for the CL output. Here the first and the second rows are in opposite directions with the first row being in the direction of the pattern. The manufacturing leader is the first tool path in the CL output.

- PRO_E_PAT_MFG_ORD_ALT_DIR—Specify 1 to generate the CL output with the direction of the first row being treated as the direction of the second row and the direction of second row treated as the direction of the first row until the tool paths for all the pattern members are generated. The manufacturing leader is the first tool path in the CL output.
- 2—Specifies the manufacturing order based on the shortest distance between the pattern members.
- 3—Select the manufacturing order for each of the pattern member from the pattern UI.

Note: The functionality to select the manufacturing order for each pattern member is currently not supported through Pro/TOOLKIT.

- PRO_E_PAT_MFG_ORD_SHARED— Specify 1 to sequentially set the orders for the 4-axis or 5-axis tool paths with a common Z-axis orientation.
- PRO_E_PAT_MFG_FIX_OFFSET—Specifies the parameters for the fixture offset. Specify 1 to set the following fixture options:
 - PRO_E_PAT_MFG_FIX_OFF_INIT—Specifies the initial value of the fixture Offset.
 - PRO_E_PAT_MFG_FIX_OFF_INCR—Specifies the increment value for the fixture offset.

The following table lists the contents of each PRO_E_PAT_MFG_TOOL_PATH element..

Element ID Values	Element Name	Data Type	Valid Values
PRO_E_PAT_MFG_ORD_LEADER	Number of the leader	PRO_VALUE_TYPE_INT	0 <= value < number of instances
PRO_E_PAT_MFG_ORDER_OPT	Order options	PRO_VALUE_TYPE_INT	1, 2, 3 (if you specify 3, order may be selected only through Pro/ENGINEER UI)
PRO_E_PAT_MFG_ORD_REVERSE	Reverse option	PRO_VALUE_TYPE_INT	0 or 1

Element ID Values	Element Name	Data Type	Valid Values
PRO_E_PAT_MF G_ORD_ALT_ROWS	Alternate rows	PRO_VALUE_TYPE _INT	0 or 1
PRO_E_PAT_MF G_ORD_ALT_DIR	Alternate direction	PRO_VALUE_TYPE _INT	0 or 1
PRO_E_PAT_MF G_ORD_SHARED	Shared orientation	PRO_VALUE_TYPE _INT	0 or 1
PRO_E_PAT_MF G_FIX_OFFSET	Fixture offsets	PRO_VALUE_TYPE _INT	0 or 1
PRO_E_PAT_MF G_FIX_OFF_INIT	Initial fixture offsets	PRO_VALUE_TYPE _INT	1 <= value
PRO_E_PAT_MF G_FIX_OFF_INCREMENT	Fixture offsets increment	PRO_VALUE_TYPE _INT	1 <= value

For more information on the patterning options related to an NC sequence, refer to Manufacturing section of the Pro/ENGINEER help.

Obtaining the Element Tree for a Pattern

Function Introduced:

- **ProPatternElemtreeCreate()**

To obtain the element tree for a pattern, call the function **ProPatternElemtreeCreate()**. You can then use the element tree read-access functions described in the sections Feature Elements (on page 29 - 13) and Feature Element Paths (on page 29 - 11), such as the functions **ProElement*Get()**, **ProElement*Visit()**, and **ProElementArrayGet()**.

Note: Inspection of Fill, Axis, and Directional patterns is not supported via the element tree in Pro/ENGINEER Wildfire 2.0. **ProPatternElemtreeCreate()** returns PRO_TK_NOT_IMPLEMENTED for fill patterns.

Visiting and Creating a Pattern

Functions Introduced:

- **ProPatternMemberVisit()**
- **ProPatternMembersGet()**
- **ProPatternCreate()**

The function **ProPatternMemberVisit()** visits the feature members in a pattern. This function takes the visit action function **ProFeatureVisitAction()** and the filter action function **ProFeatureFilterAction()** as its input arguments. The function **ProFeatureFilterAction()** is a generic action function for filtering features from a pattern. It returns the filter status of the features in the pattern. This status is used as an input argument by the function **ProFeatureVisitAction()**.

The function **ProPatternMembersGet()** returns the feature members in a pattern. For a group pattern, the output argument is a group pattern feature.

When your element tree is complete, create the pattern by calling the function **ProPatternCreate()**. This function requires as input the feature (*ProFeature*) to be patterned and the pattern class (feature or group) of the new pattern.

To obtain the *ProPattern* handle for the new pattern, call the function **ProFeaturePatternGet()** with the same input feature as **ProPatternCreate()**. For more information on the function **ProFeaturePatternGet()**, refer to the section Manipulating Patterns in the Core: Features chapter.

Element Trees: Sections

A section is a parametric two-dimensional cross section used to define the shape of three-dimensional features, such as extrusions. In Pro/ENGINEER, you create a section interactively using Sketcher mode. In a Pro/TOOLKIT application, you can create sections completely programmatically using the functions described in this section.

Topic	Page
Overview	40 - 2
Creating Section Models	40 - 2

Overview

A section is a parametric two-dimensional model used to define the shape of three-dimensional features in parts and assemblies. When using Pro/ENGINEER interactively, you create a section using Sketcher mode. In a Pro/TOOLKIT application, you can create sections completely programmatically using the functions described in this section.

Sections fall into two types: 2D and 3D. Both types are represented by the object *ProSection* (an opaque handle) and manipulated by the same functions.

The difference between the types arises out of the context in which the section is being used, and affects the requirements for the contents of the section and also of the feature element tree in which it is placed when creating a sketched feature.

Put simply, a 2D section is self-contained, whereas a 3D section contains references to 3D geometry in a parent part or assembly.

You can use Intent Datums such Intent Axis, Intent Point, Intent Plane, and Intent Coordinate System as references for sketcher dimensions. You can use Intent Point and Intent Axis to create sections using projections.

In a Pro/TOOLKIT application, you can work with a section either in an Intent Manager or a non-Intent Manager mode. In the non-Intent Manager mode, if you make any changes to the section, you must solve and regenerate the section to apply the changes. On the other hand, in the Intent Manager mode, all the changes are applied immediately.

You can create section constraints programmatically using the Intent Manager property. This corresponds to creating sections within the Intent Manager mode in Pro/ENGINEER.

This chapter is concerned with 2D sections, which are the simplest. The extra steps required to construct a 3D section are described in the chapter 'Element Trees: Sketched Features', which follows this one.

Creating Section Models

A 2D section, because it is self-contained, may be stored as a Pro/ENGINEER model file. It then has the extension *.sec*.

The steps required to create and save a section model using Pro/TOOLKIT follow closely those used in creating a section interactively using Sketcher mode in Pro/ENGINEER.

To Create and Save a Section Model

1. Allocate the two-dimensional section and define its name.
2. Add section entities (lines, arcs, splines, and so on) to define the section geometry, in section coordinates.
3. Add section dimensions that parametrically drive the shape of the entities.
4. Solve and regenerate the section.
5. Save the section.

When you are creating a section that is to be used in a sketched feature, Steps 1 and 5 will be replaced by different techniques. these techniques are described fully in the chapter on 'Element Trees: Sketched Features'.

The steps are described in more detail in the following sections.

Allocating a Two-Dimensional Section

Functions Introduced:

- **ProSection2DAlloc()**
- **ProSectionFree()**
- **ProSectionNameSet()**
- **ProSectionNameGet()**

A two-dimensional section is identified in Pro/TOOLKIT by an opaque pointer called *ProSection*. This type, and the functions in this section, are declared in the include file *ProSection.h*.

The function **ProSection2DAlloc()** allocates memory for a new, standalone section and outputs a *ProSection* handle to identify it. All the other Pro/TOOLKIT functions that operate on sections take this *ProSection* as their first input argument.

The function **ProSectionNameSet()** enables you to set the name of a section. Calling this function places the section in the Pro/ENGINEER namelist and enables it to be recognized by Pro/ENGINEER as a section model in the database.

The following code fragment shows how to use these two functions.

```
ProSection    section;  
ProName      wname;  
  
ProSection2DAlloc (&section);  
ProStringToWstring (wname, "demo");  
ProSectionNameSet (section, wname);
```

Such sections created programmatically are in the non-Intent Manager mode by default.

To free a section allocated with **ProSection2DAlloc()**, you must use **ProSectionFree()**.

Setting the Mode of a Section

Functions Introduced:

- **ProSectionIntentManagerModeGet()**
- **ProSectionIntentManagerModeSet()**

Use the function **ProSectionIntentManagerModeGet()** to check if the Intent Manager property is ON or OFF for the specified section.

Use the function **ProSectionIntentManagerModeSet()** to set the Intent Manager property to ON or OFF for the specified section. This function must be called before using the other Pro/TOOLKIT functions to access sections with the Intent Manager property set to ON.

Copying the Current Section

Functions Introduced:

- **ProSectionActiveGet()**
- **ProSectionActiveSet()**

Use the function **ProSectionActiveGet()** to create a copy of the section that you are using currently; this copy is created within the same Sketcher session. The mode of such a section depends on the current Sketcher mode. Starting from Pro/ENGINEER Wildfire 5.0, the Intent Manager mode is the default sketcher mode. Use the function **ProSectionFree()** to free the memory allocated to the section obtained with the function **ProSectionActiveGet()**.

Use the function **ProSectionActiveSet()** to set the specified section as the current active Sketcher section.

Note: The function call **ProSectionActiveSet()** makes the **Undo** and **Redo** menu options available in Pro/ENGINEER.

Section Constraints

Functions Introduced:

- **ProSectionConstraintsIdsGet()**
- **ProSectionConstraintsGet()**
- **ProSectionConstraintDeny()**
- **ProSectionConstraintCreate()**
- **ProSectionConstraintDelete()**

The function **ProSectionConstraintsIdsGet()** returns an array of section constraint identifiers that currently exist in the specified section.

Note: You must solve the section first by calling the function **ProSectionSolve()** to get the section constraints. Because adding or deleting section entities might invalidate the current list of section constraint identifiers, you must solve the section again to get the up-to-date list.

If a section has not been fully dimensioned with dimensions created explicitly by the user, the Sketcher will make assumptions in order to solve the section. If the Sketcher can assume enough constraints to find a unique solution to the section, it solves the section successfully.

However, you might want to disable certain Sketcher constraints to have more control over the way the section is dimensioned and solved. To do this, use the function **ProSectionConstraintDeny()** to deny a certain section constraint.

Note: The function **ProSectionConstraintDeny()** is not supported for sections that have the Intent Manager property set to ON.

The function **ProSectionConstraintsGet()** returns information about the specified section constraint. It takes as input the section handle and the constraint identifier for which the information is requested. The function returns details about the section constraint including its type, status, and references. The constraint types are defined in the include file *ProSecConstr.h*. The following table lists the possible constraint types.

Constraint Type	Description
PRO_CONSTRAINT_SAME_POINT	Make the points coincident.
PRO_CONSTRAINT_HORIZONTAL_ENT	Make the entity horizontal.
PRO_CONSTRAINT_VERTICAL_ENT	Make the entity vertical.
PRO_CONSTRAINT_PNT_ON_ENT	Place the point on the entity.
PRO_CONSTRAINT_TANGENT_ENTS	Make the entities tangent.
PRO_CONSTRAINT_ORTHOG_ENTS	Make the entities perpendicular.
PRO_CONSTRAINT_EQUAL_RADII	Make the arcs or circles of equal radius.
PRO_CONSTRAINT_PARALLEL_ENTS	Make the entities parallel.
PRO_CONSTRAINT_EQUAL_SEGMENTS	Make the segments of equal length.
PRO_CONSTRAINT_COLLINEAR_LINES	Make lines co-linear.
PRO_CONSTRAINT_SYMMETRY	Impose symmetry.
PRO_CONSTRAINT_SAME_X/Y_COORD	Assume the endpoints and centers of arcs to have the same coordinates.

The possible types of constraint status are as follows:

- **PRO_TK_CONSTRAINT_DENIED**—The constraint is denied. This gives you more control over the section.
- **PRO_TK_CONSTRAINT_ENABLED**—The constraint is enabled. The Sketcher uses the predefined assumption.

Use the function **ProSectionConstraintCreate()** to create constraints between entities in the specified section. Use the function **ProSectionConstraintDelete()** to delete the specified section constraint.

Note: The function **ProSectionConstrainCreate()** works only if the Intent Manager property of the specified section is set to ON.

Solving and Regenerating a Section

Functions Introduced:

- **ProSectionEpsilonGet()**
- **ProSectionEpsilonSet()**
- **ProSectionSolve()**
- **ProSectionSolveRigid()**
- **ProSecdimValueGet()**
- **ProSecdimValueSet()**
- **ProSectionRegenerate()**

Although the action of the **Regenerate** command in Sketcher mode is seen as a single operation by the Pro/ENGINEER user, it is in fact composed of two distinct actions. These two operations are invoked separately from a Pro/TOOLKIT application. The two operations are as follows:

- *Solving*—Calculating the way in which the geometry of the entities is driven by the dimensions. It is at this stage that Sketcher constraints are applied, under- or over-dimensioning is discovered and reported, and values are assigned to new dimensions.
- *Regenerating*—Reconstructing the geometry of the section to obey the current dimension values.

You invoke these stages using the functions **ProSectionSolve()** and **ProSectionRegenerate()**, respectively.

Note: The **ProSectionSolve()** and **ProSectionRegenerate()** are not supported for sections that have the Intent Manager property set to ON.

You *must* solve a programmatically-created section before using it to build three-dimensional geometry. You need to regenerate the section only if you have explicitly modified the dimension values since you solved the section.

When you create a section interactively using Sketcher mode, you normally adjust the values of dimensions after the first regeneration, because the initial values assigned to them correspond to the free-hand, initial sketch and are therefore not exact. When you create a section with Pro/TOOLKIT, the entities are usually created with exactly the geometry needed in the finished section. Therefore, although solving is always necessary, it is not usually necessary to explicitly reset dimension values or regenerate the section.

Solving a section in Pro/TOOLKIT involves applying the same constraints used in interactive Sketcher mode. Pro/TOOLKIT, like the Sketcher, identifies situations of near symmetry in the section, assumes them to be intended as exact symmetry, and constrains them to be symmetrical in future regenerations. For example, lines that are nearly the same length are assumed to be intended to be the same length, and are therefore constrained to be so.

The function **ProSectionSolveRigid()** solves the specified section by fixing the coordinates of all the section entities with respect to a coordinate system. In this way, the section entities do not have to be solved individually. To use this function, a coordinate system within the section must exist; the function uses the first coordinate system found in the section.

Note: You must ensure that the added section entities are correct because potential errors will *not* be solved and may show up only during later stages.

When there are a lot of section entities, this function dramatically reduces the amount of time required to solve a section.

Note: The function **ProSectionSolveRigid()** is not supported for sections that have the Intent Manager property set to ON.

If you are building a section in which some geometry is intended as nearly symmetric, it is important to be able control the resolution with which the solving constraints are applied. To do this, use the function **ProSectionEpsilonSet()**. For example, if your section has two lines that differ in length by 0.5, set the epsilon to a value less than 0.5 to ensure that **ProSectionSolve()** does not constrain the lines to be the same length. To get the current epsilon value for the section, use the function **ProSectionEpsilonGet()**.

Note: You cannot set the epsilon to zero.

The functions **ProSecdimValueGet()** and **ProSecdimValueSet()** enable you to access the value of a dimension. If you change dimension values, you must call **ProSectionRegenerate()** to recalculate the new section shape.

Automatic Section Dimensioning

Function Introduced:

- **ProSectionAutodim()**

The function **ProSectionAutodim()** is used to automatically add needed dimensions to a section to make it fully constrained. It takes as input a *ProSection* handle and a pointer to the opaque structure called *ProWSecerror*. Before calling this function, be sure to allocate the pointer to *ProWSecerror* using **ProSecerrorAlloc()**. Any errors resulting from the call to the function **ProSectionAutodim()** are stored in the *ProWSecerror* structure. To free the allocated memory, call the function **ProSecerrorFree()**.

The **ProSectionAutodim()** function can be used on a section where no dimensions have been created yet, as well as on a partially dimensioned section.

If dimensions have been added successfully, the function **ProSectionAutodim()** also solves the input section.

Note: The function **ProSectionAutodim()** is not supported for sections that have the Intent Manager property set to ON.

Adding Section Entities

Functions Introduced:

- **ProSectionEntityAdd()**
- **ProSectionEntityDelete()**
- **ProSectionEntityReplace()**

The function **ProSectionEntityAdd()** takes as input the *ProSection* that identifies the section, and a pointer to a user-visible structure called *Pro2dEntdef*, which defines the entity.

The *Pro2dEntdef* structure is a generic structure that contains only a field indicating the type of entity. For each type of entity, there is a dedicated structure that has the entity type as its first field; these structures are named *Pro2dLinedef*, *Pro2dArcdef*, and so on. The Pro/TOOLKIT application builds up the structure appropriate to the entity to be added, and inputs it to **ProSectionEntityAdd()** by casting its address to (*Pro2dEntdef**). The entity structures are declared in the include file *Pro2dEntdef.h*.

The function **ProSectionEntityAdd()** outputs an integer that is the identifier of the new entity within the section. The Pro/TOOLKIT application needs these values because they are used to refer to entities when adding dimensions.

The following code fragment demonstrates how to add a single line entity.

```
Pro2dLinedef    line;
int             line_id;

line.type       = PRO_2D_LINE;
line.end1[0]    = 0.0;
line.end1[1]    = 0.0;
line.end2[0]    = 10.0;
line.end2[1]    = 0.0;

ProSectionEntityAdd (section,
                    (Pro2dEntdef*)&line, &line_id);
```

The function **ProSectionEntityDelete()** enables you to delete a section entity from the specified section.

The function **ProSectionEntityReplace()** enables you to replace an existing entity from the specified section with another entity in the same section. This functionality enables you to redefine an existing section programmatically.

To use the function **ProSectionEntityReplace()**, you must first add the new entity to the section (to get its identifier), then replace the old entity identifier with the new one.

Accessing Selection Reference of the Entity

Functions Introduced:

- **ProSectionEntityGetSelected()**

The function **ProSectionEntityGetSelected()** provides the references of the selected entity. The input arguments of this function are:

- *handle*—The section handle.
- *entity_id*—The identifier of the section entity.
- *pnt_type*—Specifies the type of point selection on the entity. The valid values are:
 - PRO_ENT_WHOLE—Specifies the whole entity.
 - PRO_ENT_START—Specifies the start point of the entity.
 - PRO_ENT_END—Specifies the end point of the entity.
 - PRO_ENT_CENTER—Specifies the center of the entity.
 - PRO_ENT_LEFT_TANGENT—Specifies the point on the entity, where the normalized param value is 0.5.
 - PRO_ENT_RIGHT_TANGENT—Specifies the point on the entity, where the normalized param value is 0.0.
 - PRO_ENT_TOP_TANGENT—Specifies the point on the entity, where the normalized param value is 0.25.
 - PRO_ENT_BOTTOM_TANGENT—Specifies the point on the entity, where the normalized param value is 0.75.
- *pnt*—Specifies the location of the point on the entity geometry.
- *idx_pnt*—Specifies the index of interpolation point relative to *pnt_type*, if the section entity is a spline. In this case the value of the argument *pnt* is ignored. Specify PRO_VALUE_UNUSED if the entity is not a spline.

This function creates the selection object programmatically for use in functions that require selection references of entities as the input.

Construction Entities

Functions Introduced:

- **ProSectionEntityIsConstruction()**
- **ProSectionEntityConstructionSet()**

Use the function **ProSectionEntityIsConstruction()** to determine if the specified entity is a construction entity. Construction entities are used for reference and are not used to create feature geometry.

The function **ProSectionEntityConstructionSet()** sets the specified entity to be of type construction.

Modifying Entities

Functions Introduced:

- **ProSectionEntityIntersectionGet()**
- **ProSectionEntityParamEval()**
- **ProSectionEntityCorner()**
- **ProSectionEntityDivide()**

The function **ProSectionEntityIntersectionGet()** returns the intersection points between the two section entities. Use the function **ProArrayFree()** to free the memory.

The function **ProSectionEntityParamEval()** to find the corresponding normalized parameter on the curve, given the XY point.

The function **ProSectionEntityCorner()** trims the selected entities to each other. The selected entities may not intersect with each other. The entities may be trimmed by either cutting them or extending them.

Use the function **ProSectionEntityDivide()** to divide a section entity into two or more new entities. If the entity is dimensioned then delete the dimensions before dividing it.

Adding Section Dimensions

Functions Introduced:

- **ProSecdimCreate()**
- **ProSecdimDelete()**
- **ProSecdimDiameterSet()**
- **ProSecdimDiameterClear()**
- **ProSecdimDiameterInquire()**
- **ProSecdimStrengthen()**

- **ProSecdimLockSet()**
- **ProSecdimIsLocked()**

When you create a dimension interactively in Sketcher mode, you select entities and points on entities and Pro/ENGINEER deduces from those picks what type of dimension is being added. When you add a dimension using the function **ProSecdimCreate()**, you must specify the dimension type. The dimension types are defined in the include file *ProSecdimTypes.h*. The following table lists the possible values.

Constant	Description
PRO_TK_DIM_LINE	Length of a line
PRO_TK_DIM_LINE_POINT	Distance between a line and a vertex
PRO_TK_DIM_RAD	Radius of an arc or a circle
PRO_TK_DIM_DIA	Diameter of an arc or a circle
PRO_TK_DIM_LINE_LINE	Distance between two lines
PRO_TK_DIM_PNT_PNT	Distance between two points
PRO_TK_DIM_PNT_PNT_HORIZ	Distance between two points (X coordinates)
PRO_TK_DIM_PNT_PNT_VERT	Distance between two points (Y coordinates)
PRO_TK_DIM_AOC_AOC_TAN_HORIZ	Horizontal distance between two arcs or circles
PRO_TK_DIM_AOC_AOC_TAN_VERT	Vertical distance between two arcs or circles
PRO_TK_DIM_ARC_ANGLE	Angle of an arc
PRO_TK_DIM_LINES_ANGLE	Angle between two lines
PRO_TK_DIM_LINE_AOC	Distance between a line and an arc or a circle
PRO_TK_DIM_LINE_CURVE_ANGLE	Angle between a spline and a line
PRO_TK_DIM_3_PNT_ANGLE	Angular dimension defined by three points
PRO_TK_DIM_DIA_LINEAR	Linear diameter dimension
PRO_TK_DIM_PNT_PNT_ORI	Distance between two points in specified orientation
PRO_TK_DIM_AOC_AOC_ORI	Distance between two arcs or circles in specified orientation

Constant	Description
PRO_TK_DIM_TOT_INC_ANG	Total included angle

The function **ProSecdimCreate()** takes several input arguments, including the following:

- *int entity_ids[]*—An array of integers that are the identifiers of the section entities to which the dimension refers.
- *ProSectionPointType point_types[]*—A dimension can reference a vertex (the end of an entity), the center of an arc or a circle, a line or circle itself (the whole entity), or tangent points on an arc or a circle. To specify these types of dimension reference points, specify the appropriate point type constant for each dimension in the *entity_ids* array. These constants are listed in the include file *ProSecdimType.h*.
- *int num_ids*—The number of section dimension identifiers in the *entity_ids* array. This is typically 1 or 2 (line length versus a point-to-point dimension).
- *ProSecdimType dim_type*—The type of section dimension to create, as listed in the *ProSecdimType.h* file.
- *Pro2dPnt place_pnt*—The two-dimensional location of the dimension label. This is equivalent to the middle mouse button pick when you are using Sketcher mode.

Note that the position of this label can sometimes determine the exact role of the dimension. For example, a dimension of type PRO_TK_DIM_LINES_ANGLE may refer to the acute or obtuse angle between two lines, depending on where the label is positioned.

The **ProSecdimCreate()** function outputs the identifier of the dimension, which is needed to identify the dimension if its value needs to be changed at a later time.

Note: The dimensions do *not* need to be given values to create a complete and correct section of any form. See the section Solving and Regenerating a Section for a detailed explanation of the assignment of values.

The following code fragment shows how to create a dimension for the length of a line entity.

```
int                line_id[1], width_dim;
Pro2dPnt          point;
ProSectionPointType pnt_type[1];
```



```

line_id[0] = 1;
point[0] = 5.0;
point[1] = 1.0;
pnt_type[0] = PRO_ENT_WHOLE;

ProSecdimCreate (section, line_id, pnt_type, 1,
                 PRO_TK_DIM_LINE, point,
                 &width_dim);

```

The following code fragment shows how to create a dimension for the horizontal distance between two arc ends.

```

int          arc1_id, arc2_id, arc1_end2, arc2_end1,
              dist_dim;
Pro2dPnt     point;
int          entities[2];
ProSectionPointType pnt_types[2];

pnt_types[0] = PRO_ENT_START;
pnt_types[1] = PRO_ENT_END;
entity[0]    = arc1_end2;
entity[1]    = arc2_end1;
point[0]     = 5.0;
point[1]     = 5.0;

ProSecdimCreate (section, entities, pnt_types, 2,
                 PRO_TK_DIM_PNT_PNT_HORIZ, point, &dist_dim);

```

The **ProSecdimDiam...**() functions extend the dimension creation functionality to include diameters for sections used to create revolved features. Function **ProSecdimDiamSet()** converts a specified section dimension (between a centerline and another entity) into a diameter dimension. **ProSecdimDiamClear()** does the opposite, converting a diameter dimension into a regular one. Use function **ProSecdimDiamInquire()** to determine if a dimension is a diameter dimension.

The function **ProSecdimStrengthen()** converts a weak dimension to a strong dimension.

You can lock or unlock sketch dimensions. Locking of dimensions avoids modifications to the sections outside the sketcher mode. The function **ProSecdimIsLocked()** determines whether a sketch dimension is locked. Use the function **ProSecdimLockSet()** to lock or unlock a specified dimension.

Example 1: Creating Spline Point Dimensions in Sections

The following sample code creates a Section dimension between the first and last interpolation point of a Spline.

```
/*=====*\
FILE      : Ug3DSectSplineDim.c
PURPOSE   : Creates a Section dimension between the first and last
            interpolation point of a Spline.

\*=====*/

/*----- Pro/Toolkit Includes -----*/
#include "ProToolkit.h"
#include "ProSection.h"
#include "ProSecdim.h"
#include "ProSecerror.h"
#include "Pro2dEntdef.h"
#include "ProSecdimType.h"
#include "ProMdl.h"
#include "UtilMath.h"
#include "ProMenu.h"
#include "ProUtil.h"
#include "ProGraphic.h"

/*----- Application Includes -----*/
#include "TestError.h"

#define MSGFILE L##"msg_ug3dsketch.txt"

ProError UserAddSplineDimension();
ProError UserGetFeatureSection(ProFeature* feat, ProSection *section);
ProError UserCreatePoint (double x, double y, ProSection* section);
ProError UserSetFeatureSection(ProFeature* feat, ProSection *section);

/*=====*\
FUNCTION   : UserAddSplineDimension()
PURPOSE    : Adds a sectiond dimension to spline entity
\*=====*/

ProError UserAddSplineDimension()
{
    wchar_t w_pane_name[PRO_LINE_SIZE];
    ProCharName pane_name;
```

```

ProMdl model;
ProSelection *sel;
int num_sel;
ProFeature sk_feat;
ProSection feat_section;
int *int_ids, no_of_ids, i;
int no_of_points, irange[2];
int usr_pnt1, usr_pnt2;
Pro2dEntdef *entity_def;
Pro2dSplinedef *spline_def;
Pro2dPointdef *point_def;
ProBoolean spline_found = PRO_B_FALSE;
Pro2dPnt pnt1, pnt2, pnt3;
ProSelection spline_sel, spline_sel2, point_sels[2],
                point1_sel, point2_sel;
int ent_ids[2], count = 0;
ProSelection* sel_array = NULL, *sel_array2 = NULL;
int constr_id;
ProWSecerror sec_error;
ProSectionPointType point_types[2];
Pro2dPnt place_point;
int sec_dim_id;

status = ProMdlCurrentGet(&model);
ERROR_CHECK("UserAddSplineDimension",
            "ProMdlCurrentGet",status);

if (status != PRO_TK_NO_ERROR)
{
    ProMessageDisplay (MSGFILE, "USER Model does not
                        exist or bad context");

    return status;
}

status = ProSelect("feature", 1, NULL, NULL, NULL, NULL, &sel,
                  &num_sel);
ERROR_CHECK("UserAddSplineDimension", "ProSelect",status);

status = ProSelectionModelitemGet(sel[0], (ProModelitem*)&sk_feat);
ERROR_CHECK("UserAddSplineDimension",
            "ProSelectionModelitemGet",status);

/*=====*\
Access the feature's section pointer from its element tree.
\*=====*/

status = UserGetFeatureSection(&sk_feat, &feat_section);
ERROR_CHECK("UserAddSplineDimension",
            "UserGetFeatureSection",status);

```

```

if (status != PRO_TK_NO_ERROR)
    return status;

status = ProSectionEntityIdsGet(feats_section,
                                &int_ids, &no_of_ids);
ERROR_CHECK("UserAddSplineDimension",
            "ProSectionEntityIdsGet",status);

/*=====*\
Get the number of interpolation points of the spline entity.
\*=====*/
for (i=0 ; i<no_of_ids ; i++)
{
    status = ProSectionEntityGet (feats_section, int_ids[i],
                                &entity_def);
    ERROR_CHECK("UserAddSplineDimension",
                "ProSectionEntityGet",status);

    if(entity_def->type == PRO_2D_SPLINE)
    {
        spline_found = PRO_B_TRUE;

        spline_def = (Pro2dSplinedef *)entity_def;
        no_of_points = spline_def->n_points;

    }
}

if (!spline_found)
{
    status = ProMessageDisplay (MSGFILE,
                                "USER Feature section does not contain spline");
    ERROR_CHECK("UserAddSplineDimension",
                "ProMessageDisplay",status);

    return PRO_TK_E_NOT_FOUND;
}

status = ProMessageDisplay (MSGFILE,
                            "USER Spline has %0d interpolation points", &no_of_points);
ERROR_CHECK("UserAddSplineDimension", "ProMessageDisplay",status);

/*=====*\
Create a point entity on the first and last interpolation point
of the spline entity.
\*=====*/

status = UserCreatePoint (
    spline_def->point_arr[0][0],
    spline_def->point_arr[0][1],

```

```

        &feat_section); /* Spline start point */
ERROR_CHECK("UserAddSplineDimension", "ProMdlCurrentGet", status);

status = UserCreatePoint (
    spline_def->point_arr[no_of_points-1][0],
    spline_def->point_arr[no_of_points-1][1],
    &feat_section); /* Spline end point */
ERROR_CHECK("UserAddSplineDimension", "UserCreatePoint", status);

status = ProSectionEntityIdsGet(feat_section, &int_ids, &no_of_ids);
ERROR_CHECK("UserAddSplineDimension",
    "ProSectionEntityIdsGet", status);

for (i=0 ; i<no_of_ids ; i++)
{
    status = ProSectionEntityGet (feat_section, int_ids[i],
                                &entity_def);
    ERROR_CHECK("UserAddSplineDimension",
        "ProSectionEntityGet", status);

    if(entity_def->type == PRO_2D_SPLINE)
    {
        pnt1[0] = spline_def->point_arr[0][0];
        pnt1[1] = spline_def->point_arr[0][1];

        pnt2[0] = spline_def->point_arr[no_of_points-1][0];
        pnt2[1] = spline_def->point_arr[no_of_points-1][1];

        status = ProSectionEntityGetSelected(feat_section, int_ids[i],
            PRO_ENT_START, pnt1, PRO_VALUE_UNUSED, &spline_sel);
        ERROR_CHECK("UserAddSplineDimension",
            "ProSectionEntityGetSelected", status);

        status = ProSectionEntityGetSelected(feat_section, int_ids[i],
            PRO_ENT_END, pnt2, PRO_VALUE_UNUSED,
            &spline_sel2);
        ERROR_CHECK("UserAddSplineDimension",
            "ProSectionEntityGetSelected", status);

    }

    if (entity_def->type == PRO_2D_POINT)
    {
        point_def = (Pro2dPointdef*)entity_def;

        pnt3[0] = point_def->pnt[0];
        pnt3[1] = point_def->pnt[1];

        ent_ids[count] = int_ids[i];

        status = ProSectionEntityGetSelected(feat_section, int_ids[i],

```

```

        PRO_ENT_WHOLE, pnt3, PRO_VALUE_UNUSED,
        &(point_sels[count++]));
    ERROR_CHECK("UserAddSplineDimension",
        "ProSectionEntityGetSelected",status);

}

}

/*=====*\
Create "Same point" constraint between the point and spline
entity at the two locations.
\*=====*/

status = ProArrayAlloc(0, sizeof(ProSelection), 1,
    (ProArray*)&sel_array);
ERROR_CHECK("UserAddSplineDimension", "ProArrayAlloc",status);

status = ProArrayObjectAdd((ProArray*)&sel_array,
    PRO_VALUE_UNUSED, 1,
    &spline_sel);
ERROR_CHECK("UserAddSplineDimension", "ProArrayObjectAdd",status);
status = ProArrayObjectAdd((ProArray*)&sel_array,
    PRO_VALUE_UNUSED, 1,
    &(point_sels[1]));
ERROR_CHECK("UserAddSplineDimension", "ProArrayObjectAdd",status);
status = ProSectionConstraintCreate(feat_section, sel_array, 2,
    PRO_CONSTRAINT_SAME_POINT, &constr_id);
ERROR_CHECK("UserAddSplineDimension", "ProSectionConstraintCreate",
    status);
status = ProArrayFree((ProArray*)&sel_array);
ERROR_CHECK("UserAddSplineDimension", "ProArrayFree",status);

status = ProArrayAlloc(0, sizeof(ProSelection), 1,
    (ProArray*)&sel_array2);
ERROR_CHECK("UserAddSplineDimension", "ProArrayAlloc",status);

status = ProArrayObjectAdd((ProArray*)&sel_array2,
    PRO_VALUE_UNUSED, 1, &spline_sel2);
ERROR_CHECK("UserAddSplineDimension", "ProArrayObjectAdd",status);

status = ProArrayObjectAdd((ProArray*)&sel_array2,
    PRO_VALUE_UNUSED, 1, &(point_sels[0]));
ERROR_CHECK("UserAddSplineDimension", "ProArrayObjectAdd",status);

status = ProSectionConstraintCreate(feat_section, sel_array2, 2,
    PRO_CONSTRAINT_SAME_POINT, &constr_id);
ERROR_CHECK("UserAddSplineDimension", "ProSectionConstraintCreate",
    status);

status = ProArrayFree((ProArray*)&sel_array2);

```

```

ERROR_CHECK("UserAddSplineDimension", "ProArrayFree", status);

status = ProSecerrorAlloc(&sec_error);
ERROR_CHECK("UserAddSplineDimension", "ProSecerrorAlloc", status);

status = ProSectionSolve(feats_section, &sec_error);
ERROR_CHECK("UserAddSplineDimension", "ProSectionSolve", status);

status = ProSecerrorFree(&sec_error);
ERROR_CHECK("UserAddSplineDimension", "ProSecerrorFree", status);

/*=====*\
    Create the section dimension
\*=====*/

point_types[0] = PRO_ENT_START;
point_types[1] = PRO_ENT_END;

/*=====*\
    Create the place point with spline's interpolation point
    coordinates to maintain scale for dimension placement.
\*=====*/

place_point[0] = pnt1[0];
place_point[1] = pnt2[1];

status = ProSecdimCreate(feats_section, ent_ids, point_types, 2,
    PRO_TK_DIM_PNT_PNT_HORIZ, place_point, &sec_dim_id);
ERROR_CHECK("UserAddSplineDimension", "ProSecdimCreate", status);

status = ProSecerrorAlloc(&sec_error);
ERROR_CHECK("UserAddSplineDimension", "ProSecerrorAlloc", status);

status = ProSectionSolve(feats_section, &sec_error);
ERROR_CHECK("UserAddSplineDimension", "ProSectionSolve", status);

if (status != PRO_TK_NO_ERROR)
    UserSecerrorPrint (&sec_error);

status = ProSecerrorFree(&sec_error);
ERROR_CHECK("UserAddSplineDimension", "ProSecerrorFree", status);

status = ProSecerrorAlloc(&sec_error);
ERROR_CHECK("UserAddSplineDimension", "ProSecerrorAlloc", status);

status = ProSectionRegenerate(feats_section, &sec_error);
ERROR_CHECK("UserAddSplineDimension", "ProSectionRegenerate", status);

if (status != PRO_TK_NO_ERROR)
    UserSecerrorPrint (&sec_error);

```

```

    status = ProSecerrorFree(&sec_error);
    ERROR_CHECK("UserAddSplineDimension", "ProSecerrorFree", status);

/*=====*\
    Set the modified section pointer back to feature and redefine it.
\*=====*/

    status = UserSetFeatureSection(&sk_feat, &feat_section);
    ERROR_CHECK("UserAddSplineDimension", "UserSetFeatureSection", status);

    status = ProMessageDisplay (MSGFILE,
        "USER Spline dimension between interpolation points 1 and
        %0d has been created.", &no_of_points);
    ERROR_CHECK("UserAddSplineDimension", "ProMessageDisplay", status);

    return (status);
}

/*=====*\
    FUNCTION : UserGetFeatureSection()
    PURPOSE  : Retrieve's the feature's section pointer.
\*=====*/

ProError UserGetFeatureSection(ProFeature* feat, ProSection *section)
{
    ProError status;
    ProElement elem_tree, elem;
    ProElempath elem_path;
    ProValue val;
    ProValueData val_data;

    ProElempathItem sketch_path_item[] = {
        {PRO_ELEM_PATH_ITEM_TYPE_ID, PRO_E_STD_SECTION},
        {PRO_ELEM_PATH_ITEM_TYPE_ID, PRO_E_SKETCHER}
    };

    status = ProFeatureElemtreeExtract (feat, NULL,
        PRO_FEAT_EXTRACT_NO_OPTS, &elem_tree );
    ERROR_CHECK("UserGetFeatureSection",
        "ProFeatureElemtreeExtract", status);

    status = ProElempathAlloc(&elem_path);
    ERROR_CHECK("UserGetFeatureSection", "ProElempathAlloc", status);

    status = ProElempathDataSet(elem_path, sketch_path_item, 2);
    ERROR_CHECK("UserGetFeatureSection", "ProElempathDataSet", status);

    status = ProElemtreeElementGet(elem_tree, elem_path, &elem);
    ERROR_CHECK("UserGetFeatureSection", "ProElemtreeElementGet", status);
}

```



```

    status = ProElementValueGet(elem, &val);
    ERROR_CHECK("UserGetFeatureSection", "ProElementValueGet", status);

    status = ProValueDataGet(val, &val_data);
    ERROR_CHECK("UserGetFeatureSection", "ProValueDataGet", status);

    *section = val_data.v.p;

    return status;
}

/*=====*\
FUNCTION   : UserCreatePoint()
PURPOSE    : Creates a section point entity at given coordinates
              in provided section.
\*=====*/

ProError UserCreatePoint (double x, double y, ProSection* section)
{
    ProError status;
    Pro2dPointdef point;
    ProWSecerror sec_error;
    int ent_id = 1;

    point.type = PRO_2D_POINT;
    point.pnt[0] = x;
    point.pnt[1] = y;

    status = ProSectionEntityAdd(*section,
                                (Pro2dEntdef *)&point, &ent_id);
    ERROR_CHECK("UserCreatePoint", "ProSectionEntityAdd", status);

    return status;
}

/*=====*\
FUNCTION   : UserSetFeatureSection()
PURPOSE    : Sets the section pointer to given feature and
              redefine it.
\*=====*/

ProError UserSetFeatureSection(ProFeature* feat, ProSection *section)
{
    ProError status;
    ProElement elem_tree, elem;
    ProElempath elem_path;
    ProFeatureCreateOptions opts[] = {PRO_FEAT_CR_NO_OPTS};
    ProErrorlist err_list;

```

```

ProElempathItem sketch_path_item[] = {
    {PRO_ELEM_PATH_ITEM_TYPE_ID, PRO_E_STD_SECTION},
    {PRO_ELEM_PATH_ITEM_TYPE_ID, PRO_E_SKETCHER}
};

status = ProFeatureElemtreeExtract (feat, NULL,
    PRO_FEAT_EXTRACT_NO_OPTS, &elem_tree );
ERROR_CHECK("UserSetFeatureSection",
    "ProFeatureElemtreeExtract",status);

status = ProElempathAlloc(&elem_path);
ERROR_CHECK("UserSetFeatureSection", "ProElempathAlloc",status);

status = ProElempathDataSet(elem_path, sketch_path_item, 2);
ERROR_CHECK("UserSetFeatureSection", "ProElempathDataSet",status);

status = ProElemtreeElementGet(elem_tree, elem_path, &elem);
ERROR_CHECK("UserSetFeatureSection", "ProElemtreeElementGet",status);

status = ProElementSpecialvalueSet(elem, (ProAppData)*section);
ERROR_CHECK("UserSetFeatureSection", "ProElementSpecialvalueSet",
    status);

status = ProElemtreeElementAdd(elem_tree, elem_path, elem);
ERROR_CHECK("UserSetFeatureSection", "ProElemtreeElementAdd",status);

status = ProFeatureRedefine(NULL, feat, elem_tree, opts, 1,
    &err_list);
ERROR_CHECK("UserSetFeatureSection", "ProFeatureRedefine",status);

return status;
}

```

Error Reporting

Functions Introduced:

- **ProSecerrorAlloc()**
- **ProSecerrorCount()**
- **ProSecerrorMsgGet()**
- **ProSecerrorItemGet()**
- **ProSecerrorAdd()**
- **ProSecerrorFree()**

Both **ProSectionSolve()** and **ProSectionRegenerate()** might result in a list of errors about the entities in the section. These errors are stored in an opaque structure called *ProWSecerror*. Before calling one of these functions, use **ProSecerrorAlloc()** to allocate memory for an error structure, then pass the pointer to the error structure to **ProSectionSolve()** or **ProSectionRegenerate()**.

You can add application-specific section errors to an error structure. To do this, call the function **ProSecerrorAdd()**.

The function **ProSecerrorCount()** tells you how many error messages are contained in the error structure. The errors themselves are identified by sequential integers, so you can step through the list. Use the function **ProSecerrorMsgGet()** to get the text of each message. Use the function **ProSecerrorItemGet()** to get the identifier of the problem entity that caused a specific error message. To free the allocated memory, call the function **ProSecerrorFree()**.

A Pro/TOOLKIT application that builds sections generally aims to make them complete and correct without any interactive help from the Pro/ENGINEER user. Therefore, the errors reported by the functions **ProSectionSolve()** and **ProSectionRegenerate()** are directed at the Pro/TOOLKIT developer as a debugging aid, rather than at the final Pro/ENGINEER user.

The following code fragment shows a call to **ProSectionSolve()** and an analysis of the errors produced.

```
ProWSecerror   errors;

int            n_errors, e;

ProError       status;

ProMsg         wmsg;

char           msg[PRO_PATH_SIZE];

int            ent_id;


ProSecerrorAlloc (&errors);

status = ProSectionSolve (section, &errors);

if (status != PRO_TK_NO_ERROR)
{
    ProSecerrorCount (&errors, &n_errors);
```

```

    for (e = 0; e < n_errors; e++)
    {
        ProSecerrorMsgGet (errors, e, wmsg);
        ProWstringToString (msg, wmsg);
        ProSecerrorItemGet (errors, e, &ent_id);
        printf ("%s: Problem ID, %d\n", msg, ent_id);
    }
    ProSecerrorFree (&errors);
    return (-1);
}

```

Retrieving and Saving a Section

Functions Introduced:

- **ProFeatureNumSectionsGet()**
- **ProFeatureSectionCopy()**

To retrieve a section from disk, use the function **ProMdlRetrieve()** with the model type **PRO_2DSECTION**. You can save a section to a file using the function **ProMdlSave()**.

You can also retrieve or copy a section from a feature. The function **ProFeatureNumSectionsGet()** finds the number of sections in the specified feature. Given a feature handle and section index, **ProFeatureSectionCopy()** initializes and returns a section handle to a section copied from the specified feature. Memory for this section is controlled by the Pro/TOOLKIT application and must therefore be freed by a call to **ProSectionFree()**.

Example 2: Creating a Section Model

The following example code illustrates how to use all the functions described in this chapter to create a section model.

```

/*=====*\
FILE      : UgSectModelCreate.c
PURPOSE   : Feature Creation
\*=====*/

/*----- Pro/Toolkit Includes -----*/
#include "ProToolkit.h"
#include "ProSection.h"
#include "ProSecdim.h"
#include "ProSecerror.h"

```

```
#include "Pro2dEntdef.h"
#include "ProSecdimType.h"
#include "ProMdl.h"
#include "UtilMath.h"
#include "ProMenu.h"
#include "ProUtil.h"

/*----- Application Includes -----*/
#include "TestError.h"

/*----- Function Prototypes -----*/
ProError UserSectionCreateExample();
ProError ProDemoSectCreate();

/*----- Global Data -----*/
typedef char ProUtilCname[PRO_NAME_SIZE];

/*=====*\
FUNCTION : ProDemoSectCreate()
PURPOSE : Creates a section model whose shape is a
          rectangle with a semi-circular "bite" from
          the left side.
\*=====*/

ProError ProDemoSectCreate(
    double      width,      /* (In) The rectangle width */
    double      height,     /* (In) The rectangle height */
    double      bite_radius, /* (In) The radius of the bite */
    double      bite_height, /* (In) The height of the bite */
    ProUtilCname name,      /* (In) The sketch name */
    ProBoolean   alloc,     /* (In) mem Alloc specifier */
    ProSection   *p_section) /* (In/Out) The section handle for the
                                sketch */
{
    ProError      status, solve_status;
    ProSection     section;
    ProName       wname;
    Pro2dArcdef   arc;
    Pro2dLinedef  line;
    int           bottom_id, right_id, top_id,
                left_top_id, left_bottom_id, arc_id;
    int           width_dim, height_dim, bite_height_dim,
                bite_radius_dim;
    ProSectionPointType pnt_types[2];
    Pro2dPnt       point;
    int           n_errors, e;
    ProWSecerror   errors;
    ProMsg         wmsg;
    char          msg[PRO_PATH_SIZE];
    int           dims[2];
/*-----*\
```

```

    Check that the dimensions are possible.
/*-----*/
    if (width < EPSM6 || height < EPSM6)
        return (-1);
    if (bite_height <= bite_radius)
        return (-1);
    if (bite_height + bite_radius > height)
        return (-1);
    if (bite_radius >= width)
        return (-1);
/*-----*\
    Allocate the handle for the 2-D section.
/*-----*/
    if (alloc == PRO_B_TRUE)
    {
status = ProSection2DAlloc (&section);
    }
    else
        section = *p_section;
/*-----*\
    Set the name of the section.
/*-----*/
    pro_str_to_wstr (wname, name);
    status = ProSectionNameSet (section, wname);
/*-----*\
    Set the epsilon value.
/*-----*/
    status = ProSectionEpsilonSet (section, 0.5);
/*-----*\
    Add a straight-line entity for the bottom of the rectangle.
/*-----*/
    line.type      = PRO_2D_LINE;
    line.end1[0]   = 0.0;
    line.end1[1]   = 0.0;
    line.end2[0]   = width + 0.1;
    line.end2[1]   = 0.0;
    status = ProSectionEntityAdd (section,
                                (Pro2dEntdef*)&line, &bottom_id);
/*-----*\
    ...right
/*-----*/
    line.type      = PRO_2D_LINE;
    line.end1[0]   = width + 0.1;
    line.end1[1]   = 0.0;
    line.end2[0]   = width;
    line.end2[1]   = height;
    status = ProSectionEntityAdd (section,
                                (Pro2dEntdef*)&line, &right_id);
/*-----*\
    ...top
/*-----*/

```

```

    line.type      = PRO_2D_LINE;
    line.end1[0] = width;
    line.end1[1] = height;
    line.end2[0] = 0.0;
    line.end2[1] = height;
    status = ProSectionEntityAdd (section, (Pro2dEntdef*)&line,
                                &top_id);
/*-----*\
    ...left above the bite
\*-----*/
    line.type      = PRO_2D_LINE;
    line.end1[0] = 0.0;
    line.end1[1] = bite_height - bite_radius;
    line.end2[0] = 0.0;
    line.end2[1] = 0.0;
    status = ProSectionEntityAdd (section, (Pro2dEntdef*)&line,
                                &left_top_id);
/*-----*\
    ...left below the bite
\*-----*/
    line.type      = PRO_2D_LINE;
    line.end1[0] = 0.0;
    line.end1[1] = height;
    line.end2[0] = 0.0;
    line.end2[1] = bite_height + bite_radius;
    status = ProSectionEntityAdd (section, (Pro2dEntdef*)&line,
                                &left_bottom_id);
/*-----*\
    Add an arc entity for the bite itself.
\*-----*/
    arc.type        = PRO_2D_ARC;
    arc.center[0]   = 0.0;
    arc.center[1]   = bite_height;
    arc.start_angle = PI * 1.5;    /* 270 degrees counterclockwise
                                   from the X axis */
    arc.end_angle   = PI / 2.0;    /* 90 degrees counterclockwise
                                   from the X axis */
    arc.radius       = bite_radius;
    status = ProSectionEntityAdd (section, (Pro2dEntdef*)&arc,
                                &arc_id);
/*-----*\
    Add a dimension for the width (the length of the top
    entity).
\*-----*/
    point[0] = width/2.0;
    point[1] = height + 10.0;
    pnt_types[0] = PRO_ENT_WHOLE;
    status = ProSecdimCreate (section, &top_id, pnt_types, 1,
                             PRO_TK_DIM_LINE, point, &width_dim);
/*-----*\
    Add a dimension for the height (the length of the

```

```

    right entity).
/*-----*/
    point[0] = width + 1.0;
    point[1] = height/2.0;
    pnt_types[0] = PRO_ENT_WHOLE;
    status = ProSecdimCreate (section, &right_id, pnt_types, 1,
                             PRO_TK_DIM_LINE, point, &height_dim);
/*-----*\
    Add a dimension for the height of the bite.
/*-----*/
    point[0] = -1.0;
    point[1] = bite_height/2.0;
    dims[0] = bottom_id;
    dims[1] = arc_id;
    pnt_types[0] = PRO_ENT_WHOLE;
    pnt_types[1] = PRO_ENT_CENTER;
    status = ProSecdimCreate (section, dims, pnt_types, 2,
                             PRO_TK_DIM_LINE_POINT, point, &bite_height_dim);
/*-----*\
    Add a dimension for the radius of the bite.
/*-----*/
    point[0] = bite_radius + 1.0;
    point[1] = bite_height;
    pnt_types[0] = PRO_ENT_WHOLE;
    status = ProSecdimCreate (section, &arc_id, pnt_types, 1,
                             PRO_TK_DIM_RAD, point, &bite_radius_dim);
/*-----*\
    Claim memory for the error structure.
/*-----*/
    status = ProSecerrorAlloc (&errors);
/*-----*\
    Solve the section.
/*-----*/
    solve_status = ProSectionSolve (section, &errors);
/*-----*\
    If the solve failed, report error messages and exit.
/*-----*/
    if (solve_status != PRO_TK_NO_ERROR)
    {
        status = ProSecerrorCount (&errors, &n_errors);
        for (e = 0; e < n_errors; e++)
        {
            status = ProSecerrorMsgGet (errors, e, wmsg);
            ProWstringToString (msg, wmsg);
            ProTKPrintf ("Error %d message : %s\n", e, msg);
        }
        return (-1);
    }
    status = ProSecerrorFree (&errors);
/*-----*\
    Save the section.

```



```

\*-----*/
    status = ProMdlSave (section);
\*-----*/
    Return the section handle
\*-----*/
    *p_section = section;
    return (0);
}

/*=====*\
FUNCTION : UserSectionCreateExample()
PURPOSE  : Invokes the function to create the Section model
\*=====*/

ProError UserSectionCreateExample()
{
    ProError status;
    double width;
    double height;
    double bite_radius;
    double bite_height;
    char *name = "test";
    ProBoolean alloc;
    char filename[PRO_NAME_SIZE];
    ProSection section;
    int win_id;
    ProMacro wmacro;
    char *macro = "#MODE;#SKETCHER;#SEARCH/RETR;#IN SESSION;#TEST.SEC;";

    width = 200;
    height = 150;
    bite_radius = 20;
    bite_height = 30;
    alloc = PRO_B_TRUE;

    status = ProDemoSectCreate( width, height, bite_radius,
                               bite_height, name, alloc, &section );
    ERROR_CHECK( "UserSectionCreateExample", "ProDemoSectCreate", status
);

    ProStringToWstring( wmacro, macro );
    ProMacroLoad(wmacro);

    return ( PRO_TK_NO_ERROR );
}

```


Element Trees: Sketched Features

This chapter describes the Pro/TOOLKIT functions that enable you to create and manipulate sketched features.

Sketched features are features that require one or more sections to completely define the feature, such as extruded and revolved protrusions.

This chapter outlines the necessary steps to programmatically create sketched features using Pro/TOOLKIT.

Topic	Page
Overview	41 - 2
Creating Features Containing Sections	41 - 4
Creating Features with 2D Sections	41 - 5
Verifying Section Shapes	41 - 5
Creating Features with 3D Sections	41 - 6
Reference Entities and Use Edge	41 - 8
Reusing Existing Sketches	41 - 10

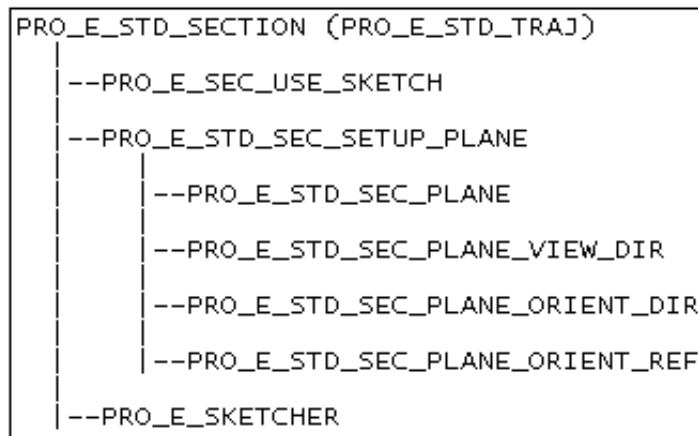
Overview

The chapter *Element Trees: Principles of Feature Creation* explains how to create a simple feature using the feature element tree, and the chapter on *Element Trees: Sections* explains how to create a section. This chapter explains how to put these methods together, with a few additional techniques, to create features that contain sketched sections.

Element Tree for Sketched Features

The element tree of any feature that contains a sketch contains a subtree that identifies the sketch object and describes how it is positioned in the model. As this subtree is the same for every sketched feature, it is documented in its own header file, called *ProStdSection.h*. The diagram below shows the structure of that subtree.

Figure 41-1: Element Tree for Sketched Features



The subtree of the `PRO_E_STD_SEC_SETUP_PLANE` element defines the sketch plane, the location of the sketch plane, the orientation plane and the orientation direction, and the viewing direction. You can use Intent Planes as sketch orientation or placement references.

The element `PRO_E_SKETCHER` is of type `POINTER`, and its value is the object *ProSection*, introduced in the chapter on Sections.

The element PRO_E_SEC_USE_SKETCH refers to any valid selected 'Sketch' (sketched datum curve) suitable to the current Sketch Based Feature. When this element is used, the sketch will be stored in the feature as a reference to the sketch feature, and no internal section, sketch plane or orientation will be required. Using this element in the PRO_E_STD_SECTION element tree allows a feature to be created in one step, without creating the feature first as incomplete.

Note: The use of internal sections, and the process by which an internal-section based feature is created, remains unchanged in Pro/ENGINEER Wildfire 2.0.

The following table shows the sketched features that are supported by Pro/TOOLKIT, the names of the corresponding header files which show their element trees, and the IDs of the elements in each tree which contain the standard sketch subtree as shown the Element Trees for Sketched Features.

Feature	Header	Element containing Subtree
Extrude	<i>ProExtrude.h</i>	PRO_E_STD_SECTION
Revolve	<i>ProRevolve.h</i>	PRO_E_STD_SECTION
Rib	<i>ProRib.h</i>	PRO_E_STD_SECTION
Hole	<i>ProHole.h</i>	PRO_E_SKETCHER (2D)
Fill (Flat datum surface)	<i>ProFlatSrf.h</i>	PRO_E_STD_SECTION
Draft	<i>ProDraft.h</i>	PRO_E_STD_SECTION
Sketched datum curve	<i>ProDtmCrv.h</i>	PRO_E_STD_SECTION
Sketched datum point	<i>ProDtmPnt.h</i>	PRO_E_STD_SECTION
Simple (constant) sweep	<i>ProSweep.h</i>	PRO_E_SWEEP_SPINE PRO_E_SWEEP_SECTION (2D)

Creating Features Containing Sections

The chapter Element Trees: Principles of Feature Creation explained that to create a feature from an element tree, you build the tree of elements using **ProElementAlloc()**, **ProElemtreeElementAdd()**, and so on, and then call **ProFeatureCreate()** to create the feature using the tree. If the feature is to contain a sketch, the sequence is a little more complex.

As explained in the chapter on Element Trees: Sections, a 2D section stored in a model file can be allocated by calling **ProSection2dAlloc()**. Instead, Pro/ENGINEER must allocate as part of the initial creation of the sketched feature, a section that will be part of a feature. The allocation is done by calling **ProFeatureCreate()** with an element tree which describes at minimum the feature type and form, in order to create an incomplete feature. In creating the feature, Pro/ENGINEER calculates the location and orientation of the section, and allocates the ProSection object. This section is then retrieved from the value of the PRO_E_SKETCHER element that is found in the element tree extracted from the created feature. Fill the empty section using ProSection related functions.

After adding the section contents and the remaining elements in the tree, add the new information to the feature using **ProFeatureRedefine()**.

To Create Sketched Features Element Trees

1. Build an element tree but do not include the element PRO_E_SKETCHER.
2. Call **ProFeatureCreate()** with the option PRO_FEAT_CR_INCOMPLETE_FEAT, so that the incomplete element tree is accepted.
3. Extract the value of the element PRO_E_SKETCHER created by Pro/ENGINEER from an element tree extracted using **ProFeatureElemtreeExtract()** on the incomplete feature.
4. Using that value as the ProSection object, create the necessary section entities and dimensions, and solve the section.
5. Add any other elements not previously added to the tree, such as extrusion depth. The depth elements may also be added before the creation of incomplete feature (before step 2).
6. Call **ProFeatureRedefine()** with the completed element tree.

Creating Features with 2D Sections

Sketched features using 2D sections do not require references to other geometry in the Pro/ENGINEER model. Some examples of where 2D sections are used are:

- Base features, sometimes called first features. This type of feature must be the first feature created in the model, and be of type PRO_FEAT_FIRST_FEAT.
- Sketched hole features.
- The PRO_E_SWEEP_SECTION section of a simple sweep feature.

To create 2D sketched features, follow the steps outlined in the section To Create Sketched Features Element Trees.

Note: For 2D sketched features, you need not specify section references or use projected 3D entities. Entities in a 2D section are dimensioned to themselves only. A 2D section does not require any elements in the tree to setup the sketch plane or the orientation of the sketch. Thus, the PRO_E_STD_SEC_SETUP_PLANE subtree is not included.

Element Trees:
Sketched Features

Verifying Section Shapes

Function Introduced:

- **ProSectionShapeGet()**

Certain features may or may not be able to use a section due to the shape of the section. Different sketched feature tools such as extrude and revolve have different requirements for sections. For example, solid protrusions contain only closed and non-intersecting sections. Solid cuts or datum surfaces have open non-intersecting sections. Fill features must have closed sections.

After the section is regenerated, **ProSectionShapeGet()** obtains the shape of a given section. This information is used to determine if the section is acceptable for feature creation.

The section shapes are as follows:

Constant	Function	Description
PRO_SECSHAPE_EMPTY	ProSectionShapeGet()	An empty section
PRO_SECSHAPE_POINTS	ProSectionShapeGet()	Section contains only sketched datum points
PRO_SECSHAPE_1_OPEN_LOOP	ProSectionShapeGet()	Section contains a single open loop (and possibly points)
PRO_SECSHAPE_1_CLOSED_LOOP	ProSectionShapeGet()	Section contains a single closed loop (and possibly points)
PRO_SECSHAPE_MIXED_LOOPS	ProSectionShapeGet()	Section contains at least one open and one closed loop (and possibly points)
PRO_SECSHAPE_MULTI_OPEN_LOOPS	ProSectionShapeGet()	Section contains multiple open loops (and possibly points)
PRO_SECSHAPE_MULTI_CLOSED_LOOPS	ProSectionShapeGet()	Section contains multiple closed loops (and possibly points)
PRO_SECSHAPE_INTERSECTING	ProSectionShapeGet()	Section contains loops that intersect each other (and possibly points)

Note: Use geometry entities and not construction entities to define section shapes that are then used to create solid or surface geometry. To convert the construction entities to geometry entities, use the function **ProSectionEntityConstructionSet()** with the input argument *construction* set to PRO_B_FALSE.

Creating Features with 3D Sections

A 3D section needs to define its location with respect to the existing geometrical features. The subtree contained in the element PRO_STD_SEC_SETUP_PLANE defines the location of the sketch planeEdge entities; any other 2D entities in the sketch must be dimensioned to those entities, so that their 3D location is fully defined.

3D Section Location in the Owing Model

Function Introduced:

- **ProSectionLocationGet()**

For a 2D section in a feature, Pro/ENGINEER decides where the section will be positioned in 3D.

If the section is 3D, the feature tree elements below PRO_E_STD_SEC_SETUP_PLANE specify the sketch plane, the direction from which it is being viewed, an orientation reference, and a direction which that reference represents (TOP, BOTTOM, LEFT or RIGHT). When you call **ProFeatureCreate()**, this information is used to calculate the 3D plane in which the section lies, and its orientation in that plane.

The position of the section origin in the plane is not implied by the element tree, and cannot be specified by the Pro/TOOLKIT application: position is chosen arbitrarily by Pro/ENGINEER. This is because the interactive user of Pro/ENGINEER never deals in absolute coordinates, and doesn't need to specify, or even know, the location of the origin of the section. In Pro/TOOLKIT describe all section entities in terms of their coordinate values, so you need to find out where Pro/ENGINEER has put the origin of the section. This is the role of the function **ProSectionLocationGet()**.

ProSectionLocationGet() provides the transformation matrix that goes from 2D coordinates within the section to 3D coordinates of the owning part or assembly. This is equivalent to describing the position and orientation of the 2D section coordinate system with respect to the base coordinate system of the 3D model.

So **ProSectionLocationGet()** can be called in order to calculate where to position new section entities so that they are in the correct 3D position in the part or assembly.

Reference Entities and Use Edge

Functions introduced:

- **ProSectionEntityFromProjection()**
- **ProSectionEntityIsProjection()**
- **ProSectionEntityUseEdge()**
- **ProSectionEntityUseEdgeLoop()**
- **ProSectionEntityUseEdgeChain()**
- **ProSectionEntityReferenceGet()**

The previous section explained how to set the correct 3D position of new section entities. You also need to make the entities parametric, that is, to ensure that Pro/ENGINEER knows how to calculate their new positions during regeneration.

When sketching a section using Pro/ENGINEER, entities are positioned parametrically by dimensioning them or aligning them to items in the 3D model. Pro/TOOLKIT does not allow you to explicitly align section entities, but you can add dimensions which relate section entities to 3D entities in the owning model. You can do this using references. A reference entity represents a position in the section of an item in a 3D model that is used as a dimension reference. The reference entity itself does not give rise to 3D geometry in the owning feature. Reference entities are visible in interactive sketcher operations; they are shown as dashed and are used during autodimensioning and alignment operations.

In Pro/TOOLKIT reference entities are created using **ProSectionEntityFromProjection()**. This function takes as input a ProSelection describing the 3D model entity being projected, and outputs the integer ID of the resulting known section entity. This ID is used to specify the attachment of a section dimension, as described in the chapter on “Element Trees: Sections”. Reference entities are included in the output of **ProSectionEntityIdsGet()**, but can be distinguished from regular section entities by calling the function **ProSectionEntityIsProjection()**.

To align a section entity with a 3D model entity, project the 3D entity to create a reference entity, and then either add a dimension between this reference entity and the one to be aligned or use **ProSectionAutodim()** to do this.

To create a regular section entity whose geometry is itself an exact projection of a 3D model entity, create it and align it in a single step using the function **ProSectionEntityUseEdge()**. This function has the same arguments as **ProSectionEntityFromProjection()**, and it creates a reference entity in the same way, but it requires an additional step of copying the reference entity to a regular entity with the same geometry. It outputs the ID of the regular entity it creates. The ID of the reference entity is always 1 less than the ID of the regular entity.

ProSectionEntityUseEdge() is equivalent to the Pro/ENGINEER sketcher command **Use Edge**. The functions **ProSectionEntityUseEdgeLoop()** and **ProSectionEntityUseEdgeChain()** allow you to execute a **Use Edge** operation on multiple edges simultaneously.

Note: If you create the known and projected entities first, you need not call **ProSectionLocationGet()** as described above; instead you can look at the geometry of the known and projected entities, and then position the new entities relative to the projected entities.

The function **ProSectionEntityReferenceGet()** provides 3D geometry that is a reference for a projected entity in a given section.

Creating Geometry by Offsetting

The functions described in this section enable you to create offset entities from edges and 3D curve segments from normal, chain and loop selection.

Functions Introduced:

- **ProSectionEntityUseOffset()**
- **ProSectionEntityUseOffsetChain()**
- **ProSectionEntityUseOffsetLoop()**

The function **ProSectionEntityUseOffset()** creates a sketched entity that is offset at a specified distance from a single edge. This function takes as input a ProSelection object describing the 3D model entity.

The function **ProSectionEntityUseOffsetChain()** creates sketched entities that are offset from a chain of edges or entities and the function **ProSectionEntityUseOffsetLoop()** creates sketched entities offset from a loop of edges or entities.

The behavior of the functions in this section is similar to the offset operation achieved using **Sketch>Edge>Offset** in Pro/ENGINEER.

Reusing Existing Sketches

Functions introduced:

- **ProFeatureSketchAdd()**
- **ProFeatureSketchedCreate()**

Pro/ENGINEER Wildfire allows you to copy sections from previously created features into new sketched features.

The function **ProFeatureSketchAdd()** copies the selected section from one feature to another feature.

The function **ProFeatureSketchedCreate()** includes the element tree feature creation along with the section copy operation. This reduces the sketched feature creation effort to a single Pro/TOOLKIT function call. The element tree must contain all of the required elements except the PRO_E_STD_SECTION subtree.

Example 1: Creating an Extruded Protrusion Base Feature

The following code fragment shows the creation of an extruded protrusion as a base feature.

```
/*=====*\
FILE      : UgSktExtrusionCreate.c
\*=====*/

/*----- Pro/Toolkit Includes -----*/
#include "ProToolkit.h"
#include "ProFeature.h"
#include "ProElemId.h"
#include "ProExtrude.h"
#include "ProModFeat.h"
#include "ProStdSection.h"
#include "ProElement.h"
#include "ProElempath.h"
#include "ProFeatType.h"
#include "ProFeatForm.h"
#include "ProSelection.h"
#include "ProSection.h"

static ProFileName message_file;

#define C_LOG(a) ProTKPrintf ("Status for %s is = %d\n", a, status ); \
ERROR_CHECK( a, "UgSktExtrusionCreate.c", status );
```

```

/*----- Function Prototypes -----*/
int ProDemoBaseExtrudeProtrCreate();

/*----- External Data -----*/
ProError ProDemoSectCreate();

/*----- Global Data -----*/
typedef struct tree_element_data
{
    ProElement    tree;
    ProElement    parent_element;
    ProElemId     elem_id;
    ProValueData  value_data;
} ProTreeElemdata;

ProError ProTestFeatElemAdd (ProTreeElemdata *elem);

/*=====*\
FUNCTION : ProDemoBaseExtrudeProtrCreate
PURPOSE  : Demonstrates the creation of the extruded protrusion
           base feature.
\*=====*/
int ProDemoBaseExtrudeProtrCreate()
{
    ProTreeElemdata    elem;
    ProErrorlist       errors;
    ProMdl             model;
    ProModelitem       model_item;
    ProSelection       model_sel;
    ProFeature         feature;
    ProFeatureCreateOptions opts[1];
    ProElempath        path;
    ProElempathItem    path_items[2];
    ProSection         section;
    ProAsmcomppath     comp_path;
    ProAsmcomppath     *p_comp_path = NULL;
    ProElement         parent_elem;
    ProElement         parent_elem2;
    ProValue           value;
    double             width;
    double             height;
    double             bite_radius;
    double             bite_height;
    char               name[PRO_NAME_SIZE];
    ProBoolean         alloc;
    ProError           status;
    ProSelection *     sketch_selection;
    ProSelection *     selection_array;
    int                n_select;

```

```

    ProStringToWstring ( message_file, "utilities.txt" );
/*-----*\
    Allocate the element tree.
\*-----*/
    status = ProElementAlloc (PRO_E_FEATURE_TREE, &(elem.tree));
    C_LOG(" ProElementAlloc");

/*-----*\
    Add the feature type element to the tree.
\*-----*/
    elem.parent_element = elem.tree;
    elem.elem_id = PRO_E_FEATURE_TYPE;
    elem.value_data.type = PRO_VALUE_TYPE_INT;
    elem.value_data.v.i = PRO_FEAT_PROTRUSION;
    status = ProTestFeatElemAdd (&elem);
    C_LOG(" ProTestFeatElemAdd");

/*-----*\
    Add the feature form element to the tree.
\*-----*/
    elem.parent_element = elem.tree;
    elem.elem_id = PRO_E_FEATURE_FORM;
    elem.value_data.type = PRO_VALUE_TYPE_INT;
    elem.value_data.v.i = PRO_EXTRUDE;
    status = ProTestFeatElemAdd (&elem);
    C_LOG(" ProTestFeatElemAdd");

/*-----*\
    Add the feature solid/surface/cut element to the tree.
\*-----*/
    elem.parent_element = elem.tree;
    elem.elem_id = PRO_E_EXT_SURF_CUT_SOLID_TYPE;
    elem.value_data.type = PRO_VALUE_TYPE_INT;
    elem.value_data.v.i = PRO_EXT_FEAT_TYPE_SOLID;
    status = ProTestFeatElemAdd (&elem);
    C_LOG(" ProTestFeatElemAdd");

/*-----*\
    Add the feature addition/removal material element to the tree.
\*-----*/
    elem.parent_element = elem.tree;
    elem.elem_id = PRO_E_REMOVE_MATERIAL;
    elem.value_data.type = PRO_VALUE_TYPE_INT;
    elem.value_data.v.i = PRO_EXT_MATERIAL_ADD;
    status = ProTestFeatElemAdd (&elem);
    C_LOG(" ProTestFeatElemAdd");

/*-----*\
    Add the feature thin element to the tree.
\*-----*/
    elem.parent_element = elem.tree;

```

```

elem.elem_id = PRO_E_FEAT_FORM_IS_THIN;
elem.value_data.type = PRO_VALUE_TYPE_INT;
elem.value_data.v.i = PRO_EXT_FEAT_FORM_NO_THIN;
status = ProTestFeatElemAdd (&elem);
C_LOG(" ProTestFeatElemAdd");

/*-----*\
Add the standard section element to the tree.
\*-----*/
elem.parent_element = elem.tree;
elem.elem_id = PRO_E_STD_SECTION;
elem.value_data.type = -1;
elem.value_data.v.i = -1;
status = ProTestFeatElemAdd (&elem);
C_LOG(" ProTestFeatElemAdd");

/*-----*\
Add the section depth elements to the element tree.
\*-----*/
elem.parent_element = elem.tree;
elem.elem_id = PRO_E_STD_EXT_DEPTH;
elem.value_data.type = -1;
elem.value_data.v.i = -1;
status = ProTestFeatElemAdd (&elem);
C_LOG(" ProTestFeatElemAdd");
parent_elem2 = elem.parent_element;

elem.elem_id = PRO_E_EXT_DEPTH_FROM;
elem.value_data.type = -1;
elem.value_data.v.i = -1;
status = ProTestFeatElemAdd (&elem);
C_LOG(" ProTestFeatElemAdd");
parent_elem = elem.parent_element;
elem.elem_id = PRO_E_EXT_DEPTH_FROM_TYPE;
elem.value_data.type = PRO_VALUE_TYPE_INT;
elem.value_data.v.i = PRO_EXT_DEPTH_FROM_BLIND;
status = ProTestFeatElemAdd (&elem);
C_LOG(" ProTestFeatElemAdd");
elem.parent_element = parent_elem;
elem.elem_id = PRO_E_EXT_DEPTH_FROM_VALUE;
elem.value_data.type = PRO_VALUE_TYPE_DOUBLE;
elem.value_data.v.d = 50.0;
status = ProTestFeatElemAdd (&elem);
C_LOG(" ProTestFeatElemAdd");

elem.parent_element = parent_elem2;
elem.elem_id = PRO_E_EXT_DEPTH_TO;
elem.value_data.type = -1;
elem.value_data.v.i = -1;
status = ProTestFeatElemAdd (&elem);
C_LOG(" ProTestFeatElemAdd");

```

```

parent_elem = elem.parent_element;
elem.elem_id = PRO_E_EXT_DEPTH_TO_TYPE;
elem.value_data.type = PRO_VALUE_TYPE_INT;
elem.value_data.v.i = PRO_EXT_DEPTH_TO_BLIND;
status = ProTestFeatElemAdd (&elem);
C_LOG(" ProTestFeatElemAdd");
elem.parent_element = parent_elem;
elem.elem_id = PRO_E_EXT_DEPTH_TO_VALUE;
elem.value_data.type = PRO_VALUE_TYPE_DOUBLE;
elem.value_data.v.d = 0.0;
status = ProTestFeatElemAdd (&elem);
C_LOG(" ProTestFeatElemAdd");

status = ProMdlCurrentGet (&model);
C_LOG(" ProMdlCurrentGet");
if ( status != PRO_TK_NO_ERROR ) return ( status );
status = ProMdlToModelitem( model, &model_item );
C_LOG(" ProMdlToModelitem");
status = ProSelectionAlloc (p_comp_path, &model_item, &model_sel);
C_LOG(" ProSelectionAlloc");

status = ProMessageDisplay ( message_file,
"Select a Sketched Datum Curve Feature");
C_LOG(" ProMessageDisplay");
ProTKPrintf ( "Please Select a Sketched Datum Curve Feature \n");

status = ProSelect ( "feature", 1, NULL, NULL, NULL, NULL,
&sketch_selection, &n_select );
C_LOG(" ProSelect");

if ( n_select <= 0 ) return -1;

status = ProArrayAlloc ( n_select, sizeof (ProSelection),
1, ( ProArray *)&selection_array );
C_LOG(" ProArrayAlloc");

status = ProSelectionCopy ( sketch_selection[0], &selection_array[0]
);
C_LOG(" ProSelectionCopy");

/*-----*\
Create the protrusion using the section of the selected feature
\*-----*/

opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMS;
status = ProFeatureSketchedCreate ( model_sel, elem.tree,
opts, 1, selection_array,
&feature, &errors );
C_LOG(" ProFeatureSketchedCreate");

/*-----*\

```



```

    Free up the allocated memory.
/*-----*/
    status = ProElementFree (&elem.tree);
    C_LOG(" ProElementFree");

    return (status);
}

/*=====*\
FUNCTION : ProTestFeatElemAdd
PURPOSE  : Adds a generic feature element to the element tree.
/*=====*/
ProError ProTestFeatElemAdd (ProTreeElemdata *elem)
{
    ProValue    value;
    ProElement  element;
    ProError    status;
    if (elem->value_data.type != -1)
    {
        status = ProValueAlloc (&value);
        C_LOG(" ProValueAlloc");
        status = ProValueDataSet (value, &elem->value_data);
        C_LOG(" ProValueDataSet");
    }
    status = ProElementAlloc (elem->elem_id, &element);
    C_LOG(" ProElementAlloc");

    if (elem->value_data.type != -1)
    {
        status = ProElementValueSet (element, value);
        C_LOG(" ProElementValueSet");
    }
    status = ProElemtreeElementAdd (elem->parent_element, NULL, element);
    C_LOG(" ProElemtreeElementAdd");
    elem->parent_element = element;
    return (status);
}

```

Example 2: Creating a Sketched Datum Curve

The following code fragment shows the creation of a sketched datum curve using the conventional approach.

```

/*----- Pro/Toolkit Includes -----*/
#include "ProToolkit.h"
#include "ProFeature.h"
#include "ProElemId.h"
#include "ProExtrude.h"
#include "ProModFeat.h"
#include "ProStdSection.h"
#include "ProElement.h"

```

```

#include "ProElempath.h"
#include "ProFeatType.h"
#include "ProFeatForm.h"
#include "ProSelection.h"
#include "ProSection.h"
#include <PTApplsUnicodeUtils.h>

#include "ProDtmCrv.h"

static ProFileName message_file;
static FILE * log_file;

#define C_LOG(a) ProTKPrintf ("Status for %s is = %d\n", a, status ); \
ERROR_CHECK( a, "UgSktExtrusionCreate.c", status );

#define C_PRINT(a) ProTKPrintf ( "%s\n", a);

/*----- Function Prototypes -----*/
ProError ProDemoSketchedCurveCreate();

/*----- External Data -----*/
ProError ProDemoSectCreate();

/*----- Global Data -----*/

/*=====*\
FUNCTION : ProDemoSketchedCurveCreate
PURPOSE : Demonstrates the creation of the extruded protrusion
          base feature.
\*=====*/
ProError ProDemoSketchedCurveCreate()
{
    ProErrorlist          errors;
    ProMdl                model;
    ProModelitem          model_item;
    ProSelection          model_sel;
    ProFeature            feature;
    ProFeatureCreateOptions opts[1];
    ProElempath           path;
    ProElempathItem       path_items[2];
    ProSection            section;
    ProAsmcomppath        comp_path;
    ProAsmcomppath        *p_comp_path = NULL;
    ProValue              value;
    double                width;
    double                height;
    double                bite_radius;
    double                bite_height;
    char                  name[PRO_NAME_SIZE];
    ProBoolean            alloc;

```

```

ProElement sketch_element;
ProElement created_elemtree;

ProElement pro_e_feature_tree;
ProElement pro_e_feature_type;
ProElement pro_e_curve_type;
ProElement pro_e_std_section;
ProElement pro_e_std_sec_method;
ProElement pro_e_std_sec_setup_plane;
ProElement pro_e_std_sec_plane;
ProElement pro_e_sketcher;
ProElement pro_e_std_sec_plane_view_dir;
ProElement pro_e_std_sec_plane_orient_dir;
ProElement pro_e_std_sec_plane_orient_ref;

ProSelection *sketch_refs;

ProName wide_string;
ProError status;
ProValueData value_data;
ProSelection * p_select;
int n_select;
ProBoolean is_interactive = PRO_B_TRUE;

ProStringToWstring ( message_file, "utilities.txt" );

log_file = PTApplsUnicodeFopen ( "ug_sketched_curve.log", "w" );

/*-----*\
   Populating root element PRO_E_FEATURE_TREE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TREE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TREE, &pro_e_feature_tree );
C_LOG( " ProElementAlloc " );

/*-----*\
   Populating element PRO_E_FEATURE_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TYPE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TYPE, &pro_e_feature_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_FEAT_CURVE; /* 949 */
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_feature_type, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_feature_type );

```

```

C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_CURVE_TYPE
\*-----*/

C_PRINT( " *** Processing Element PRO_E_CURVE_TYPE *** " );
status = ProElementAlloc ( PRO_E_CURVE_TYPE, &pro_e_curve_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_CURVE_TYPE_SKETCHED; /* 0 */
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_curve_type, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_curve_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_SECTION
\*-----*/

C_PRINT( " *** Processing Element PRO_E_STD_SECTION *** " );
status = ProElementAlloc ( PRO_E_STD_SECTION, &pro_e_std_section );
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_std_section );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Not required to populate PRO_E_STD_SEC_METHOD
\*-----*/

/*
C_PRINT( " *** Processing Element PRO_E_STD_SEC_METHOD *** " );
status = ProElementAlloc ( PRO_E_STD_SEC_METHOD,
&pro_e_std_sec_method );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_SEC_SKETCH ** 25;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_sec_method, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_std_section, NULL,
pro_e_std_sec_method );

```

```

C_LOG( " ProElemtreeElementAdd" );

*/
/*-----*\
    Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SEC_SETUP_PLANE *** " );
status = ProElementAlloc ( PRO_E_STD_SEC_SETUP_PLANE,
&pro_e_std_sec_setup_plane );
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_std_section, NULL,
pro_e_std_sec_setup_plane );
C_LOG( " ProElemtreeElementAdd" );

sketch_refs = ( ProSelection *) calloc ( 2, sizeof ( ProSelection ));

/*-----*\
    Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    -> PRO_E_STD_SEC_PLANE
\*-----*/

C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE *** " );
status = ProMessageDisplay ( message_file, "Select Surface for sketch
placement");
C_LOG( " ProMessageDisplay" );
ProTKPrintf ( "Please select datum,surface,sldface,qltface_ID_5 type of
Modelitem\n");
status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL, NULL,
NULL, NULL, &p_select, &n_select );
C_LOG( " ProSelect" );
if ( n_select <= 0 ) return -1;
else
{
    status = ProSelectionCopy( p_select[0], &sketch_refs[0]);
    C_LOG ( " ProSelectionCopy " );
}
status = ProElementAlloc ( PRO_E_STD_SEC_PLANE, &pro_e_std_sec_plane );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_sec_plane, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane );
C_LOG( " ProElemtreeElementAdd" );

```

```

/*-----*\
    Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    -> PRO_E_STD_SEC_PLANE_VIEW_DIR
/*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_VIEW_DIR *** " );
status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_VIEW_DIR,
&pro_e_std_sec_plane_view_dir );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_SEC_VIEW_DIR_SIDE_ONE; /* 1 */
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_std_sec_plane_view_dir, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_view_dir );
C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    -> PRO_E_STD_SEC_PLANE_ORIENT_DIR
/*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_ORIENT_DIR *** "
);
status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_ORIENT_DIR,
&pro_e_std_sec_plane_orient_dir );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_SEC_ORIENT_DIR_UP; /* 1 */
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_std_sec_plane_orient_dir, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_orient_dir );
C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    -> PRO_E_STD_SEC_PLANE_ORIENT_REF
/*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_ORIENT_REF *** "
);

```

```

    status = ProMessageDisplay ( message_file,
"Select Surface for sketch orientation");
    C_LOG( " ProMessageDisplay" );
    ProTKPrintf ( "Select datum,surface,sldface,qltface_ID_5 type of
Modelitem\n");
    status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL,
NULL, NULL, NULL, &p_select, &n_select );
    C_LOG( " ProSelect" );
    if ( n_select <= 0 ) return -1;
    else
    {
        status = ProSelectionCopy( p_select[0], &sketch_refs[1]);
        C_LOG ( " ProSelectionCopy " );
    }

    status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_ORIENT_REF,
&pro_e_std_sec_plane_orient_ref );
    C_LOG( " ProElementAlloc " );
    value_data.type = PRO_VALUE_TYPE_SELECTION;
    value_data.v.r = p_select[0];
    status = ProValueAlloc ( &value );
    C_LOG( " ProValueAlloc" );
    status = ProValueDataSet ( value, &value_data );
    C_LOG( " ProValueDataSet" );
    status = ProElementValueSet ( pro_e_std_sec_plane_orient_ref, value );
    C_LOG( " ProElementValueSet" );
    status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_orient_ref );
    C_LOG( " ProElemtreeElementAdd" );

    /*-----*\
        Creating incomplete feature in the current model.
    \*-----*/
    status = ProMdlCurrentGet (&model);
    if ( status != PRO_TK_NO_ERROR ) return ( status );
    status = ProMdlToModelitem( model, &model_item );
    status = ProSelectionAlloc (p_comp_path, &model_item,
        &model_sel);

    opts[0] = PRO_FEAT_CR_INCOMPLETE_FEAT;
    status = ProFeatureCreate (model_sel, pro_e_feature_tree, opts, 1,
&feature, &errors);
    C_LOG ( " ProFeatureCreate");

    /* Using the element tree from created feature */
    status = ProFeatureElemtreeExtract (&feature, NULL,
        PRO_FEAT_EXTRACT_NO_OPTS,
        &created_elemtree);
    C_LOG ( "ProFeatureElemtreeExtract()");

    /*-----*\

```

```

    Getting the initialized section element from the database.
/*-----*/
/* path to PRO_E_SKETCHER element */
path_items[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[0].path_item.elem_id = PRO_E_STD_SECTION;
path_items[1].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[1].path_item.elem_id = PRO_E_SKETCHER;
status = ProElempathAlloc (&path);
status = ProElempathDataSet (path, path_items, 2);

status = ProElemtreeElementGet ( created_elemtree, path,
&sketch_element);
C_LOG (" ProElemtreeElementGet PRO_E_SKETCHER ");

status = ProElementValueGet ( sketch_element, &value);
C_LOG (" ProElementValueGet PRO_E_SKETCHER ");

status = ProValueDataGet (value, &value_data);
C_LOG (" ProValueDataGet PRO_E_SKETCHER ");
section = (ProSection)value_data.v.p;

/*-----*\
    Creating a 3-D section
/*-----*/

status = UserSectionBuild ( ( ProSection )(value_data.v.p),
    sketch_refs );
C_LOG ("UserSectionBuild");

/*-----*\
    Redefining the feature to make it complete.
/*-----*/
opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMS;
status = ProSelectionAsmcomppathGet (model_sel, &comp_path);
C_LOG (" ProSelectionAsmcomppathGet");

status = ProFeatureRedefine (&comp_path, &feature, created_elemtree,
    opts, 1, &errors);
C_LOG (" ProFeatureRedefine");

/*-----*\
    Free up the allocated memory.
/*-----*/
status = ProFeatureElemtreeFree (&feature, created_elemtree );
C_LOG (" ProFeatureElemtreeFree");

status = ProElementFree (&pro_e_feature_tree );
C_LOG (" ProElementFree");

return (status);
}

```


Element Trees: Extrude and Revolve

This chapter describes how to use the include files *ProExtrude.h*, and *ProRevolve.h* so that you can create extruded and revolved features programmatically. As Extrude and Revolve features are sketched features; we recommend you to read the chapters 'Element Trees: Principles of Feature Creation' and 'Element Trees: Sketched Features' before referring to this chapter.

Topic	Page
The Element Tree for Extruded Features	42 - 2
The Element Tree for Revolved Features	42 - 68
The Element Tree for First Features	42 - 106

The Element Tree for Extruded Features

The element tree for extrude features is documented in the header file *ProExtrude.h*. The functions **ProFeatureTypeGet()** and **ProFeatureSubtypeGet()** return an Extrude feature. The types of Extrude features are:

- Protrusion
- Cut
- Surface
- Surface Trim
- Thin Protrusion
- Thin Cut
- Sheetmetal Cut
- Sheetmetal Unattached Wall

Refer to the chapter Production Applications: Sheetmetal for element details on sheetmetal features.

The extrude element tree contains toggles to switch between different feature types. An extruded feature tree also contains subtrees supporting the section and depth parameters for the feature.

You can use Intent Datums such as Intent Point, Intent Axis, and Intent Plane for depth reference in extrude features.

The following figure shows the element tree for extruded features.

Figure 42-1: The Element Tree for Extruded Feature

```
PRO_E_FEATURE_TREE
|
|--PRO_E_STD_FEATURE_NAME
|
|--PRO_E_EXT_SURF_CUT_SOLID_TYPE
|
|--PRO_E_REMOVE_MATERIAL
|
|--PRO_E_IS_SMT_CUT
|
|--PRO_E_SMT_CUT_NORMAL_DIR
|
|--PRO_E_FEATURE_FORM
|
|--PRO_E_STD_SECTION
|
|--PRO_E_FEAT_FORM_IS_THIN
|
|--PRO_E_STD_MATRLSIDE
|
|--PRO_E_THICKNESS
|
|--PRO_E_SRF_END_ATTRIBUTES
|
|--PRO_E_TRIM_QLT_SIDE
|
|--PRO_E_TRIM_QUILT
|
|--PRO_E_STD_DIRECTION
|
|--PRO_E_STD_EXT_DEPTH
|
|--PRO_E_STD_SMT_THICKNESS
|
|--PRO_E_STD_SMT_SWAP_DRV_SIDE
|
|--PRO_E_SMT_WALL_SHARPS_TO_BENDS
|
|--PRO_E_SMT_FILLETS
|
|--PRO_E_SMT_DEV_LEN_CALCULATION
```

Element Trees:
Extrude and Revolve

The elements are assigned values depending on the type of extrusion you want to create.

The following table lists the common elements for all types of extrusions and their permissible values:

Element ID	Value
PRO_E_FEATURE_TYPE	Feature type, not required for creation: PRO_FEAT_PROTRUSION PRO_FEAT_CUT PRO_FEAT_DATUM_SURFN
PRO_E_FEATURE_FORM	Mandatory = PRO_EXTRUDE
PRO_E_EXT_SURF_CUT_SOLID_TYPE	Mandatory Of type ProExtFeatType = PRO_EXT_FEAT_TYPE_SOLID for Solid feature type = PRO_EXT_FEAT_TYPE_SURFACE for Surface feature type
PRO_E_FEAT_FORM_IS_THIN	Feature Form Of Type ProExtFeatForm = PRO_EXT_FEAT_FORM_NO_THIN for a feature that is not Thin = PRO_EXT_FEAT_FORM_THIN for a Thin feature
PRO_E_REMOVE_MATERIAL	Material Removal Of type ProExtRemMaterial = PRO_EXT_MATERIAL_ADD for a Protruded feature = PRO_EXT_MATERIAL_REMOVE for a Cut feature
PRO_E_STD_SECTION	Standard section elements
PRO_E_STD_DIRECTION*	Direction of creation. Of type ProExtDirection = PRO_EXT_CR_IN_SIDE_ONE for depth in side one = PRO_EXT_CR_IN_SIDE_TWO for depth in side two
PRO_E_STD_MATRLSIDE*	Direction of material affected with respect to the sketch. Required for all cuts, all thin features, and for solid protrusions with open sections.
PRO_E_STD_EXT_DEPTH	Compound Element

Element ID	Value
PRO_E_EXT_DEPTH_TO	Compound Element
PRO_E_EXT_DEPTH_TO_TYPE	Mandatory Of type ProExtDepthToType
PRO_E_EXT_DEPTH_TO_VALUE	Depends on PRO_E_EXT_DEPTH_TO_TYPE. Of type PRO_VALUE_TYPE_DOUBLE
PRO_E_EXT_DEPTH_TO_REF	Depends on PRO_E_EXT_DEPTH_TO_TYPE. Of type listed in the Depth Type table that follows.
PRO_E_EXT_DEPTH_FROM	Compound Element
PRO_E_EXT_DEPTH_FROM_TYPE	Mandatory Of type ProExtDepthFromType
PRO_E_EXT_DEPTH_FROM_VALUE	Depends on PRO_E_EXT_DEPTH_FROM_TYPE. Of type PRO_VALUE_TYPE_DOUBLE
PRO_E_EXT_DEPTH_FROM_REF	Depends on PRO_E_EXT_DEPTH_FROM_TYPE Of type listed in the Depth Type table that follows.
PRO_E_STD_FEATURE_NAME	Default given by application depending on the feature type. Can be modified by the user.

Elements identified with “*” depend on the definition of the standard section. These elements are not assigned values until the standard section has been completely allocated (which typically happens during redefine of the feature). Values assigned to these elements while the section is not complete are ignored.

The following table lists the Depth Types for extruded features along with possible valid references:

Depth Type	Valid Reference Types
PRO_EXT_DEPTH_TO_REF	PRO_SURFACE, PRO_AXIS, PRO_EDGE, PRO_CURVE, PRO_POINT, PRO_EDGE_START, PRO_EDGE_END, PRO_CRV_START and PRO_CRV_END.

Depth Type	Valid Reference Types
PRO_EXT_DEPTH_FROM_REF	PRO_SURFACE, PRO_AXIS, PRO_EDGE, PRO_CURVE, PRO_POINT, PRO_EDGE_START, PRO_EDGE_END, PRO_CRV_START and PRO_CRV_END.

The following table lists the elements needed to create extruded features, in addition to those already discussed:

Feature Type	Element ID	Comment
Solid		No Additional Elements Required
Thin	PRO_E_THICKNESS	Mandatory ≥ 0.0 Of type PRO_VALUE_TYPE_DOUBLE
Solid Cut	PRO_E_STD_MATRLSIDE	Mandatory Of type ProExtMatlSide
Thin Cut	PRO_E_STD_MATRLSIDE	Mandatory Of type ProExtMatlSide
	PRO_E_THICKNESS	Mandatory ≥ 0.0 Of type PRO_VALUE_TYPE_DOUBLE
Surface	PRO_E_SRF_END_ATTRIBUTES	Mandatory Of type ProExtSurfEndAttr It must be assigned at the same time or after the section is fully completed.
Surface Trim	PRO_E_STD_MATRLSIDE	Mandatory Of type ProExtMatlSide
	PRO_E_TRIM_QUILT	Mandatory Of type Quilt
	PRO_E_TRIM_QLT_SIDE	Mandatory Of type ProExtTrimQltSide
	PRO_E_SRF_END_ATTRIBUTES	Mandatory Of type ProExtSurfEndAttr

Feature Type	Element ID	Comment
Thin Surface Trim	PRO_E_STD_MATRLSIDE	Mandatory Of type ProExtMatlSide
	PRO_E_THICKNESS	Mandatory ≥ 0.0 Of type PRO_VALUE_TYPE_DOUBLE
	PRO_E_TRIM_QUILT	Mandatory Of type Quilt
	PRO_E_TRIM_QLT_SIDE	Mandatory Of type ProExtTrimQltSide if PRO_E_STD_MATRLSIDE is "both". Must be assigned at the same time as PRO_E_STD_MATRLSIDE.
	PRO_E_SRF_END_ATTRIBUTES	Mandatory Of type ProExtSurfEndAttr

Examples: Creating Extruded Features

The following examples demonstrate creation of extrude features of various forms. These examples are adapted from an example template file *UgSktExtrusionTemplate.c* available on the Pro/ENGINEER load point under
protoolkit/protk_appls/pt_userguide/ptu_featcreat.

- Example 1: Creating an Extruded Feature
- Example 2: To Create an Extruded Cut with Two-sided Thru-all Depth
- Example 3: To Create an Extruded Thin Cut
- Example 4: To Create an Extruded Datum Surface Feature
- Example 5: To Create a Surface Trim Extruded Feature

Conventional Approach

Example 1: Creating an Extruded Feature

The example shows how to create a Extruded Protrusion by the conventional approach for sketched features. The example creates an incomplete feature using **ProFeatureCreate()**, extracts the section from the element tree of the incomplete feature, builds the section on the section handle obtained, and, completes the feature using **ProFeatureRedefine()**.

The user is prompted to select the sketching and the orientation planes and then the reference edges for the sketch. The user is also required to enter the X and Y offsets to be applied to the sketch from the projected edges.

The source code for the application is as follows:

```
/*----- Pro/Toolkit Includes -----*/
#include "ProToolkit.h"
#include "ProFeature.h"
#include "ProElemId.h"
#include "ProExtrude.h"
#include "ProModFeat.h"
#include "ProStdSection.h"
#include "ProElement.h"
#include "ProElemPath.h"
#include "ProFeatType.h"
#include "ProFeatForm.h"
#include "ProSelection.h"
#include "ProSection.h"

#include "ProExtrude.h"

static ProFileName message_file;

#define C_LOG(a) ProTKPrintf ("Status for %s is = %d\n", a, status ); \
ERROR_CHECK( a, "UgSkExtrusionCreate.c", status );

#define C_PRINT(a) ProTKPrintf ( "%s\n", a);

/*=====*\
FUNCTION : UserSkExtrusionProtrusion
PURPOSE  : Demonstrates the creation of the extruded protrusion
           base feature.
\*=====*/
ProError UserSkExtrusionProtrusion()
{
    ProErrorlist      errors;
    ProMdl            model;
    ProModelitem      model_item;
```



```

ProSelection      model_sel;
ProFeature        feature;
ProFeatureCreateOptions opts[1];
ProElemppath      path;
ProElemppathItem  path_items[2];
ProSection        section;
ProAsmcomppath    comp_path;
ProAsmcomppath    *p_comp_path = NULL;
ProValue          value;

ProElement sketch_element;
ProElement created_elemtree;

ProElement pro_e_feature_tree;
ProElement pro_e_feature_form;
ProElement pro_e_ext_surf_cut_solid_type;
ProElement pro_e_remove_material;
ProElement pro_e_feat_form_is_thin;
ProElement pro_e_std_direction;
ProElement pro_e_std_matrlside;

ProElement pro_e_std_ext_depth;
ProElement pro_e_ext_depth_from;
ProElement pro_e_ext_depth_from_type;
ProElement pro_e_ext_depth_from_value;
ProElement pro_e_ext_depth_to;
ProElement pro_e_ext_depth_to_type;

ProElement pro_e_std_section;
ProElement pro_e_std_sec_method;
ProElement pro_e_std_sec_setup_plane;
ProElement pro_e_std_sec_plane;
ProElement pro_e_std_sec_plane_view_dir;
ProElement pro_e_std_sec_plane_orient_dir;
ProElement pro_e_std_sec_plane_orient_ref;

ProSelection *sketch_refs;
ProError status;
ProValueData value_data;
ProSelection * p_select;
int n_select;

ProStringToWstring ( message_file, "utilities.txt" );

log_file = PTApplsUnicodeFopen ( "ug_sketched_curve.log", "w" );

/*-----*\
   Populating root element PRO_E_FEATURE_TREE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TREE *** " );

```

```

status = ProElementAlloc ( PRO_E_FEATURE_TREE, &pro_e_feature_tree );
C_LOG( " ProElementAlloc " );

/*-----*\
    Populating element PRO_E_FEATURE_FORM
\*-----*/

C_PRINT( " *** Processing Element PRO_E_FEATURE_FORM *** " );
status = ProElementAlloc ( PRO_E_FEATURE_FORM, &pro_e_feature_form );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXTRUDE;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_feature_form, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_feature_form );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_EXT_SURF_CUT_SOLID_TYPE
\*-----*/

C_PRINT( " *** Processing Element PRO_E_EXT_SURF_CUT_SOLID_TYPE *** " );
status = ProElementAlloc ( PRO_E_EXT_SURF_CUT_SOLID_TYPE,
&pro_e_ext_surf_cut_solid_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_FEAT_TYPE_SOLID;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_ext_surf_cut_solid_type, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_ext_surf_cut_solid_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_REMOVE_MATERIAL
\*-----*/

C_PRINT( " *** Processing Element PRO_E_REMOVE_MATERIAL *** " );
status = ProElementAlloc ( PRO_E_REMOVE_MATERIAL, &pro_e_remove_material
);
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;

```

```

value_data.v.i = PRO_EXT_MATERIAL_ADD;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_remove_material, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_remove_material );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_FEAT_FORM_IS_THIN
\*-----*/

C_PRINT( " *** Processing Element PRO_E_FEAT_FORM_IS_THIN *** " );
status = ProElementAlloc ( PRO_E_FEAT_FORM_IS_THIN,
&pro_e_feat_form_is_thin );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_FEAT_FORM_NO_THIN;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_feat_form_is_thin, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_feat_form_is_thin );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating compound element PRO_E_STD_EXT_DEPTH
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_EXT_DEPTH *** " );
status = ProElementAlloc ( PRO_E_STD_EXT_DEPTH, &pro_e_std_ext_depth );
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_std_ext_depth );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_EXT_DEPTH
        -> PRO_E_EXT_DEPTH_FROM
\*-----*/
C_PRINT( " *** Processing Element PRO_E_EXT_DEPTH_FROM *** " );
status = ProElementAlloc ( PRO_E_EXT_DEPTH_FROM, &pro_e_ext_depth_from
);
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_std_ext_depth, NULL,
pro_e_ext_depth_from );

```

```

C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_EXT_DEPTH
-> PRO_E_EXT_DEPTH_FROM
    -> PRO_E_EXT_DEPTH_FROM_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_EXT_DEPTH_FROM_TYPE *** " );
status = ProElementAlloc ( PRO_E_EXT_DEPTH_FROM_TYPE,
&pro_e_ext_depth_from_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_DEPTH_FROM_BLIND;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_ext_depth_from_type, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_ext_depth_from, NULL,
pro_e_ext_depth_from_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_EXT_DEPTH
-> PRO_E_EXT_DEPTH_FROM
    -> PRO_E_EXT_DEPTH_FROM_VALUE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_EXT_DEPTH_FROM_VALUE *** " );
status = ProElementAlloc ( PRO_E_EXT_DEPTH_FROM_VALUE,
&pro_e_ext_depth_from_value );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 120.000000;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_ext_depth_from_value, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_ext_depth_from, NULL,
pro_e_ext_depth_from_value );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_EXT_DEPTH
-> PRO_E_EXT_DEPTH_TO
\*-----*/
C_PRINT( " *** Processing Element PRO_E_EXT_DEPTH_TO *** " );
status = ProElementAlloc ( PRO_E_EXT_DEPTH_TO, &pro_e_ext_depth_to );
C_LOG( " ProElementAlloc" );

```

```

    status = ProElemtreeElementAdd ( pro_e_std_ext_depth, NULL,
    pro_e_ext_depth_to );
    C_LOG( " ProElemtreeElementAdd" );

    /*-----*\
        Populating element PRO_E_STD_EXT_DEPTH
    -> PRO_E_EXT_DEPTH_TO
        -> PRO_E_EXT_DEPTH_TO_TYPE
    \*-----*/
    C_PRINT( " *** Processing Element PRO_E_EXT_DEPTH_TO_TYPE *** " );
    status = ProElementAlloc ( PRO_E_EXT_DEPTH_TO_TYPE,
    &pro_e_ext_depth_to_type );
    C_LOG( " ProElementAlloc " );
    value_data.type = PRO_VALUE_TYPE_INT;
    value_data.v.i = PRO_EXT_DEPTH_SYMMETRIC;
    status = ProValueAlloc ( &value );
    C_LOG( " ProValueAlloc" );
    status = ProValueDataSet ( value, &value_data );
    C_LOG( " ProValueDataSet" );
    status = ProElementValueSet ( pro_e_ext_depth_to_type, value );
    C_LOG( " ProElementValueSet" );
    status = ProElemtreeElementAdd ( pro_e_ext_depth_to, NULL,
    pro_e_ext_depth_to_type );
    C_LOG( " ProElemtreeElementAdd" );

    /*-----*\
        Populating element PRO_E_STD_SECTION
    \*-----*/

    C_PRINT( " *** Processing Element PRO_E_STD_SECTION *** " );
    status = ProElementAlloc ( PRO_E_STD_SECTION, &pro_e_std_section );
    C_LOG( " ProElementAlloc" );
    status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
    pro_e_std_section );
    C_LOG( " ProElemtreeElementAdd" );

    /*-----*\
        Populating element PRO_E_STD_SECTION
    -> PRO_E_STD_SEC_SETUP_PLANE
    \*-----*/
    C_PRINT( " *** Processing Element PRO_E_STD_SEC_SETUP_PLANE *** " );
    status = ProElementAlloc ( PRO_E_STD_SEC_SETUP_PLANE,
    &pro_e_std_sec_setup_plane );
    C_LOG( " ProElementAlloc" );
    status = ProElemtreeElementAdd ( pro_e_std_section, NULL,
    pro_e_std_sec_setup_plane );
    C_LOG( " ProElemtreeElementAdd" );

    sketch_refs = ( ProSelection *) calloc ( 2, sizeof ( ProSelection ));

    /*-----*\

```

```

    Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    -> PRO_E_STD_SEC_PLANE
    /*-----*/

    C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE *** " );
    status = ProMessageDisplay ( message_file, "Select Surface for sketch
placement");
    C_LOG( " ProMessageDisplay" );
    ProTKPrintf ( "Please select datum,surface,sldface,qltface_ID_5 type of
Modelitem\n");
    status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL, NULL,
NULL, NULL, &p_select, &n_select );
    C_LOG( " ProSelect" );
    if ( n_select <= 0 ) return -1;
    else
    {
        status = ProSelectionCopy( p_select[0], &sketch_refs[0]);
        C_LOG ( " ProSelectionCopy " );
    }
    status = ProElementAlloc ( PRO_E_STD_SEC_PLANE, &pro_e_std_sec_plane );
    C_LOG( " ProElementAlloc " );
    value_data.type = PRO_VALUE_TYPE_SELECTION;
    value_data.v.r = p_select[0];
    status = ProValueAlloc ( &value );
    C_LOG( " ProValueAlloc" );
    status = ProValueDataSet ( value, &value_data );
    C_LOG( " ProValueDataSet" );
    status = ProElementValueSet ( pro_e_std_sec_plane, value );
    C_LOG( " ProElementValueSet" );
    status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane );
    C_LOG( " ProElemtreeElementAdd" );

    /*-----*\
    Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    -> PRO_E_STD_SEC_PLANE_VIEW_DIR
    /*-----*/
    C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_VIEW_DIR *** " );
    status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_VIEW_DIR,
&pro_e_std_sec_plane_view_dir );
    C_LOG( " ProElementAlloc " );
    value_data.type = PRO_VALUE_TYPE_INT;
    value_data.v.i = PRO_SEC_VIEW_DIR_SIDE_ONE; /* 1 */
    status = ProValueAlloc ( &value );
    C_LOG( " ProValueAlloc" );
    status = ProValueDataSet ( value, &value_data );
    C_LOG( " ProValueDataSet" );
    status = ProElementValueSet ( pro_e_std_sec_plane_view_dir, value );
    C_LOG( " ProElementValueSet" );

```

```

    status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
    pro_e_std_sec_plane_view_dir );
    C_LOG( " ProElemtreeElementAdd" );

    /*-----*\
        Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    -> PRO_E_STD_SEC_PLANE_ORIENT_DIR
    \*-----*/
    C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_ORIENT_DIR *** "
);
    status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_ORIENT_DIR,
    &pro_e_std_sec_plane_orient_dir );
    C_LOG( " ProElementAlloc " );
    value_data.type = PRO_VALUE_TYPE_INT;
    value_data.v.i = PRO_SEC_ORIENT_DIR_UP; /* 1 */
    status = ProValueAlloc ( &value );
    C_LOG( " ProValueAlloc " );
    status = ProValueDataSet ( value, &value_data );
    C_LOG( " ProValueDataSet " );
    status = ProElementValueSet ( pro_e_std_sec_plane_orient_dir, value );
    C_LOG( " ProElementValueSet " );
    status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
    pro_e_std_sec_plane_orient_dir );
    C_LOG( " ProElemtreeElementAdd" );

    /*-----*\
        Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    -> PRO_E_STD_SEC_PLANE_ORIENT_REF
    \*-----*/
    C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_ORIENT_REF *** "
);
    status = ProMessageDisplay ( message_file,
    "Select Surface for sketch orientation");
    C_LOG( " ProMessageDisplay " );
    ProTKPrintf ( "Select datum,surface,sldface,qltface_ID_5 type of
    Modelitem\n");
    status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL,
    NULL, NULL, NULL, &p_select, &n_select );
    C_LOG( " ProSelect " );
    if ( n_select <= 0 ) return -1;
    else
    {
        status = ProSelectionCopy( p_select[0], &sketch_refs[1]);
        C_LOG ( " ProSelectionCopy " );
    }

    status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_ORIENT_REF,
    &pro_e_std_sec_plane_orient_ref );
    C_LOG( " ProElementAlloc " );

```

```

value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_sec_plane_orient_ref, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_orient_ref );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Creating incomplete feature in the current model.
\*-----*/
status = ProMdlCurrentGet (&model);
if ( status != PRO_TK_NO_ERROR ) return ( status );
status = ProMdlToModelitem( model, &model_item );
status = ProSelectionAlloc (p_comp_path, &model_item,
    &model_sel);

opts[0] = PRO_FEAT_CR_INCOMPLETE_FEAT;
status = ProFeatureCreate (model_sel, pro_e_feature_tree, opts, 1,
    &feature, &errors);
C_LOG ( " ProFeatureCreate");

/* Using the element tree from created feature */
status = ProFeatureElemtreeCreate ( &feature, &created_elemtree );
C_LOG ( " ProFeatureElemtreeCreate");

/*-----*\
    Getting the initialized section element from the database.
\*-----*/
/* path to PRO_E_SKETCHER element */
path_items[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[0].path_item.elem_id = PRO_E_STD_SECTION;
path_items[1].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[1].path_item.elem_id = PRO_E_SKETCHER;
status = ProElemppathAlloc (&path);
status = ProElemppathDataSet (path, path_items, 2);

status = ProElemtreeElementGet ( created_elemtree, path,
&sketch_element);
C_LOG ( " ProElemtreeElementGet PRO_E_SKETCHER ");

status = ProElementValueGet ( sketch_element, &value);
C_LOG ( " ProElementValueGet PRO_E_SKETCHER ");

status = ProValueDataGet (value, &value_data);
C_LOG ( " ProValueDataGet PRO_E_SKETCHER ");
section = (ProSection)value_data.v.p;

```



```

/*-----*\
    Creating a 3-D section
\*-----*/

status = UserSectionBuild (( ProSection ) (value_data.v.p),
    sketch_refs );
C_LOG ("UserSectionBuild");

ProElempathFree (&path);

/*-----*\
    Populating element PRO_E_STD_MATRLSIDE (must be done once section is
set)
\*-----*/

C_PRINT( " *** Processing Element PRO_E_STD_MATRLSIDE *** " );
path_items[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[0].path_item.elem_id = PRO_E_STD_MATRLSIDE;

status = ProElempathAlloc (&path);
status = ProElempathDataSet (path, path_items, 1);

status = ProElemtreeElementGet ( created_elemtree, path,
&pro_e_std_matrlside);
C_LOG ( " ProElemtreeElementGet PRO_E_SKETCHER " );

status = ProElementValueGet ( pro_e_std_matrlside, &value );
C_LOG( " ProElementValueGet " );

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_MATERIAL_SIDE_ONE;
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_std_matrlside, value );
C_LOG( " ProElementValueSet " );

ProElempathFree (&path);

/*-----*\
    Populating element PRO_E_STD_DIRECTION (must be done once section is
set)
\*-----*/

C_PRINT( " *** Processing Element PRO_E_STD_DIRECTION *** " );
path_items[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[0].path_item.elem_id = PRO_E_STD_DIRECTION;

status = ProElempathAlloc (&path);
status = ProElempathDataSet (path, path_items, 1);

```

```

    status = ProElemtreeElementGet ( created_elemtree, path,
&pro_e_std_direction);
    C_LOG ( " ProElemtreeElementGet PRO_E_SKETCHER " );

    status = ProElementValueGet ( pro_e_std_direction, &value );
    C_LOG( " ProElementValueGet " );

    value_data.type = PRO_VALUE_TYPE_INT;
    value_data.v.i = PRO_EXT_CR_IN_SIDE_TWO;
    status = ProValueDataSet ( value, &value_data );
    C_LOG( " ProValueDataSet " );
    status = ProElementValueSet ( pro_e_std_direction, value );
    C_LOG( " ProElementValueSet " );

    ProElempathFree (&path);

/*-----*\
    Redefining the feature to make it complete.
\*-----*/
    opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMS;
    status = ProSelectionAsmcomppathGet (model_sel, &comp_path);
    C_LOG ( " ProSelectionAsmcomppathGet" );

    status = ProFeatureRedefine (&comp_path, &feature, created_elemtree,
        opts, 1, &errors);
    C_LOG ( " ProFeatureRedefine" );

/*-----*\
    Free up the allocated memory.
\*-----*/
    status = ProFeatureElemtreeFree (&feature,created_elemtree );
    C_LOG ( " ProFeatureElemtreeFree" );

    status = ProElementFree (&pro_e_feature_tree );
    C_LOG ( " ProElementFree" );

    free(sketch_refs);

    return (status);
}

```

Example 2: To Create an Extruded Cut with Two-sided Thru-all Depth

The following example shows how to create an extruded cut with two-sided thru-all depth.

```

/*=====*\
FILE      : UserSktExtrusionCut.c
PURPOSE   : Includes all possible element assignments for extrude features.

/*----- Pro/Toolkit Includes -----*/

```

```
#include "ProToolkit.h"
#include "ProFeature.h"
#include "ProElemId.h"
#include "ProExtrude.h"
#include "ProModFeat.h"
#include "ProStdSection.h"
#include "ProElement.h"
#include "ProElempath.h"
#include "ProFeatType.h"
#include "ProFeatForm.h"
#include "ProSelection.h"
#include "ProSection.h"

#include "ProExtrude.h"

static ProFileName message_file;

#define C_LOG(a) ProTKPrintf ("Status for %s is = %d\n", a, status ); \
ERROR_CHECK( a, "UgSktExtrusionCut.c", status );

#define C_PRINT(a) ProTKPrintf ( "%s\n", a);

/*----- Function Prototypes -----*/
ProError ProDemoExtrudeProtrConvCreate();

/*----- External Data -----*/
ProError ProDemoSectCreate();

/*----- Global Data -----*/

/*=====*\
FUNCTION : UserSktExtrusionCut()
PURPOSE  : Demonstrates the creation of the extruded cut (thru-all, two
sides)
\*=====*/
ProError UserSktExtrusionCut()
{
    ProErrorlist      errors;
    ProMdl            model;
    ProModelitem      model_item;
    ProSelection      model_sel;
    ProFeature        feature;
    ProFeatureCreateOptions opts[1];
    ProElempath       path;
    ProElempathItem   path_items[2];
    ProSection        section;
    ProAsmcomppath    comp_path;
    ProAsmcomppath    *p_comp_path = NULL;
    ProValue          value;

    ProElement sketch_element;
```

```

ProElement created_elemtree;

ProElement pro_e_feature_tree;
ProElement pro_e_feature_form;
ProElement pro_e_ext_surf_cut_solid_type;
ProElement pro_e_remove_material;
ProElement pro_e_feat_form_is_thin;
ProElement pro_e_std_matrlside;
ProElement pro_e_std_direction;

ProElement pro_e_std_ext_depth;
ProElement pro_e_ext_depth_from;
ProElement pro_e_ext_depth_from_type;
ProElement pro_e_ext_depth_to;
ProElement pro_e_ext_depth_to_type;

ProElement pro_e_std_section;
ProElement pro_e_std_sec_method;
ProElement pro_e_std_sec_setup_plane;
ProElement pro_e_std_sec_plane;
ProElement pro_e_std_sec_plane_view_dir;
ProElement pro_e_std_sec_plane_orient_dir;
ProElement pro_e_std_sec_plane_orient_ref;

ProSelection *sketch_refs;
ProError status;
ProValueData value_data;
ProSelection * p_select;
int n_select;
ProBoolean is_interactive = PRO_B_TRUE;

ProStringToWstring ( message_file, "utilities.txt" );

log_file = PTApplsUnicodeFopen ( "ug_sketched_curve.log", "w" );

status = ProMdlCurrentGet (&model);
if ( status != PRO_TK_NO_ERROR ) return ( status );

/*-----*\
   Populating root element PRO_E_FEATURE_TREE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TREE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TREE, &pro_e_feature_tree );
C_LOG( " ProElementAlloc " );

/*-----*\
   Populating element PRO_E_FEATURE_FORM
\*-----*/

```

```

C_PRINT( " *** Processing Element PRO_E_FEATURE_FORM *** " );
status = ProElementAlloc ( PRO_E_FEATURE_FORM, &pro_e_feature_form );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXTRUDE;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_feature_form, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_feature_form );
C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Populating element PRO_E_EXT_SURF_CUT_SOLID_TYPE
\*-----*/

C_PRINT( " *** Processing Element PRO_E_EXT_SURF_CUT_SOLID_TYPE *** " );
status = ProElementAlloc ( PRO_E_EXT_SURF_CUT_SOLID_TYPE,
&pro_e_ext_surf_cut_solid_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_FEAT_TYPE_SOLID;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_ext_surf_cut_solid_type, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_ext_surf_cut_solid_type );
C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Populating element PRO_E_REMOVE_MATERIAL
\*-----*/

C_PRINT( " *** Processing Element PRO_E_REMOVE_MATERIAL *** " );
status = ProElementAlloc ( PRO_E_REMOVE_MATERIAL, &pro_e_remove_material
);
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_MATERIAL_REMOVE;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_remove_material, value );

```

```

C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_remove_material );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_FEAT_FORM_IS_THIN
\*-----*/

C_PRINT( " *** Processing Element PRO_E_FEAT_FORM_IS_THIN *** " );
status = ProElementAlloc ( PRO_E_FEAT_FORM_IS_THIN,
&pro_e_feat_form_is_thin );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_FEAT_FORM_NO_THIN;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_feat_form_is_thin, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_feat_form_is_thin );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating compound element PRO_E_STD_EXT_DEPTH
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_EXT_DEPTH *** " );
status = ProElementAlloc ( PRO_E_STD_EXT_DEPTH, &pro_e_std_ext_depth );
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_std_ext_depth );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_EXT_DEPTH
                    -> PRO_E_EXT_DEPTH_FROM
\*-----*/
C_PRINT( " *** Processing Element PRO_E_EXT_DEPTH_FROM *** " );
status = ProElementAlloc ( PRO_E_EXT_DEPTH_FROM, &pro_e_ext_depth_from
);
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_std_ext_depth, NULL,
pro_e_ext_depth_from );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_EXT_DEPTH
                    -> PRO_E_EXT_DEPTH_FROM
                    -> PRO_E_EXT_DEPTH_FROM_TYPE
\*-----*/

```

```

/*-----*/
C_PRINT( " *** Processing Element PRO_E_EXT_DEPTH_FROM_TYPE *** " );
status = ProElementAlloc ( PRO_E_EXT_DEPTH_FROM_TYPE,
&pro_e_ext_depth_from_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_DEPTH_FROM_ALL;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_ext_depth_from_type, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_ext_depth_from, NULL,
pro_e_ext_depth_from_type );
C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Populating element PRO_E_STD_EXT_DEPTH
        -> PRO_E_EXT_DEPTH_TO
/*-----*/
C_PRINT( " *** Processing Element PRO_E_EXT_DEPTH_TO *** " );
status = ProElementAlloc ( PRO_E_EXT_DEPTH_TO, &pro_e_ext_depth_to );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_std_ext_depth, NULL,
pro_e_ext_depth_to );
C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Populating element PRO_E_STD_EXT_DEPTH
        -> PRO_E_EXT_DEPTH_TO
        -> PRO_E_EXT_DEPTH_TO_TYPE
/*-----*/
C_PRINT( " *** Processing Element PRO_E_EXT_DEPTH_TO_TYPE *** " );
status = ProElementAlloc ( PRO_E_EXT_DEPTH_TO_TYPE,
&pro_e_ext_depth_to_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_DEPTH_TO_ALL;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_ext_depth_to_type, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_ext_depth_to, NULL,
pro_e_ext_depth_to_type );
C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Populating element PRO_E_STD_SECTION

```

```

\*-----*/

C_PRINT( " *** Processing Element PRO_E_STD_SECTION *** " );
status = ProElementAlloc ( PRO_E_STD_SECTION, &pro_e_std_section );
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_std_section );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_SECTION
        -> PRO_E_STD_SEC_SETUP_PLANE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SEC_SETUP_PLANE *** " );
status = ProElementAlloc ( PRO_E_STD_SEC_SETUP_PLANE,
&pro_e_std_sec_setup_plane );
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_std_section, NULL,
pro_e_std_sec_setup_plane );
C_LOG( " ProElemtreeElementAdd" );

sketch_refs = ( ProSelection *) calloc ( 2, sizeof ( ProSelection ));

/*-----*\
    Populating element PRO_E_STD_SECTION
        -> PRO_E_STD_SEC_SETUP_PLANE
        -> PRO_E_STD_SEC_PLANE
\*-----*/

C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE *** " );
status = ProMessageDisplay ( message_file, "Select Surface for sketch
placement");
C_LOG( " ProMessageDisplay" );
ProTKPrintf ( "Please select datum,surface,sldface,qltface_ID_5 type of
Modelitem\n");
status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL, NULL,
NULL, NULL, &p_select, &n_select );
C_LOG( " ProSelect" );
if ( n_select <= 0 ) return -1;
else
{
    status = ProSelectionCopy( p_select[0], &sketch_refs[0]);
    C_LOG ( " ProSelectionCopy " );
}
status = ProElementAlloc ( PRO_E_STD_SEC_PLANE, &pro_e_std_sec_plane );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );

```



```

C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_sec_plane, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_SECTION
        -> PRO_E_STD_SEC_SETUP_PLANE
        -> PRO_E_STD_SEC_PLANE_VIEW_DIR
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_VIEW_DIR *** " );
status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_VIEW_DIR,
&pro_e_std_sec_plane_view_dir );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_SEC_VIEW_DIR_SIDE_ONE /* PRO_SEC_VIEW_DIR_SIDE_TWO
*/;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_sec_plane_view_dir, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_view_dir );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_SECTION
        -> PRO_E_STD_SEC_SETUP_PLANE
        -> PRO_E_STD_SEC_PLANE_ORIENT_DIR
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_ORIENT_DIR *** "
);
status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_ORIENT_DIR,
&pro_e_std_sec_plane_orient_dir );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_SEC_ORIENT_DIR_UP /* PRO_SEC_ORIENT_DIR_DOWN */
/* PRO_SEC_ORIENT_DIR_LEFT */ /* PRO_SEC_ORIENT_DIR_RIGHT */ ;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_sec_plane_orient_dir, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_orient_dir );
C_LOG( " ProElemtreeElementAdd" );

```

```

/*-----*\
    Populating element PRO_E_STD_SECTION
        -> PRO_E_STD_SEC_SETUP_PLANE
        -> PRO_E_STD_SEC_PLANE_ORIENT_REF
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_ORIENT_REF *** "
);
    status = ProMessageDisplay ( message_file,
"Select Surface for sketch orientation");
    C_LOG( " ProMessageDisplay " );
    ProTKPrintf ( "Select datum,surface,sldface,qltface_ID_5 type of
Modelitem\n");
    status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL,
NULL, NULL, NULL, &p_select, &n_select );
    C_LOG( " ProSelect " );
    if ( n_select <= 0 ) return -1;
    else
    {
        status = ProSelectionCopy( p_select[0], &sketch_refs[1]);
        C_LOG ( " ProSelectionCopy " );
    }

    status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_ORIENT_REF,
&pro_e_std_sec_plane_orient_ref );
    C_LOG( " ProElementAlloc " );
    value_data.type = PRO_VALUE_TYPE_SELECTION;
    value_data.v.r = p_select[0];
    status = ProValueAlloc ( &value );
    C_LOG( " ProValueAlloc " );
    status = ProValueDataSet ( value, &value_data );
    C_LOG( " ProValueDataSet " );
    status = ProElementValueSet ( pro_e_std_sec_plane_orient_ref, value );
    C_LOG( " ProElementValueSet " );
    status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_orient_ref );
    C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Creating incomplete feature in the current model.
\*-----*/
    status = ProMdlToModelitem( model, &model_item );
    status = ProSelectionAlloc (p_comp_path, &model_item,
    &model_sel);

    opts[0] = PRO_FEAT_CR_INCOMPLETE_FEAT;
    status = ProFeatureCreate (model_sel, pro_e_feature_tree, opts, 1,
    &feature, &errors);
    C_LOG ( " ProFeatureCreate");

/* Using the element tree from created feature */

```

```

status = ProFeatureElemtreeCreate ( &feature, &created_elemtree );
C_LOG ( " ProFeatureElemtreeCreate" );

/*-----*\
    Getting the initialized section element from the database.
\*-----*/
/* path to PRO_E_SKETCHER element */
path_items[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[0].path_item.elem_id = PRO_E_STD_SECTION;
path_items[1].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[1].path_item.elem_id = PRO_E_SKETCHER;
status = ProElemppathAlloc (&path);
status = ProElemppathDataSet (path, path_items, 2);

status = ProElemtreeElementGet ( created_elemtree, path,
&sketch_element);
C_LOG ( " ProElemtreeElementGet PRO_E_SKETCHER " );

status = ProElementValueGet ( sketch_element, &value);
C_LOG ( " ProElementValueGet PRO_E_SKETCHER " );

status = ProValueDataGet (value, &value_data);
C_LOG ( " ProValueDataGet PRO_E_SKETCHER " );
section = (ProSection)value_data.v.p;

/*-----*\
    Creating a 3-D section
\*-----*/

status = UserSectionBuild ( ( ProSection )(value_data.v.p),
    sketch_refs );
C_LOG ("UserSectionBuild");

ProElemppathFree (&path);

/*-----*\
    Populating element PRO_E_STD_MATRLSIDE (must be done once section is
set)
\*-----*/

C_PRINT( " *** Processing Element PRO_E_STD_MATRLSIDE *** " );
path_items[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[0].path_item.elem_id = PRO_E_STD_MATRLSIDE;

status = ProElemppathAlloc (&path);
status = ProElemppathDataSet (path, path_items, 1);

status = ProElemtreeElementGet ( created_elemtree, path,
&pro_e_std_matrlside);
C_LOG ( " ProElemtreeElementGet PRO_E_SKETCHER " );

```

```

status = ProElementValueGet ( pro_e_std_matrlside, &value );
C_LOG( " ProElementValueGet" );

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_MATERIAL_SIDE_TWO;
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_matrlside, value );
C_LOG( " ProElementValueSet" );

ProElempathFree (&path);

/*-----*\
    Redefining the feature to make it complete.
\*-----*/
opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMS;
status = ProSelectionAsmcomppathGet (model_sel, &comp_path);
C_LOG ( " ProSelectionAsmcomppathGet");

status = ProFeatureRedefine (&comp_path, &feature, created_elemtree,
    opts, 1, &errors);
C_LOG ( " ProFeatureRedefine");

/*-----*\
    Free up the allocated memory.
\*-----*/
status = ProFeatureElemtreeFree (&feature, created_elemtree );
C_LOG ( " ProFeatureElemtreeFree");

status = ProElementFree (&pro_e_feature_tree );
C_LOG ( " ProElementFree");

free(sketch_refs);

return (status);
}

```

Example 3: To Create an Extruded Thin Cut

The following example shows how to create an extruded thin cut.
Its depth is two-sided, up to a selected reference.

```

/*=====*\
FILE      : UserSktExtrusionThin.c
PURPOSE   : Includes all possible element assignments for extrude features.
\*=====*/

/*----- Pro/Toolkit Includes -----*/
#include "ProToolkit.h"
#include "ProFeature.h"
#include "ProElemId.h"

```

```
#include "ProExtrude.h"
#include "ProModFeat.h"
#include "ProStdSection.h"
#include "ProElement.h"
#include "ProElempath.h"
#include "ProFeatType.h"
#include "ProFeatForm.h"
#include "ProSelection.h"
#include "ProSection.h"

#include "ProExtrude.h"

static ProFileName message_file;

#define C_LOG(a) ProTKPrintf ("Status for %s is = %d\n", a, status ); \
ERROR_CHECK( a, "UgSktExtrusionCreate.c", status );

#define C_PRINT(a) ProTKPrintf ( "%s\n", a);

/*=====*\
FUNCTION : UserSktExtrusionThin()
PURPOSE  : Demonstrates the creation of the extruded thin cut.
\*=====*/
ProError UserSktExtrusionThin()
{
    ProErrorlist      errors;
    ProMdl            model;
    ProModelitem      model_item;
    ProSelection       model_sel;
    ProFeature         feature;
    ProFeatureCreateOptions opts[1];
    ProElempath        path;
    ProElempathItem    path_items[2];
    ProSection         section;
    ProAsmcomppath     comp_path;
    ProAsmcomppath     *p_comp_path = NULL;
    ProValue           value;
    double             width;
    double             height;
    double             bite_radius;
    double             bite_height;
    char               name[PRO_NAME_SIZE];
    ProBoolean         alloc;

    ProElement sketch_element;
    ProElement created_elemtree;

    ProElement pro_e_feature_tree;
    ProElement pro_e_feature_type;
    ProElement pro_e_feature_form;
    ProElement pro_e_ext_surf_cut_solid_type;
```

```

ProElement pro_e_remove_material;
ProElement pro_e_feat_form_is_thin;
ProElement pro_e_thickness;
ProElement pro_e_std_matrlside;
ProElement pro_e_std_direction;


ProElement pro_e_std_ext_depth;
ProElement pro_e_ext_depth_from;
ProElement pro_e_ext_depth_from_ref;
int from_ref_id, from_ref_type;
ProModelitem from_ref_item;
ProElement pro_e_ext_depth_from_type;
ProElement pro_e_ext_depth_to;
ProElement pro_e_ext_depth_to_type;
ProElement pro_e_ext_depth_to_ref;
int to_ref_id, to_ref_type;
ProModelitem to_ref_item;


ProElement pro_e_std_section;
ProElement pro_e_std_sec_method;
ProElement pro_e_std_sec_setup_plane;
ProElement pro_e_std_sec_plane;
ProElement pro_e_std_sec_plane_view_dir;
ProElement pro_e_std_sec_plane_orient_dir;
ProElement pro_e_std_sec_plane_orient_ref;


ProSelection *sketch_refs;
ProError status;
ProValueData value_data;
ProSelection * p_select;
int n_select;


ProStringToWstring ( message_file, "utilities.txt" );


log_file = PTApplsUnicodeFopen ( "ug_sketched_curve.log", "w" );


status = ProMdlCurrentGet (&model);
if ( status != PRO_TK_NO_ERROR ) return ( status );


/*-----*\
   Populating root element PRO_E_FEATURE_TREE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TREE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TREE, &pro_e_feature_tree );
C_LOG( " ProElementAlloc " );


/*-----*\
   Populating element PRO_E_FEATURE_FORM
\*-----*/

```

```

C_PRINT( " *** Processing Element PRO_E_FEATURE_FORM *** " );
status = ProElementAlloc ( PRO_E_FEATURE_FORM, &pro_e_feature_form );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXTRUDE;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_feature_form, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_feature_form );
C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Populating element PRO_E_EXT_SURF_CUT_SOLID_TYPE
\*-----*/

C_PRINT( " *** Processing Element PRO_E_EXT_SURF_CUT_SOLID_TYPE *** " );
status = ProElementAlloc ( PRO_E_EXT_SURF_CUT_SOLID_TYPE,
&pro_e_ext_surf_cut_solid_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_FEAT_TYPE_SOLID;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_ext_surf_cut_solid_type, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_ext_surf_cut_solid_type );
C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Populating element PRO_E_REMOVE_MATERIAL
\*-----*/

C_PRINT( " *** Processing Element PRO_E_REMOVE_MATERIAL *** " );
status = ProElementAlloc ( PRO_E_REMOVE_MATERIAL, &pro_e_remove_material
);
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_MATERIAL_REMOVE;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );

```

```

    status = ProElementValueSet ( pro_e_remove_material, value );
    C_LOG( " ProElementValueSet" );
    status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_remove_material );
    C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_FEAT_FORM_IS_THIN
\*-----*/

    C_PRINT( " *** Processing Element PRO_E_FEAT_FORM_IS_THIN *** " );
    status = ProElementAlloc ( PRO_E_FEAT_FORM_IS_THIN,
&pro_e_feat_form_is_thin );
    C_LOG( " ProElementAlloc " );
    value_data.type = PRO_VALUE_TYPE_INT;
    value_data.v.i = PRO_EXT_FEAT_FORM_THIN;
    status = ProValueAlloc ( &value );
    C_LOG( " ProValueAlloc" );
    status = ProValueDataSet ( value, &value_data );
    C_LOG( " ProValueDataSet" );
    status = ProElementValueSet ( pro_e_feat_form_is_thin, value );
    C_LOG( " ProElementValueSet" );
    status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_feat_form_is_thin );
    C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_THICKNESS (thin only)
\*-----*/

    C_PRINT( " *** Processing Element PRO_E_THICKNESS *** " );
    status = ProElementAlloc ( PRO_E_THICKNESS, &pro_e_thickness );
    C_LOG( " ProElementAlloc " );
    value_data.type = PRO_VALUE_TYPE_DOUBLE;
    value_data.v.d = 0.50 ;
    status = ProValueAlloc ( &value );
    C_LOG( " ProValueAlloc" );
    status = ProValueDataSet ( value, &value_data );
    C_LOG( " ProValueDataSet" );
    status = ProElementValueSet ( pro_e_thickness, value );
    C_LOG( " ProElementValueSet" );
    status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_thickness );
    C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating compound element PRO_E_STD_EXT_DEPTH
\*-----*/

    C_PRINT( " *** Processing Element PRO_E_STD_EXT_DEPTH *** " );
    status = ProElementAlloc ( PRO_E_STD_EXT_DEPTH, &pro_e_std_ext_depth );
    C_LOG( " ProElementAlloc" );

```



```

    status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
    pro_e_std_ext_depth );
    C_LOG( " ProElemtreeElementAdd" );

    /*-----*\
        Populating element PRO_E_STD_EXT_DEPTH
-> PRO_E_EXT_DEPTH_FROM
    \*-----*/
    C_PRINT( " *** Processing Element PRO_E_EXT_DEPTH_FROM *** " );
    status = ProElementAlloc ( PRO_E_EXT_DEPTH_FROM, &pro_e_ext_depth_from
);
    C_LOG( " ProElementAlloc" );
    status = ProElemtreeElementAdd ( pro_e_std_ext_depth, NULL,
    pro_e_ext_depth_from );
    C_LOG( " ProElemtreeElementAdd" );

    /*-----*\
        Populating element PRO_E_STD_EXT_DEPTH
-> PRO_E_EXT_DEPTH_FROM
    -> PRO_E_EXT_DEPTH_FROM_TYPE
    \*-----*/
    C_PRINT( " *** Processing Element PRO_E_EXT_DEPTH_FROM_TYPE *** " );
    status = ProElementAlloc ( PRO_E_EXT_DEPTH_FROM_TYPE,
&pro_e_ext_depth_from_type );
    C_LOG( " ProElementAlloc " );
    value_data.type = PRO_VALUE_TYPE_INT;
    value_data.v.i = PRO_EXT_DEPTH_FROM_REF;
    status = ProValueAlloc ( &value );
    C_LOG( " ProValueAlloc" );
    status = ProValueDataSet ( value, &value_data );
    C_LOG( " ProValueDataSet" );
    status = ProElementValueSet ( pro_e_ext_depth_from_type, value );
    C_LOG( " ProElementValueSet" );
    status = ProElemtreeElementAdd ( pro_e_ext_depth_from, NULL,
    pro_e_ext_depth_from_type );
    C_LOG( " ProElemtreeElementAdd" );

    /*-----*\
        Populating element PRO_E_STD_EXT_DEPTH
-> PRO_E_EXT_DEPTH_FROM
    -> PRO_E_EXT_DEPTH_FROM_REF
    \*-----*/
    C_PRINT( " *** Processing Element PRO_E_EXT_DEPTH_FROM_REF *** " );
    status = ProElementAlloc ( PRO_E_EXT_DEPTH_FROM_REF,
&pro_e_ext_depth_from_ref );
    C_LOG( " ProElementAlloc " );
    value_data.type = PRO_VALUE_TYPE_SELECTION;

    /* PRO_SURFACE, PRO_AXIS, PRO_EDGE, PRO_CURVE,
        PRO_POINT, PRO_EDGE_START, PRO_EDGE_END,
        PRO_CRV_START, PRO_CRV_END */

```

```

UserUtilItemSelect
("surface,sldface,qltface,datum,axis,edge,curve,edge_end,curve_end",
"USER Select reference.", &from_ref_id, ( ProType * )&from_ref_type);

status = ProModelitemInit ( model, from_ref_id, from_ref_type,
&from_ref_item);
C_LOG ( " ProModelitemInit" );

status = ProSelectionAlloc (NULL, &from_ref_item, &value_data.v.r);
C_LOG ( " ProSelectionAlloc" );

status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_ext_depth_from_ref, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_ext_depth_from, NULL,
pro_e_ext_depth_from_ref );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
Populating element PRO_E_STD_EXT_DEPTH
-> PRO_E_EXT_DEPTH_TO
\*-----*/
C_PRINT( " *** Processing Element PRO_E_EXT_DEPTH_TO *** " );
status = ProElementAlloc ( PRO_E_EXT_DEPTH_TO, &pro_e_ext_depth_to );
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_std_ext_depth, NULL,
pro_e_ext_depth_to );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
Populating element PRO_E_STD_EXT_DEPTH
-> PRO_E_EXT_DEPTH_TO
-> PRO_E_EXT_DEPTH_TO_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_EXT_DEPTH_TO_TYPE *** " );
status = ProElementAlloc ( PRO_E_EXT_DEPTH_TO_TYPE,
&pro_e_ext_depth_to_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_DEPTH_TO_REF;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_ext_depth_to_type, value );
C_LOG( " ProElementValueSet" );

```

```

    status = ProElemtreeElementAdd ( pro_e_ext_depth_to, NULL,
    pro_e_ext_depth_to_type );
    C_LOG( " ProElemtreeElementAdd" );

    /*-----*\
        Populating element PRO_E_STD_EXT_DEPTH
-> PRO_E_EXT_DEPTH_TO
    -> PRO_E_EXT_DEPTH_TO_REF
    \*-----*/
    C_PRINT( " *** Processing Element PRO_E_EXT_DEPTH_TO_REF *** " );
    status = ProElementAlloc ( PRO_E_EXT_DEPTH_TO_REF,
    &pro_e_ext_depth_to_ref );
    C_LOG( " ProElementAlloc " );
    value_data.type = PRO_VALUE_TYPE_SELECTION;

    /* PRO_SURFACE, PRO_AXIS, PRO_EDGE, PRO_CURVE,
        PRO_POINT, PRO_EDGE_START, PRO_EDGE_END,
        PRO_CRV_START, PRO_CRV_END */

    UserUtilItemSelect ( "surf,datum,axis,edge,curve,edge_end,curve_end",
    "USER Select reference.", &to_ref_id, ( ProType *) &to_ref_type);

    status = ProModelitemInit ( model, to_ref_id, to_ref_type,
    &to_ref_item);
    C_LOG ( " ProModelitemInit" );

    status = ProSelectionAlloc ( NULL, &to_ref_item, &value_data.v.r);
    C_LOG ( " ProSelectionAlloc" );

    status = ProValueAlloc ( &value );
    C_LOG( " ProValueAlloc" );
    status = ProValueDataSet ( value, &value_data );
    C_LOG( " ProValueDataSet" );
    status = ProElementValueSet ( pro_e_ext_depth_to_ref, value );
    C_LOG( " ProElementValueSet" );
    status = ProElemtreeElementAdd ( pro_e_ext_depth_to, NULL,
    pro_e_ext_depth_to_ref );
    C_LOG( " ProElemtreeElementAdd" );

    /*-----*\
        Populating element PRO_E_STD_SECTION
    \*-----*/

    C_PRINT( " *** Processing Element PRO_E_STD_SECTION *** " );
    status = ProElementAlloc ( PRO_E_STD_SECTION, &pro_e_std_section );
    C_LOG( " ProElementAlloc" );
    status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
    pro_e_std_section );
    C_LOG( " ProElemtreeElementAdd" );

    /*-----*\

```

```

    Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    /*-----*/
    C_PRINT( " *** Processing Element PRO_E_STD_SEC_SETUP_PLANE *** " );
    status = ProElementAlloc ( PRO_E_STD_SEC_SETUP_PLANE,
&pro_e_std_sec_setup_plane );
    C_LOG( " ProElementAlloc" );
    status = ProElemtreeElementAdd ( pro_e_std_section, NULL,
pro_e_std_sec_setup_plane );
    C_LOG( " ProElemtreeElementAdd" );

    sketch_refs = ( ProSelection *) calloc ( 2, sizeof ( ProSelection ));

    /*-----*\
    Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
-> PRO_E_STD_SEC_PLANE
    /*-----*/

    C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE *** " );
    status = ProMessageDisplay ( message_file, "Select Surface for sketch
placement");
    C_LOG( " ProMessageDisplay" );
    ProTKPrintf ( "Please select datum,surface,sldface,qltface_ID_5 type of
Modelitem\n");
    status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL, NULL,
NULL, NULL, &p_select, &n_select );
    C_LOG( " ProSelect" );
    if ( n_select <= 0 ) return -1;
    else
    {
        status = ProSelectionCopy( p_select[0], &sketch_refs[0]);
        C_LOG ( " ProSelectionCopy ");
    }
    status = ProElementAlloc ( PRO_E_STD_SEC_PLANE, &pro_e_std_sec_plane );
    C_LOG( " ProElementAlloc " );
    value_data.type = PRO_VALUE_TYPE_SELECTION;
    value_data.v.r = p_select[0];
    status = ProValueAlloc ( &value );
    C_LOG( " ProValueAlloc" );
    status = ProValueDataSet ( value, &value_data );
    C_LOG( " ProValueDataSet" );
    status = ProElementValueSet ( pro_e_std_sec_plane, value );
    C_LOG( " ProElementValueSet" );
    status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane );
    C_LOG( " ProElemtreeElementAdd" );

    /*-----*\
    Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE

```

```

    -> PRO_E_STD_SEC_PLANE_VIEW_DIR
    \*-----*/
    C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_VIEW_DIR *** " );
    status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_VIEW_DIR,
&pro_e_std_sec_plane_view_dir );
    C_LOG( " ProElementAlloc " );
    value_data.type = PRO_VALUE_TYPE_INT;
    value_data.v.i = PRO_SEC_VIEW_DIR_SIDE_ONE /* PRO_SEC_VIEW_DIR_SIDE_TWO
*/;
    status = ProValueAlloc ( &value );
    C_LOG( " ProValueAlloc " );
    status = ProValueDataSet ( value, &value_data );
    C_LOG( " ProValueDataSet " );
    status = ProElementValueSet ( pro_e_std_sec_plane_view_dir, value );
    C_LOG( " ProElementValueSet " );
    status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_view_dir );
    C_LOG( " ProElemtreeElementAdd " );

    /*-----*\
        Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    -> PRO_E_STD_SEC_PLANE_ORIENT_DIR
    \*-----*/
    C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_ORIENT_DIR *** "
);
    status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_ORIENT_DIR,
&pro_e_std_sec_plane_orient_dir );
    C_LOG( " ProElementAlloc " );
    value_data.type = PRO_VALUE_TYPE_INT;
    value_data.v.i = PRO_SEC_ORIENT_DIR_UP /* PRO_SEC_ORIENT_DIR_DOWN */
/* PRO_SEC_ORIENT_DIR_LEFT *//* PRO_SEC_ORIENT_DIR_RIGHT */ ;
    status = ProValueAlloc ( &value );
    C_LOG( " ProValueAlloc " );
    status = ProValueDataSet ( value, &value_data );
    C_LOG( " ProValueDataSet " );
    status = ProElementValueSet ( pro_e_std_sec_plane_orient_dir, value );
    C_LOG( " ProElementValueSet " );
    status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_orient_dir );
    C_LOG( " ProElemtreeElementAdd " );

    /*-----*\
        Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    -> PRO_E_STD_SEC_PLANE_ORIENT_REF
    \*-----*/
    C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_ORIENT_REF *** "
);
    status = ProMessageDisplay ( message_file,
"Select Surface for sketch orientation");

```

```

C_LOG( " ProMessageDisplay" );
ProTKPrintf ( "Select datum,surface,sldface,qltface_ID_5 type of
Modelitem\n");
status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL,
NULL, NULL, NULL, &p_select, &n_select );
C_LOG( " ProSelect" );
if ( n_select <= 0 ) return -1;
else
{
    status = ProSelectionCopy( p_select[0], &sketch_refs[1]);
    C_LOG ( " ProSelectionCopy ");
}

status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_ORIENT_REF,
&pro_e_std_sec_plane_orient_ref );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_sec_plane_orient_ref, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_orient_ref );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Creating incomplete feature in the current model.
\*-----*/
status = ProMdlToModelitem( model, &model_item );
status = ProSelectionAlloc (p_comp_path, &model_item,
&model_sel);

opts[0] = PRO_FEAT_CR_INCOMPLETE_FEAT;
status = ProFeatureCreate (model_sel, pro_e_feature_tree, opts, 1,
&feature, &errors);
C_LOG ( " ProFeatureCreate");

/* Using the element tree from created feature */
status = ProFeatureElemtreeCreate ( &feature, &created_elemtree );
C_LOG ( " ProFeatureElemtreeCreate");

/*-----*\
    Getting the initialized section element from the database.
\*-----*/
/* path to PRO_E_SKETCHER element */
path_items[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[0].path_item.elem_id = PRO_E_STD_SECTION;
path_items[1].type = PRO_ELEM_PATH_ITEM_TYPE_ID;

```

```

path_items[1].path_item.elem_id = PRO_E_SKETCHER;
status = ProElempathAlloc (&path);
status = ProElempathDataSet (path, path_items, 2);

status = ProElemtreeElementGet ( created_elemtree, path,
&sketch_element);
C_LOG (" ProElemtreeElementGet PRO_E_SKETCHER ");

status = ProElementValueGet ( sketch_element, &value);
C_LOG (" ProElementValueGet PRO_E_SKETCHER ");

status = ProValueDataSet (value, &value_data);
C_LOG (" ProValueDataSet PRO_E_SKETCHER ");
section = (ProSection)value_data.v.p;

/*-----*\
    Creating a 3-D section
\*-----*/

status = UserSectionBuild (( ProSection ) (value_data.v.p),
    sketch_refs );
C_LOG ("UserSectionBuild");

ProElempathFree (&path);

/*-----*\
    Populating element PRO_E_STD_MATRLSIDE (must be done once section is
set)
\*-----*/

C_PRINT( " *** Processing Element PRO_E_STD_MATRLSIDE *** " );
path_items[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[0].path_item.elem_id = PRO_E_STD_MATRLSIDE;

status = ProElempathAlloc (&path);
status = ProElempathDataSet (path, path_items, 1);

status = ProElemtreeElementGet ( created_elemtree, path,
&pro_e_std_matrlside);
C_LOG (" ProElemtreeElementGet PRO_E_SKETCHER ");

status = ProElementValueGet ( pro_e_std_matrlside, &value );
C_LOG( " ProElementValueGet" );

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_MATERIAL_BOTH_SIDES;
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_matrlside, value );
C_LOG( " ProElementValueSet" );

```

```

ProElempathFree (&path);

/*-----*\
    Populating element PRO_E_STD_DIRECTION (must be done once section is
set)
/*-----*/

C_PRINT( " *** Processing Element PRO_E_STD_DIRECTION *** " );
path_items[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[0].path_item.elem_id = PRO_E_STD_DIRECTION;

status = ProElempathAlloc (&path);
status = ProElempathDataSet (path, path_items, 1);

status = ProElemtreeElementGet ( created_elemtree, path,
&pro_e_std_direction);
C_LOG ( " ProElemtreeElementGet PRO_E_SKETCHER " );

status = ProElementValueGet ( pro_e_std_direction, &value );
C_LOG( " ProElementValueGet" );

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_CR_IN_SIDE_ONE;
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_direction, value );
C_LOG( " ProElementValueSet" );

ProElempathFree (&path);

/*-----*\
    Redefining the feature to make it complete.
/*-----*/
opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMS;
status = ProSelectionAsmcomppathGet (model_sel, &comp_path);
C_LOG ( " ProSelectionAsmcomppathGet" );

status = ProFeatureRedefine (&comp_path, &feature, created_elemtree,
    opts, 1, &errors);
C_LOG ( " ProFeatureRedefine" );

/*-----*\
    Free up the allocated memory.
/*-----*/
status = ProFeatureElemtreeFree (&feature, created_elemtree );
C_LOG ( " ProFeatureElemtreeFree" );

status = ProElementFree (&pro_e_feature_tree );
C_LOG ( " ProElementFree" );

free(sketch_refs);

```



```

    return (status);
}

```

Example 4: To Create an Extruded Datum Surface Feature

The following example shows how to create an extruded datum surface feature. Its depth is one-side blind.

```

/*=====*\
FILE      : UserSktExtrusionSurfaceCapped.c
PURPOSE   : Creates an extruded datum surface.
\*=====*/
/*----- Pro/Toolkit Includes -----*/
#include "ProToolkit.h"
#include "ProFeature.h"
#include "ProElemId.h"
#include "ProExtrude.h"
#include "ProModFeat.h"
#include "ProStdSection.h"
#include "ProElement.h"
#include "ProElempath.h"
#include "ProFeatType.h"
#include "ProFeatForm.h"
#include "ProSelection.h"
#include "ProSection.h"

#include "ProExtrude.h"

static ProFileName message_file;

#define C_LOG(a) ProTKPrintf ("Status for %s is = %d\n", a, status ); \
ERROR_CHECK( a, "UgSktExtrusionCreate.c", status );

#define C_PRINT(a) ProTKPrintf ( "%s\n", a);

/*=====*\
FUNCTION  : UserSktExtrusionSurfaceCapped()
PURPOSE   : Demonstrates the creation of the extruded datum surface.
\*=====*/
ProError UserSktExtrusionSurfaceCapped()
{
    ProErrorlist      errors;
    ProMdl             model;
    ProModelitem       model_item;
    ProSelection       model_sel;
    ProFeature         feature;
    ProFeatureCreateOptions opts[1];
    ProElempath        path;
    ProElempathItem    path_items[2];
    ProSection         section;

```

```

ProAsmcomppath      comp_path;
ProAsmcomppath      *p_comp_path = NULL;
ProValue            value;

ProElement sketch_element;
ProElement created_elemtree;

ProElement pro_e_feature_tree;
ProElement pro_e_feature_form;
ProElement pro_e_ext_surf_cut_solid_type;
ProElement pro_e_remove_material;
ProElement pro_e_feat_form_is_thin;
ProElement pro_e_srf_end_attributes;
ProElement pro_e_std_direction;

ProElement pro_e_std_ext_depth;
ProElement pro_e_ext_depth_from;
ProElement pro_e_ext_depth_from_type;
ProElement pro_e_ext_depth_to;
ProElement pro_e_ext_depth_to_type;
ProElement pro_e_ext_depth_to_value;

ProElement pro_e_std_section;
ProElement pro_e_std_sec_method;
ProElement pro_e_std_sec_setup_plane;
ProElement pro_e_std_sec_plane;
ProElement pro_e_std_sec_plane_view_dir;
ProElement pro_e_std_sec_plane_orient_dir;
ProElement pro_e_std_sec_plane_orient_ref;

ProSelection *sketch_refs;
ProError status;
ProValueData value_data;
ProSelection * p_select;
int n_select;
ProBoolean is_interactive = PRO_B_TRUE;

ProStringToWstring ( message_file, "utilities.txt" );

log_file = PTApplsUnicodeFopen ( "ug_sketched_curve.log", "w" );

status = ProMdlCurrentGet (&model);
if ( status != PRO_TK_NO_ERROR ) return ( status );

/*-----*\
   Populating root element PRO_E_FEATURE_TREE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TREE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TREE, &pro_e_feature_tree );
C_LOG( " ProElementAlloc " );

```

```

/*-----*\
    Populating element PRO_E_FEATURE_FORM
\*-----*/

C_PRINT( " *** Processing Element PRO_E_FEATURE_FORM *** " );
status = ProElementAlloc ( PRO_E_FEATURE_FORM, &pro_e_feature_form );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXTRUDE;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_feature_form, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_feature_form );
C_LOG( " ProElemtreeElementAdd " );


/*-----*\
    Populating element PRO_E_EXT_SURF_CUT_SOLID_TYPE
\*-----*/

C_PRINT( " *** Processing Element PRO_E_EXT_SURF_CUT_SOLID_TYPE *** " );
status = ProElementAlloc ( PRO_E_EXT_SURF_CUT_SOLID_TYPE,
&pro_e_ext_surf_cut_solid_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_FEAT_TYPE_SURFACE;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_ext_surf_cut_solid_type, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_ext_surf_cut_solid_type );
C_LOG( " ProElemtreeElementAdd " );


/*-----*\
    Populating element PRO_E_REMOVE_MATERIAL
\*-----*/

C_PRINT( " *** Processing Element PRO_E_REMOVE_MATERIAL *** " );
status = ProElementAlloc ( PRO_E_REMOVE_MATERIAL, &pro_e_remove_material
);
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_MATERIAL_ADD;
status = ProValueAlloc ( &value );

```

```

C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_remove_material, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_remove_material );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_FEAT_FORM_IS_THIN
\*-----*/

C_PRINT( " *** Processing Element PRO_E_FEAT_FORM_IS_THIN *** " );
status = ProElementAlloc ( PRO_E_FEAT_FORM_IS_THIN,
&pro_e_feat_form_is_thin );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_FEAT_FORM_NO_THIN;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_feat_form_is_thin, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_feat_form_is_thin );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating compound element PRO_E_STD_EXT_DEPTH
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_EXT_DEPTH *** " );
status = ProElementAlloc ( PRO_E_STD_EXT_DEPTH, &pro_e_std_ext_depth );
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_std_ext_depth );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_EXT_DEPTH
-> PRO_E_EXT_DEPTH_FROM
\*-----*/
C_PRINT( " *** Processing Element PRO_E_EXT_DEPTH_FROM *** " );
status = ProElementAlloc ( PRO_E_EXT_DEPTH_FROM, &pro_e_ext_depth_from
);
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_std_ext_depth, NULL,
pro_e_ext_depth_from );
C_LOG( " ProElemtreeElementAdd" );

```

```

/*-----*\
    Populating element PRO_E_STD_EXT_DEPTH
-> PRO_E_EXT_DEPTH_FROM
    -> PRO_E_EXT_DEPTH_FROM_TYPE
/*-----*/
C_PRINT( " *** Processing Element PRO_E_EXT_DEPTH_FROM_TYPE *** " );
status = ProElementAlloc ( PRO_E_EXT_DEPTH_FROM_TYPE,
&pro_e_ext_depth_from_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_DEPTH_FROM_NONE;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_ext_depth_from_type, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_ext_depth_from, NULL,
pro_e_ext_depth_from_type );
C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Populating element PRO_E_STD_EXT_DEPTH
-> PRO_E_EXT_DEPTH_TO
/*-----*/
C_PRINT( " *** Processing Element PRO_E_EXT_DEPTH_TO *** " );
status = ProElementAlloc ( PRO_E_EXT_DEPTH_TO, &pro_e_ext_depth_to );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_std_ext_depth, NULL,
pro_e_ext_depth_to );
C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Populating element PRO_E_STD_EXT_DEPTH
                -> PRO_E_EXT_DEPTH_TO
                -> PRO_E_EXT_DEPTH_TO_TYPE
/*-----*/
C_PRINT( " *** Processing Element PRO_E_EXT_DEPTH_TO_TYPE *** " );
status = ProElementAlloc ( PRO_E_EXT_DEPTH_TO_TYPE,
&pro_e_ext_depth_to_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_DEPTH_TO_BLIND;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_ext_depth_to_type, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_ext_depth_to, NULL,
pro_e_ext_depth_to_type );

```

```

C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_EXT_DEPTH
-> PRO_E_EXT_DEPTH_TO
    -> PRO_E_EXT_DEPTH_TO_VALUE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_EXT_DEPTH_TO_VALUE *** " );
status = ProElementAlloc ( PRO_E_EXT_DEPTH_TO_VALUE,
&pro_e_ext_depth_to_value );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 75.0;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_ext_depth_to_value, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_ext_depth_to, NULL,
pro_e_ext_depth_to_value );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_SECTION
\*-----*/

C_PRINT( " *** Processing Element PRO_E_STD_SECTION *** " );
status = ProElementAlloc ( PRO_E_STD_SECTION, &pro_e_std_section );
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_std_section );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SEC_SETUP_PLANE *** " );
status = ProElementAlloc ( PRO_E_STD_SEC_SETUP_PLANE,
&pro_e_std_sec_setup_plane );
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_std_section, NULL,
pro_e_std_sec_setup_plane );
C_LOG( " ProElemtreeElementAdd" );

sketch_refs = ( ProSelection *) calloc ( 2, sizeof ( ProSelection ));

/*-----*\
    Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE

```

```

-> PRO_E_STD_SEC_PLANE
\*-----*/

C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE *** " );
status = ProMessageDisplay ( message_file, "Select Surface for sketch
placement");
C_LOG( " ProMessageDisplay" );
ProTKPrintf ( "Please select datum,surface,sldface,qltface_ID_5 type of
Modelitem\n");
status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL, NULL,
NULL, NULL, &p_select, &n_select );
C_LOG( " ProSelect" );
if ( n_select <= 0 ) return -1;
else
{
    status = ProSelectionCopy( p_select[0], &sketch_refs[0]);
    C_LOG( " ProSelectionCopy" );
}
status = ProElementAlloc ( PRO_E_STD_SEC_PLANE, &pro_e_std_sec_plane );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_sec_plane, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    -> PRO_E_STD_SEC_PLANE_VIEW_DIR
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_VIEW_DIR *** " );
status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_VIEW_DIR,
&pro_e_std_sec_plane_view_dir );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_SEC_VIEW_DIR_SIDE_ONE /* PRO_SEC_VIEW_DIR_SIDE_TWO
*/;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_sec_plane_view_dir, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,

```

```

pro_e_std_sec_plane_view_dir );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    -> PRO_E_STD_SEC_PLANE_ORIENT_DIR
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_ORIENT_DIR *** "
);
status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_ORIENT_DIR,
&pro_e_std_sec_plane_orient_dir );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_SEC_ORIENT_DIR_UP /* PRO_SEC_ORIENT_DIR_DOWN */
/* PRO_SEC_ORIENT_DIR_LEFT */ /* PRO_SEC_ORIENT_DIR_RIGHT */ ;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_sec_plane_orient_dir, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_orient_dir );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    -> PRO_E_STD_SEC_PLANE_ORIENT_REF
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_ORIENT_REF *** "
);
status = ProMessageDisplay ( message_file,
"Select Surface for sketch orientation");
C_LOG( " ProMessageDisplay" );
ProTKPrintf ( "Select datum,surface,sldface,qltface_ID_5 type of
Modelitem\n");
status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL,
NULL, NULL, NULL, &p_select, &n_select );
C_LOG( " ProSelect" );
if ( n_select <= 0 ) return -1;
else
{
    status = ProSelectionCopy( p_select[0], &sketch_refs[1]);
    C_LOG ( " ProSelectionCopy ");
}

status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_ORIENT_REF,
&pro_e_std_sec_plane_orient_ref );
C_LOG( " ProElementAlloc " );

```



```

value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_sec_plane_orient_ref, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_orient_ref );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Creating incomplete feature in the current model.
\*-----*/
status = ProMdlToModelitem( model, &model_item );
status = ProSelectionAlloc (p_comp_path, &model_item,
    &model_sel);

opts[0] = PRO_FEAT_CR_INCOMPLETE_FEAT;
status = ProFeatureCreate (model_sel, pro_e_feature_tree, opts, 1,
    &feature, &errors);
C_LOG ( " ProFeatureCreate");

/* Using the element tree from created feature */
status = ProFeatureElemtreeCreate ( &feature, &created_elemtree );
C_LOG ( " ProFeatureElemtreeCreate");

/*-----*\
    Getting the initialized section element from the database.
\*-----*/
/* path to PRO_E_SKETCHER element */
path_items[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[0].path_item.elem_id = PRO_E_STD_SECTION;
path_items[1].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[1].path_item.elem_id = PRO_E_SKETCHER;
status = ProElempathAlloc (&path);
status = ProElempathDataSet (path, path_items, 2);

status = ProElemtreeElementGet ( created_elemtree, path,
&sketch_element);
C_LOG ( " ProElemtreeElementGet PRO_E_SKETCHER ");

status = ProElementValueGet ( sketch_element, &value);
C_LOG ( " ProElementValueGet PRO_E_SKETCHER ");

status = ProValueDataGet (value, &value_data);
C_LOG ( " ProValueDataGet PRO_E_SKETCHER ");
section = (ProSection)value_data.v.p;

/*-----*\

```

```

    Creating a 3-D section
\*-----*/

    status = UserSectionBuild ( ( ProSection ) (value_data.v.p),
        sketch_refs );
    C_LOG ( "UserSectionBuild" );

    ProElempathFree ( &path );

/*-----*\
    Populating element PRO_E_STD_DIRECTION (must be done once section is
set)
\*-----*/

    C_PRINT( " *** Processing Element PRO_E_STD_DIRECTION *** " );
    path_items[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
    path_items[0].path_item.elem_id = PRO_E_STD_DIRECTION;

    status = ProElempathAlloc ( &path );
    status = ProElempathDataSet ( path, path_items, 1 );

    status = ProElemtreeElementGet ( created_elemtree, path,
    &pro_e_std_direction );
    C_LOG ( " ProElemtreeElementGet PRO_E_SKETCHER " );

    status = ProElementValueGet ( pro_e_std_direction, &value );
    C_LOG( " ProElementValueGet " );

    value_data.type = PRO_VALUE_TYPE_INT;
    value_data.v.i = PRO_EXT_CR_IN_SIDE_TWO /* PRO_EXT_CR_IN_SIDE_ONE */;
    status = ProValueDataSet ( value, &value_data );
    C_LOG( " ProValueDataSet " );
    status = ProElementValueSet ( pro_e_std_direction, value );
    C_LOG( " ProElementValueSet " );

    ProElempathFree ( &path );

/*-----*\
    Populating element PRO_E_SRF_END_ATTRIBUTES (surface features only -
must be done once section is set)
\*-----*/

    C_PRINT( " *** Processing Element PRO_E_SRF_END_ATTRIBUTES *** " );
    path_items[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
    path_items[0].path_item.elem_id = PRO_E_SRF_END_ATTRIBUTES;

    status = ProElempathAlloc ( &path );
    status = ProElempathDataSet ( path, path_items, 1 );

    status = ProElemtreeElementGet ( created_elemtree, path,
    &pro_e_srf_end_attributes );

```

```

C_LOG ( " ProElemtreeElementGet PRO_E_SRF_END_ATTRIBUTES ");

status = ProElementValueGet ( pro_e_srf_end_attributes, &value );
C_LOG( " ProElementValueGet" );

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_SURF_END_ATTR_CAPPED;
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_srf_end_attributes, value );
C_LOG( " ProElementValueSet" );

ProElemptreeFree (&path);

/*-----*\
    Redefining the feature to make it complete.
\*-----*/
opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMS;
status = ProSelectionAsmcomppathGet (model_sel, &comp_path);
C_LOG ( " ProSelectionAsmcomppathGet");

status = ProFeatureRedefine (&comp_path, &feature, created_elemtree,
    opts, 1, &errors);
C_LOG ( " ProFeatureRedefine");

/*-----*\
    Free up the allocated memory.
\*-----*/
status = ProFeatureElemtreeFree (&feature, created_elemtree );
C_LOG ( " ProFeatureElemtreeFree");

status = ProElementFree (&pro_e_feature_tree );
C_LOG ( " ProElementFree");

free(sketch_refs);

return (status);
}

```

Example 5: To Create a Surface Trim Extruded Feature

The following example shows how to create an extruded surface trim. Its depth is one-side blind.

```

/*=====*\
FILE      : UserSktExtrusionSurfaceTrim.c
PURPOSE   : Creates a surface trim extruded feature.
\*=====*/

/*----- Pro/Toolkit Includes -----*/
#include "ProToolkit.h"

```

```

#include "ProFeature.h"
#include "ProElemId.h"
#include "ProExtrude.h"
#include "ProModFeat.h"
#include "ProStdSection.h"
#include "ProElement.h"
#include "ProElempath.h"
#include "ProFeatType.h"
#include "ProFeatForm.h"
#include "ProSelection.h"
#include "ProSection.h"

#include "ProExtrude.h"

static ProFileName message_file;

#define C_LOG(a) ProTKPrintf ("Status for %s is = %d\n", a, status ); \
ERROR_CHECK( a, "UgSktExtrusionCreate.c", status );

#define C_PRINT(a) ProTKPrintf ( "%s\n", a);

/*=====*\
FUNCTION : UserSktExtrusionSurfaceTrim()
PURPOSE  : Demonstrates the creation of the extruded surface trim.
\*=====*/
ProError UserSktExtrusionSurfaceTrim()
{
    ProErrorlist      errors;
    ProMdl            model;
    ProModelitem      model_item;
    ProSelection       model_sel;
    ProFeature         feature;
    ProFeatureCreateOptions opts[1];
    ProElempath        path;
    ProElempathItem    path_items[2];
    ProSection         section;
    ProAsmcomppath     comp_path;
    ProAsmcomppath     *p_comp_path = NULL;
    ProValue           value;

    ProElement sketch_element;
    ProElement created_elemtree;

    ProElement pro_e_feature_tree;
    ProElement pro_e_feature_form;
    ProElement pro_e_ext_surf_cut_solid_type;
    ProElement pro_e_remove_material;
    ProElement pro_e_trim_qlt_side;
    ProElement pro_e_trim_quilt;
    int quilt_id, quilt_type;
    ProModelitem quilt_item;

```

```

ProElement pro_e_std_matrlside;
ProElement pro_e_std_direction;


ProElement pro_e_std_ext_depth;
ProElement pro_e_ext_depth_from;
ProElement pro_e_ext_depth_from_type;
ProElement pro_e_ext_depth_to;
ProElement pro_e_ext_depth_to_type;
ProElement pro_e_ext_depth_to_value;


ProElement pro_e_std_section;
ProElement pro_e_std_sec_method;
ProElement pro_e_std_sec_setup_plane;
ProElement pro_e_std_sec_plane;
ProElement pro_e_std_sec_plane_view_dir;
ProElement pro_e_std_sec_plane_orient_dir;
ProElement pro_e_std_sec_plane_orient_ref;


ProSelection *sketch_refs;
ProError status;
ProValueData value_data;
ProSelection * p_select;
int n_select;


ProStringToWstring ( message_file, "utilities.txt" );

log_file = PTApplsUnicodeFopen ( "ug_sketched_curve.log", "w" );

status = ProMdlCurrentGet (&model);
if ( status != PRO_TK_NO_ERROR ) return ( status );


/*-----*\
   Populating root element PRO_E_FEATURE_TREE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TREE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TREE, &pro_e_feature_tree );
C_LOG( " ProElementAlloc " );


/*-----*\
   Populating element PRO_E_FEATURE_FORM
\*-----*/

C_PRINT( " *** Processing Element PRO_E_FEATURE_FORM *** " );
status = ProElementAlloc ( PRO_E_FEATURE_FORM, &pro_e_feature_form );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXTRUDE;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );

```

```

C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_feature_form, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_feature_form );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_EXT_SURF_CUT_SOLID_TYPE
\*-----*/

C_PRINT( " *** Processing Element PRO_E_EXT_SURF_CUT_SOLID_TYPE *** " );
status = ProElementAlloc ( PRO_E_EXT_SURF_CUT_SOLID_TYPE,
&pro_e_ext_surf_cut_solid_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_FEAT_TYPE_SURFACE;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_ext_surf_cut_solid_type, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_ext_surf_cut_solid_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_REMOVE_MATERIAL
\*-----*/

C_PRINT( " *** Processing Element PRO_E_REMOVE_MATERIAL *** " );
status = ProElementAlloc ( PRO_E_REMOVE_MATERIAL, &pro_e_remove_material
);
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_MATERIAL_REMOVE;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_remove_material, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_remove_material );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_TRIM_QUILT (surface trim features only)

```

```

/*-----*/
C_PRINT( " *** Processing Element PRO_E_TRIM_QUILT *** " );
status = ProElementAlloc ( PRO_E_TRIM_QUILT, &pro_e_trim_quilt );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_SELECTION;

UserUtilItemSelect ("dtmqilt", "USER Select quilt.", &quilt_id, (ProType
*)&quilt_type);

status = ProModelitemInit ( model, quilt_id, quilt_type, &quilt_item);
C_LOG ( " ProModelitemInit" );

status = ProSelectionAlloc (NULL, &quilt_item, &value_data.v.r);
C_LOG ( " ProSelectionAlloc" );

status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_trim_quilt, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_trim_quilt );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating compound element PRO_E_STD_EXT_DEPTH
/*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_EXT_DEPTH *** " );
status = ProElementAlloc ( PRO_E_STD_EXT_DEPTH, &pro_e_std_ext_depth );
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_std_ext_depth );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_EXT_DEPTH
-> PRO_E_EXT_DEPTH_FROM
/*-----*/
C_PRINT( " *** Processing Element PRO_E_EXT_DEPTH_FROM *** " );
status = ProElementAlloc ( PRO_E_EXT_DEPTH_FROM, &pro_e_ext_depth_from
);
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_std_ext_depth, NULL,
pro_e_ext_depth_from );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_EXT_DEPTH
-> PRO_E_EXT_DEPTH_FROM
-> PRO_E_EXT_DEPTH_FROM_TYPE

```

```

\*-----*/
C_PRINT( " *** Processing Element PRO_E_EXT_DEPTH_FROM_TYPE *** " );
status = ProElementAlloc ( PRO_E_EXT_DEPTH_FROM_TYPE,
&pro_e_ext_depth_from_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_DEPTH_FROM_NONE;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_ext_depth_from_type, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_ext_depth_from, NULL,
pro_e_ext_depth_from_type );
C_LOG( " ProElemtreeElementAdd " );

\*-----*\
    Populating element PRO_E_STD_EXT_DEPTH
-> PRO_E_EXT_DEPTH_TO
\*-----*/
C_PRINT( " *** Processing Element PRO_E_EXT_DEPTH_TO *** " );
status = ProElementAlloc ( PRO_E_EXT_DEPTH_TO, &pro_e_ext_depth_to );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_std_ext_depth, NULL,
pro_e_ext_depth_to );
C_LOG( " ProElemtreeElementAdd " );

\*-----*\
    Populating element PRO_E_STD_EXT_DEPTH
-> PRO_E_EXT_DEPTH_TO
    -> PRO_E_EXT_DEPTH_TO_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_EXT_DEPTH_TO_TYPE *** " );
status = ProElementAlloc ( PRO_E_EXT_DEPTH_TO_TYPE,
&pro_e_ext_depth_to_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_DEPTH_TO_BLIND;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_ext_depth_to_type, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_ext_depth_to, NULL,
pro_e_ext_depth_to_type );
C_LOG( " ProElemtreeElementAdd " );

\*-----*\
    Populating element PRO_E_STD_EXT_DEPTH

```



```

-> PRO_E_EXT_DEPTH_TO
    -> PRO_E_EXT_DEPTH_TO_VALUE
    /*-----*/
    C_PRINT( " *** Processing Element PRO_E_EXT_DEPTH_TO_VALUE *** " );
    status = ProElementAlloc ( PRO_E_EXT_DEPTH_TO_VALUE,
&pro_e_ext_depth_to_value );
    C_LOG( " ProElementAlloc " );
    value_data.type = PRO_VALUE_TYPE_DOUBLE;
    value_data.v.d = 40.0;
    status = ProValueAlloc ( &value );
    C_LOG( " ProValueAlloc " );
    status = ProValueDataSet ( value, &value_data );
    C_LOG( " ProValueDataSet " );
    status = ProElementValueSet ( pro_e_ext_depth_to_value, value );
    C_LOG( " ProElementValueSet " );
    status = ProElemtreeElementAdd ( pro_e_ext_depth_to, NULL,
pro_e_ext_depth_to_value );
    C_LOG( " ProElemtreeElementAdd " );
    /*-----*\
        Populating element PRO_E_STD_SECTION
    /*-----*/

    C_PRINT( " *** Processing Element PRO_E_STD_SECTION *** " );
    status = ProElementAlloc ( PRO_E_STD_SECTION, &pro_e_std_section );
    C_LOG( " ProElementAlloc " );
    status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_std_section );
    C_LOG( " ProElemtreeElementAdd " );

    /*-----*\
        Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    /*-----*/
    C_PRINT( " *** Processing Element PRO_E_STD_SEC_SETUP_PLANE *** " );
    status = ProElementAlloc ( PRO_E_STD_SEC_SETUP_PLANE,
&pro_e_std_sec_setup_plane );
    C_LOG( " ProElementAlloc " );
    status = ProElemtreeElementAdd ( pro_e_std_section, NULL,
pro_e_std_sec_setup_plane );
    C_LOG( " ProElemtreeElementAdd " );

    sketch_refs = ( ProSelection *) calloc ( 2, sizeof ( ProSelection ) );

    /*-----*\
        Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    -> PRO_E_STD_SEC_PLANE
    /*-----*/

    C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE *** " );
    status = ProMessageDisplay ( message_file, "Select Surface for sketch

```

```

placement");
    C_LOG( " ProMessageDisplay" );
    ProTKPrintf ( "Please select datum,surface,sldface,qltface_ID_5 type of
Modelitem\n");
    status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL, NULL,
NULL, NULL, &p_select, &n_select );
    C_LOG( " ProSelect" );
    if ( n_select <= 0 ) return -1;
    else
    {
        status = ProSelectionCopy( p_select[0], &sketch_refs[0]);
        C_LOG ( " ProSelectionCopy ");
    }
    status = ProElementAlloc ( PRO_E_STD_SEC_PLANE, &pro_e_std_sec_plane );
    C_LOG( " ProElementAlloc " );
    value_data.type = PRO_VALUE_TYPE_SELECTION;
    value_data.v.r = p_select[0];
    status = ProValueAlloc ( &value );
    C_LOG( " ProValueAlloc" );
    status = ProValueDataSet ( value, &value_data );
    C_LOG( " ProValueDataSet" );
    status = ProElementValueSet ( pro_e_std_sec_plane, value );
    C_LOG( " ProElementValueSet" );
    status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane );
    C_LOG( " ProElemtreeElementAdd" );

    /*-----*\
        Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
-> PRO_E_STD_SEC_PLANE_VIEW_DIR
    \*-----*/
    C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_VIEW_DIR *** " );
    status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_VIEW_DIR,
&pro_e_std_sec_plane_view_dir );
    C_LOG( " ProElementAlloc " );
    value_data.type = PRO_VALUE_TYPE_INT;
    value_data.v.i = PRO_SEC_VIEW_DIR_SIDE_ONE /* PRO_SEC_VIEW_DIR_SIDE_TWO
*/;
    status = ProValueAlloc ( &value );
    C_LOG( " ProValueAlloc" );
    status = ProValueDataSet ( value, &value_data );
    C_LOG( " ProValueDataSet" );
    status = ProElementValueSet ( pro_e_std_sec_plane_view_dir, value );
    C_LOG( " ProElementValueSet" );
    status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_view_dir );
    C_LOG( " ProElemtreeElementAdd" );

    /*-----*\
        Populating element PRO_E_STD_SECTION

```

```

-> PRO_E_STD_SEC_SETUP_PLANE
  -> PRO_E_STD_SEC_PLANE_ORIENT_DIR
  /*-----*/
  C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_ORIENT_DIR *** "
);
  status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_ORIENT_DIR,
&pro_e_std_sec_plane_orient_dir );
  C_LOG( " ProElementAlloc " );
  value_data.type = PRO_VALUE_TYPE_INT;
  value_data.v.i = PRO_SEC_ORIENT_DIR_UP /* PRO_SEC_ORIENT_DIR_DOWN */
/* PRO_SEC_ORIENT_DIR_LEFT *//* PRO_SEC_ORIENT_DIR_RIGHT */ ;
  status = ProValueAlloc ( &value );
  C_LOG( " ProValueAlloc " );
  status = ProValueDataSet ( value, &value_data );
  C_LOG( " ProValueDataSet " );
  status = ProElementValueSet ( pro_e_std_sec_plane_orient_dir, value );
  C_LOG( " ProElementValueSet " );
  status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_orient_dir );
  C_LOG( " ProElemtreeElementAdd " );

  /*-----*\
  Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
  -> PRO_E_STD_SEC_PLANE_ORIENT_REF
  /*-----*/
  C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_ORIENT_REF *** "
);
  status = ProMessageDisplay ( message_file,
"Select Surface for sketch orientation");
  C_LOG( " ProMessageDisplay " );
  ProTKPrintf ( "Select datum,surface,sldface,qltface_ID_5 type of
Modelitem\n");
  status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL,
NULL, NULL, NULL, &p_select, &n_select );
  C_LOG( " ProSelect " );
  if ( n_select <= 0 ) return -1;
  else
  {
    status = ProSelectionCopy( p_select[0], &sketch_refs[1]);
    C_LOG ( " ProSelectionCopy " );
  }

  status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_ORIENT_REF,
&pro_e_std_sec_plane_orient_ref );
  C_LOG( " ProElementAlloc " );
  value_data.type = PRO_VALUE_TYPE_SELECTION;
  value_data.v.r = p_select[0];
  status = ProValueAlloc ( &value );
  C_LOG( " ProValueAlloc " );
  status = ProValueDataSet ( value, &value_data );

```

```

C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_sec_plane_orient_ref, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_orient_ref );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Creating incomplete feature in the current model.
\*-----*/
status = ProMdlToModelitem( model, &model_item );
status = ProSelectionAlloc (p_comp_path, &model_item,
    &model_sel);

opts[0] = PRO_FEAT_CR_INCOMPLETE_FEAT;
status = ProFeatureCreate (model_sel, pro_e_feature_tree, opts, 1,
    &feature, &errors);
C_LOG ( " ProFeatureCreate");

/* Using the element tree from created feature */
status = ProFeatureElemtreeCreate ( &feature, &created_elemtree );
C_LOG ( " ProFeatureElemtreeCreate");

/*-----*\
    Getting the initialized section element from the database.
\*-----*/
/* path to PRO_E_SKETCHER element */
path_items[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[0].path_item.elem_id = PRO_E_STD_SECTION;
path_items[1].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[1].path_item.elem_id = PRO_E_SKETCHER;
status = ProElemppathAlloc (&path);
status = ProElemppathDataSet (path, path_items, 2);

status = ProElemtreeElementGet ( created_elemtree, path,
&sketch_element);
C_LOG ( " ProElemtreeElementGet PRO_E_SKETCHER ");

status = ProElementValueGet ( sketch_element, &value);
C_LOG ( " ProElementValueGet PRO_E_SKETCHER ");

status = ProValueDataGet (value, &value_data);
C_LOG ( " ProValueDataGet PRO_E_SKETCHER ");
section = (ProSection)value_data.v.p;

/*-----*\
    Creating a 3-D section
\*-----*/

status = UserSectionBuild (( ProSection )(value_data.v.p),
    sketch_refs );

```

```

C_LOG ("UserSectionBuild");

ProElempathFree (&path);

/*-----*\
Populating element PRO_E_STD_MATRLSIDE (must be done once section is
set)
\*-----*/

C_PRINT( " *** Processing Element PRO_E_STD_MATRLSIDE *** " );
path_items[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[0].path_item.elem_id = PRO_E_STD_MATRLSIDE;

status = ProElempathAlloc (&path);
status = ProElempathDataSet (path, path_items, 1);

status = ProElemtreeElementGet ( created_elemtree, path,
&pro_e_std_matrlside);
C_LOG (" ProElemtreeElementGet PRO_E_SKETCHER ");

status = ProElementValueGet ( pro_e_std_matrlside, &value );
C_LOG( " ProElementValueGet" );

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_MATERIAL_BOTH_SIDES ;
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_matrlside, value );
C_LOG( " ProElementValueSet" );

ProElempathFree (&path);

/*-----*\
Populating element PRO_E_STD_DIRECTION (must be done once section is
set)
\*-----*/

C_PRINT( " *** Processing Element PRO_E_STD_DIRECTION *** " );
path_items[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[0].path_item.elem_id = PRO_E_STD_DIRECTION;

status = ProElempathAlloc (&path);
status = ProElempathDataSet (path, path_items, 1);

status = ProElemtreeElementGet ( created_elemtree, path,
&pro_e_std_direction);
C_LOG (" ProElemtreeElementGet PRO_E_STD_DIRECTION ");

status = ProElementValueGet ( pro_e_std_direction, &value );
C_LOG( " ProElementValueGet" );

```

```

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_CR_IN_SIDE_TWO;
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_direction, value );
C_LOG( " ProElementValueSet" );

ProElempathFree (&path);

/*-----*\
   Populating element PRO_E_TRIM_QLT_SIDE (must be done once material
side is BOTH)
\*-----*/

C_PRINT( " *** Processing Element PRO_E_TRIM_QLT_SIDE *** " );
path_items[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[0].path_item.elem_id = PRO_E_TRIM_QLT_SIDE;

status = ProElempathAlloc (&path);
status = ProElempathDataSet (path, path_items, 1);

status = ProElementtreeElementGet ( created_elementtree, path,
&pro_e_trim_qlt_side);
C_LOG ( " ProElementtreeElementGet PRO_E_TRIM_QLT_SIDE " );

status = ProElementValueGet ( pro_e_trim_qlt_side, &value );
C_LOG( " ProElementValueGet" );

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_TRIMQLT_SIDE_ONE;
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_trim_qlt_side, value );
C_LOG( " ProElementValueSet" );

ProElempathFree (&path);

/*-----*\
   Redefining the feature to make it complete.
\*-----*/
opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMS;
status = ProSelectionAsmcomppathGet (model_sel, &comp_path);
C_LOG ( " ProSelectionAsmcomppathGet");

status = ProFeatureRedefine (&comp_path, &feature, created_elementtree,
opts, 1, &errors);
C_LOG ( " ProFeatureRedefine");

/*-----*\
   Free up the allocated memory.
\*-----*/

```

```

status = ProFeatureElemtreeFree (&feature, created_elemtree );
C_LOG (" ProFeatureElemtreeFree");

status = ProElementFree (&pro_e_feature_tree );
C_LOG (" ProElementFree");

free(sketch_refs);

return (status);
}

```

Direct Creation Approach

The example shows how to create an extruded protrusion by the direct approach for sketched features introduced in Pro/ENGINEER Wildfire. The user is prompted to select a sketched datum curve feature that is used as a section for the created protrusion.

The source code for the application is as follows:

```

/*----- Pro/Toolkit Includes -----*/
#include "ProToolkit.h"
#include "ProFeature.h"
#include "ProElemId.h"
#include "ProExtrude.h"
#include "ProModFeat.h"
#include "ProStdSection.h"
#include "ProElement.h"
#include "ProElemPath.h"
#include "ProFeatType.h"
#include "ProFeatForm.h"
#include "ProSelection.h"
#include "ProSection.h"

static ProFileName message_file;

#define C_LOG(a) ProTKPrintf ("Status for %s is = %d\n", a, status ); \
ERROR_CHECK( a, "UgSktExtrusionCreate.c", status );

/*----- Function Prototypes -----*/
int ProDemoBaseExtrudeProtrCreate();

/*----- External Data -----*/
ProError ProDemoSectCreate();

/*----- Global Data -----*/
typedef struct tree_element_data
{
    ProElement    tree;
    ProElement    parent_element;
}

```

```

    ProElemId      elem_id;
    ProValueData   value_data;
} ProTreeElemdata;

ProError ProTestFeatElemAdd (ProTreeElemdata *elem);

/*=====*\
FUNCTION : ProDemoBaseExtrudeProtrCreate
PURPOSE  : Demonstrates the creation of the extruded protrusion
           base feature.
\*=====*/
int ProDemoBaseExtrudeProtrCreate()
{
    ProTreeElemdata      elem;
    ProErrorlist          errors;
    ProMdl                model;
    ProModelitem          model_item;
    ProSelection          model_sel;
    ProFeature            feature;
    ProFeatureCreateOptions opts[1];
    ProElempath           path;
    ProElempathItem       path_items[2];
    ProSection            section;
    ProAsmcomppath        comp_path;
    ProAsmcomppath        *p_comp_path = NULL;
    ProElement            parent_elem;
    ProElement            parent_elem2;
    ProValue              value;
    double                width;
    double                height;
    double                bite_radius;
    double                bite_height;
    char                  name[PRO_NAME_SIZE];
    ProBoolean            alloc;
    ProError              status;
    ProSelection *        sketch_selection;
    ProSelection *        selection_array;
    int                   n_select;

    ProStringToWstring ( message_file, "utilities.txt" );
/*-----*\
    Allocate the element tree.
\*-----*/
    status = ProElementAlloc (PRO_E_FEATURE_TREE, &(elem.tree));
    C_LOG(" ProElementAlloc");

/*-----*\
    Add the feature type element to the tree.
\*-----*/
    elem.parent_element = elem.tree;
    elem.elem_id = PRO_E_FEATURE_TYPE;

```



```

elem.value_data.type = PRO_VALUE_TYPE_INT;
elem.value_data.v.i = PRO_FEAT_PROTRUSION;
status = ProTestFeatElemAdd (&elem);
C_LOG(" ProTestFeatElemAdd");

/*-----*\
Add the feature form element to the tree.
\*-----*/
elem.parent_element = elem.tree;
elem.elem_id = PRO_E_FEATURE_FORM;
elem.value_data.type = PRO_VALUE_TYPE_INT;
elem.value_data.v.i = PRO_EXTRUDE;
status = ProTestFeatElemAdd (&elem);
C_LOG(" ProTestFeatElemAdd");

/*-----*\
Add the feature solid/surface/cut element to the tree.
\*-----*/
elem.parent_element = elem.tree;
elem.elem_id = PRO_E_EXT_SURF_CUT_SOLID_TYPE;
elem.value_data.type = PRO_VALUE_TYPE_INT;
elem.value_data.v.i = PRO_EXT_FEAT_TYPE_SOLID;
status = ProTestFeatElemAdd (&elem);
C_LOG(" ProTestFeatElemAdd");

/*-----*\
Add the feature addition/removal material element to the tree.
\*-----*/
elem.parent_element = elem.tree;
elem.elem_id = PRO_E_REMOVE_MATERIAL;
elem.value_data.type = PRO_VALUE_TYPE_INT;
elem.value_data.v.i = PRO_EXT_MATERIAL_ADD;
status = ProTestFeatElemAdd (&elem);
C_LOG(" ProTestFeatElemAdd");

/*-----*\
Add the feature thin element to the tree.
\*-----*/
elem.parent_element = elem.tree;
elem.elem_id = PRO_E_FEAT_FORM_IS_THIN;
elem.value_data.type = PRO_VALUE_TYPE_INT;
elem.value_data.v.i = PRO_EXT_FEAT_FORM_NO_THIN;
status = ProTestFeatElemAdd (&elem);
C_LOG(" ProTestFeatElemAdd");

/*-----*\
Add the standard section element to the tree.
\*-----*/
elem.parent_element = elem.tree;
elem.elem_id = PRO_E_STD_SECTION;
elem.value_data.type = -1;

```

```

elem.value_data.v.i = -1;
status = ProTestFeatElemAdd (&elem);
C_LOG(" ProTestFeatElemAdd");

/*-----*\
    Add the section depth elements to the element tree.
\*-----*/

elem.parent_element = elem.tree;
elem.elem_id = PRO_E_STD_EXT_DEPTH;
elem.value_data.type = -1;
elem.value_data.v.i = -1;
status = ProTestFeatElemAdd (&elem);
C_LOG(" ProTestFeatElemAdd");
parent_elem2 = elem.parent_element;

elem.elem_id = PRO_E_EXT_DEPTH_FROM;
elem.value_data.type = -1;
elem.value_data.v.i = -1;
status = ProTestFeatElemAdd (&elem);
C_LOG(" ProTestFeatElemAdd");
parent_elem = elem.parent_element;
elem.elem_id = PRO_E_EXT_DEPTH_FROM_TYPE;
elem.value_data.type = PRO_VALUE_TYPE_INT;
elem.value_data.v.i = PRO_EXT_DEPTH_FROM_BLIND;
status = ProTestFeatElemAdd (&elem);
C_LOG(" ProTestFeatElemAdd");
elem.parent_element = parent_elem;
elem.elem_id = PRO_E_EXT_DEPTH_FROM_VALUE;
elem.value_data.type = PRO_VALUE_TYPE_DOUBLE;
elem.value_data.v.d = 50.0;
status = ProTestFeatElemAdd (&elem);
C_LOG(" ProTestFeatElemAdd");

elem.parent_element = parent_elem2;
elem.elem_id = PRO_E_EXT_DEPTH_TO;
elem.value_data.type = -1;
elem.value_data.v.i = -1;
status = ProTestFeatElemAdd (&elem);
C_LOG(" ProTestFeatElemAdd");
parent_elem = elem.parent_element;
elem.elem_id = PRO_E_EXT_DEPTH_TO_TYPE;
elem.value_data.type = PRO_VALUE_TYPE_INT;
elem.value_data.v.i = PRO_EXT_DEPTH_TO_BLIND;
status = ProTestFeatElemAdd (&elem);
C_LOG(" ProTestFeatElemAdd");
elem.parent_element = parent_elem;
elem.elem_id = PRO_E_EXT_DEPTH_TO_VALUE;
elem.value_data.type = PRO_VALUE_TYPE_DOUBLE;
elem.value_data.v.d = 0.0;
status = ProTestFeatElemAdd (&elem);
C_LOG(" ProTestFeatElemAdd");

```

```

status = ProMdlCurrentGet (&model);
C_LOG(" ProMdlCurrentGet");
if ( status != PRO_TK_NO_ERROR ) return ( status );
status = ProMdlToModelitem( model, &model_item );
C_LOG(" ProMdlToModelitem");
status = ProSelectionAlloc (p_comp_path, &model_item,
    &model_sel);
C_LOG(" ProSelectionAlloc");

status = ProMessageDisplay ( message_file,
"Select a Sketched Datum Curve Feature");
C_LOG(" ProMessageDisplay");
ProTKPrintf ( "Please Select a Sketched Datum Curve Feature \n");

status = ProSelect ( "feature", 1, NULL, NULL, NULL, NULL,
    &sketch_selection, &n_select );
C_LOG(" ProSelect");

if ( n_select <= 0 ) return -1;

status = ProArrayAlloc ( n_select, sizeof (ProSelection),
    1, ( ProArray *)&selection_array );
C_LOG(" ProArrayAlloc");

status = ProSelectionCopy ( sketch_selection[0], &selection_array[0]
);
C_LOG(" ProSelectionCopy");

/*-----*\
    Create the protrusion using the section of the selected feature
\*-----*/

opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMS;
status = ProFeatureSketchedCreate ( model_sel, elem.tree,
    opts, 1, selection_array,
    &feature, &errors );
C_LOG(" ProFeatureSketchedCreate");

/*-----*\
    Free up the allocated memory.
\*-----*/

status = ProElementFree (&elem.tree);
C_LOG(" ProElementFree");

return (status);
}

/*=====*\
FUNCTION : ProTestFeatElemAdd
PURPOSE  : Adds a generic feature element to the element tree.

```

```

\*=====*/
ProError ProTestFeatElemAdd (ProTreeElemdata *elem)
{
    ProValue    value;
    ProElement  element;
    ProError    status;
    if (elem->value_data.type != -1)
    {
        status = ProValueAlloc (&value);
        C_LOG(" ProValueAlloc");
        status = ProValueDataSet (value, &elem->value_data);
        C_LOG(" ProValueDataSet");
    }
    status = ProElementAlloc (elem->elem_id, &element);
    C_LOG(" ProElementAlloc");

    if (elem->value_data.type != -1)
    {
        status = ProElementValueSet (element, value);
        C_LOG(" ProElementValueSet");
    }
    status = ProElemtreeElementAdd (elem->parent_element, NULL, element);
    C_LOG(" ProElemtreeElementAdd");
    elem->parent_element = element;
    return (status);
}

```

The Element Tree for Revolved Features

The element tree for revolved features is documented in the header file *ProRevolve.h*, and has a fairly simple structure. It shows that, apart from the usual elements for the tree root and feature name, a revolved feature tree contains the elements to make the feature a solid protrusion, a thin protrusion, a solid cut, a thin cut, a surface, a surface trimmed feature, or a thin surface trimmed feature.

You can use Intent Datums such as Intent Axis, Intent Plane, and Intent Point for placement references.

The following figure shows the element tree for revolved features.

Figure 42-2: Element Tree for Revolved Feature

```

PRO_E_FEATURE_TREE
|--PRO_E_STD_FEATURE_NAME
|--PRO_E_EXT_SURF_CUT_SOLID_TYPE
|--PRO_E_REMOVE_MATERIAL
|--PRO_E_FEATURE_FORM
|--PRO_E_STD_SECTION          ( Sketch )
|--PRO_E_FEAT_FORM_IS_THIN    ( Feature Form )
|--PRO_E_STD_MATRLSIDE        ( Material Side )
|--PRO_E_THICKNESS            ( Thickness )
|--PRO_E_SRF_END_ATTRIBUTES
|--PRO_E_TRIM_QLT_SIDE
|--PRO_E_TRIM_QUILT
|--PRO_E_STD_DIRECTION
|--PRO_E_REVOLVE_AXIS
|--PRO_E_REVOLVE_AXIS_OPT
|--PRO_E_REV_ANGLE
    |--PRO_E_REV_ANGLE_FROM
        |--PRO_E_REV_ANGLE_FROM_TYPE
        |--PRO_E_REV_ANGLE_FROM_REF
        |--PRO_E_REV_ANGLE_FROM_VAL
    |--PRO_E_REV_ANGLE_TO
        |--PRO_E_REV_ANGLE_TO_TYPE
        |--PRO_E_REV_ANGLE_TO_REF
        |--PRO_E_REV_ANGLE_TO_VAL

```

Element Trees:
Extrude and Revolve

The elements are assigned values depending on the type of revolved feature you want to create.

The following table lists the common elements for all types of revolved features and their permissible values:

Element ID	Value
PRO_E_FEATURE_TYPE	Feature type: PRO_FEAT_PROTRUSION PRO_FEAT_CUT PRO_FEAT_DATUM_SURF Not required for creation.
PRO_E_FEATURE_FORM	Mandatory= PRO_REVOLVE
PRO_E_EXT_SURF_CUT_SOLID_TYPE	Mandatory Of type ProRevFeatType = PRO_REV_FEAT_TYPE_SOLID for Solid feature type = PRO_REV_FEAT_TYPE_SURFACE for Surface feature type
PRO_E_FEAT_FORM_IS_THIN	Feature Form Of Type ProRevFeatForm = PRO_REV_FEAT_FORM_NO_THIN for a Solid feature = PRO_REV_FEAT_FORM_THIN for a Thin feature
PRO_E_REMOVE_MATERIAL	Material Removal Of type ProRevRemMaterial = PRO_REV_MATERIAL_ADD for a Protruded feature = PRO_REV_MATERIAL_REMOVE for a Cut feature
PRO_E_STD_SECTION	Standard section elements.
PRO_E_STD_DIRECTION*	Direction of creation. Of type ProRevDirection = PRO_REV_CR_IN_SIDE_ONE for angle in side one = PRO_REV_CR_IN_SIDE_TWO for angle in side two
PRO_E_STD_MATRLSIDE*	Direction of material affected with respect to the sketch. It is required for all cuts, all thin features, and for solid protrusions with open sections.

Element ID	Value
PRO_E_REVOLVE_AXIS_OPT	Optional, of the type ProRevAxisOptAttr. Identifies if the axis to revolve about is a part of the sketch or an external datum axis.
PRO_E_REVOLVE_AXIS	Optional. Reference to external datum axis, if PRO_E_REVOLVE_AXIS = PRO_REV_AXIS_EXT_REF
PRO_E_REV_ANGLE	Compound Element
PRO_E_REV_ANGLE_TO	Compound Element
PRO_E_REV_ANGLE_TO_TYPE	Mandatory Of type ProRevAngleToType
PRO_E_REV_ANGLE_TO_VAL	Depends on PRO_E_REV_ANGLE_TO_TYPE Of type PRO_VALUE_TYPE_DOUBLE (in degrees)
PRO_E_REV_ANGLE_TO_REF	Depends on PRO_E_REV_ANGLE_TO_TYPE Of type listed in the Angle Type table that follows.
PRO_E_REV_ANGLE_FROM	Compound Element
PRO_E_REV_ANGLE_FROM_TYPE	Mandatory Of type ProRevAngleFromType
PRO_E_REV_ANGLE_FROM_VAL	Depends on PRO_E_REV_ANGLE_FROM_TYPE Of type PRO_VALUE_TYPE_DOUBLE (in degrees)
PRO_E_REV_ANGLE_FROM_REF	Depends on PRO_E_REV_ANGLE_FROM_TYPE Of type listed in the Angle Type table that follows.
PRO_E_STD_FEATURE_NAME	Default given by application depending on the feature type. Can be modified by the user.

Elements identified with “*” depend on the definition of the standard section. These elements may not be assigned values until the standard section has been completely allocated (which typically happens during redefine of the feature). Values assigned to these elements while the section is not complete are ignored.

The following table lists the angle types for revolved features along with possible valid references:

Angle Type	Valid Reference Types
PRO_REV_ANGLE_TO_REF	PRO_POINT, PRO_EDGE_START, PRO_EDGE_END, PRO_CRV_START, PRO_CRV_END, PRO_SURFACE (Plane).
PRO_REV_ANGLE_FROM_REF	PRO_POINT, PRO_EDGE_START, PRO_EDGE_END, PRO_CRV_START, PRO_CRV_END, PRO_SURFACE (Plane).

The following table lists the elements needed to create revolved features, in addition to those already discussed:

Feature Type	Element ID	Comment
Solid		No Additional Elements Required
Thin	PRO_E_THICKNESS	Mandatory ≥ 0.0 Of type PRO_VALUE_TYPE_DOUBLE
Solid Cut	PRO_E_STD_MATRLSIDE	Mandatory Of type ProRevMatlSide
Thin Cut	PRO_E_STD_MATRLSIDE	Mandatory Of type ProRevMatlSide
	PRO_E_THICKNESS	Mandatory ≥ 0.0 Of type PRO_VALUE_TYPE_DOUBLE
Surface	PRO_E_SRF_END_ATTRIBUTES	Mandatory Of type ProRevSurfEndAttr Must be assigned at the same time or after the section is fully completed.

Feature Type	Element ID	Comment
Surface Trim	PRO_E_STD_MATRLSIDE	Mandatory Of type ProRevMatlSide
	PRO_E_TRIM_QUILT	Mandatory Of type Quilt
	PRO_E_TRIM_QLT_SIDE	Mandatory Of type ProRevTrimQltSide if PRO_E_STD_MATRLSIDE is “both”. Must be assigned at the same time as PRO_E_STD_MATRLSIDE.
	PRO_E_SRF_END_ATTRIBUTES	Mandatory Of type ProRevSurfEndAttr
Thin Surface Trim	PRO_E_STD_MATRLSIDE	Mandatory Of type ProRevMatlSide
	PRO_E_THICKNESS	Mandatory ≥ 0.0 Of type PRO_VALUE_TYPE_DOUBLE
	PRO_E_TRIM_QUILT	Mandatory Of type Quilt
	PRO_E_TRIM_QLT_SIDE	Mandatory Of type ProRevTrimQltSide
	PRO_E_SRF_END_ATTRIBUTES	Mandatory Of type ProRevSurfEndAttr

Examples: Creating Revolved Features

The following examples demonstrate creation of revolved features of various forms. These examples are adapted from an example template file *UgSktRevolveTemplate.c* available on the Pro/ENGINEER load point under
protoolkit/protk_appls/pt_userguide/ptu_featcreat.

- Example 6: To Create a Revolved Protrusion
- Example 7: To Create a Revolved Thin Cut
- Example 8: To Create a Revolved Surface

Example 6: To Create a Revolved Protrusion

The following example shows how to create a revolved protrusion feature with symmetric depth.

```
/*=====*\
FILE      : UgSktRevolveProtrusion.c
PURPOSE   : Creates a revolved protrusion

\*=====*/

/*----- Pro/Toolkit Includes -----*/
#include "ProToolkit.h"
#include "ProFeature.h"
#include "ProElemId.h"
#include "ProExtrude.h"
#include "ProModFeat.h"
#include "ProStdSection.h"
#include "ProElement.h"
#include "ProElempath.h"
#include "ProFeatType.h"
#include "ProFeatForm.h"
#include "ProSelection.h"
#include "ProSection.h"

#include "ProRevolve.h"

static ProFileName message_file;

#define C_LOG(a) ProTKPrintf ("Status for %s is = %d\n", a, status ); \
ERROR_CHECK( a, "UgSktRevolveCreate.c", status );

#define C_PRINT(a) ProTKPrintf ( "%s\n", a);

/*=====*\
FUNCTION   : UserSktRevolveProtrusion()
PURPOSE    : Create a revolved protrusion (symmetric).
\*=====*/
ProError UserSktRevolveProtrusion()
{
    ProErrorlist      errors;
    ProMdl             model;
    ProModelitem       model_item;
    ProSelection       model_sel;
    ProFeature         feature;
    ProFeatureCreateOptions opts[1];
    ProElempath        path;
    ProElempathItem    path_items[2];
    ProSection         section;
    ProAsmcomppath     comp_path;
    ProAsmcomppath     *p_comp_path = NULL;
```

```

ProValue          value;

ProElement sketch_element;
ProElement created_elemtree;

ProElement pro_e_feature_tree;
ProElement pro_e_feature_type;
ProElement pro_e_feature_form;
ProElement pro_e_ext_surf_cut_solid_type;
ProElement pro_e_remove_material;
ProElement pro_e_feat_form_is_thin;

ProElement pro_e_rev_angle;
ProElement pro_e_rev_angle_from;
ProElement pro_e_rev_angle_from_type;
ProElement pro_e_rev_angle_from_val;
ProElement pro_e_rev_angle_to;
ProElement pro_e_rev_angle_to_type;

ProElement pro_e_std_section;
ProElement pro_e_std_sec_method;
ProElement pro_e_std_sec_setup_plane;
ProElement pro_e_std_sec_plane;
ProElement pro_e_std_sec_plane_view_dir;
ProElement pro_e_std_sec_plane_orient_dir;
ProElement pro_e_std_sec_plane_orient_ref;

ProSelection *sketch_refs;
ProError status;
ProValueData value_data;
ProSelection * p_select;
int n_select;

ProStringToWstring ( message_file, "utilities.txt" );

log_file = PTApplsUnicodeFopen ( "ug_sketched_curve.log", "w" );

status = ProMdlCurrentGet (&model);
if ( status != PRO_TK_NO_ERROR ) return ( status );

/*-----*\
   Populating root element PRO_E_FEATURE_TREE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TREE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TREE, &pro_e_feature_tree );
C_LOG( " ProElementAlloc " );

/*-----*\
   Populating element PRO_E_FEATURE_FORM
\*-----*/

```

```

C_PRINT( " *** Processing Element PRO_E_FEATURE_FORM *** " );
status = ProElementAlloc ( PRO_E_FEATURE_FORM, &pro_e_feature_form );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REVOLVE;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_feature_form, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_feature_form );
C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Populating element PRO_E_EXT_SURF_CUT_SOLID_TYPE
\*-----*/

C_PRINT( " *** Processing Element PRO_E_EXT_SURF_CUT_SOLID_TYPE *** " );
status = ProElementAlloc ( PRO_E_EXT_SURF_CUT_SOLID_TYPE,
                          &pro_e_ext_surf_cut_solid_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_FEAT_TYPE_SOLID;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_ext_surf_cut_solid_type, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_ext_surf_cut_solid_type );
C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Populating element PRO_E_REMOVE_MATERIAL
\*-----*/

C_PRINT( " *** Processing Element PRO_E_REMOVE_MATERIAL *** " );
status = ProElementAlloc ( PRO_E_REMOVE_MATERIAL, &pro_e_remove_material
);
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_MATERIAL_ADD;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );

```

```

status = ProElementValueSet ( pro_e_remove_material, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_remove_material );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_FEAT_FORM_IS_THIN
\*-----*/

C_PRINT( " *** Processing Element PRO_E_FEAT_FORM_IS_THIN *** " );
status = ProElementAlloc ( PRO_E_FEAT_FORM_IS_THIN,
                          &pro_e_feat_form_is_thin );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_FEAT_FORM_NO_THIN;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_feat_form_is_thin, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_feat_form_is_thin );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating compound element PRO_E_STD_REV_ANGLE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_REV_ANGLE *** " );
status = ProElementAlloc ( PRO_E_REV_ANGLE, &pro_e_rev_angle );
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_rev_angle );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_REV_ANGLE
-> PRO_E_REV_ANGLE_FROM
\*-----*/
C_PRINT( " *** Processing Element PRO_E_REV_ANGLE_FROM *** " );
status = ProElementAlloc ( PRO_E_REV_ANGLE_FROM, &pro_e_rev_angle_from
);
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_rev_angle, NULL,
                                pro_e_rev_angle_from );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_REV_ANGLE
-> PRO_E_REV_ANGLE_FROM

```

```

-> PRO_E_REV_ANGLE_FROM_TYPE
/*-----*/
C_PRINT( " *** Processing Element PRO_E_REV_ANGLE_FROM_TYPE *** " );
status = ProElementAlloc ( PRO_E_REV_ANGLE_FROM_TYPE,
                          &pro_e_rev_angle_from_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_ANG_FROM_ANGLE;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_rev_angle_from_type, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_rev_angle_from, NULL,
                                pro_e_rev_angle_from_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_REV_ANGLE
-> PRO_E_REV_ANGLE_FROM
    -> PRO_E_REV_ANGLE_FROM_VAL
/*-----*/
C_PRINT( " *** Processing Element PRO_E_REV_ANGLE_FROM_VAL *** " );
status = ProElementAlloc ( PRO_E_REV_ANGLE_FROM_VAL,
                          &pro_e_rev_angle_from_val );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 120.000000;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_rev_angle_from_val, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_rev_angle_from, NULL,
                                pro_e_rev_angle_from_val );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_REV_ANGLE
-> PRO_E_REV_ANGLE_TO
/*-----*/
C_PRINT( " *** Processing Element PRO_E_REV_ANGLE_TO *** " );
status = ProElementAlloc ( PRO_E_REV_ANGLE_TO, &pro_e_rev_angle_to );
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_rev_angle, NULL,
                                pro_e_rev_angle_to );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\

```

```

    Populating element PRO_E_REV_ANGLE
-> PRO_E_REV_ANGLE_TO
    -> PRO_E_REV_ANGLE_TO_TYPE
    /*-----*/
    C_PRINT( " *** Processing Element PRO_E_REV_ANGLE_TO_TYPE *** " );
    status = ProElementAlloc ( PRO_E_REV_ANGLE_TO_TYPE,
                              &pro_e_rev_angle_to_type );
    C_LOG( " ProElementAlloc " );
    value_data.type = PRO_VALUE_TYPE_INT;
    value_data.v.i = PRO_REV_ANG_SYMMETRIC;
    status = ProValueAlloc ( &value );
    C_LOG( " ProValueAlloc " );
    status = ProValueDataSet ( value, &value_data );
    C_LOG( " ProValueDataSet " );
    status = ProElementValueSet ( pro_e_rev_angle_to_type, value );
    C_LOG( " ProElementValueSet " );
    status = ProElemtreeElementAdd ( pro_e_rev_angle_to, NULL,
                                     pro_e_rev_angle_to_type );
    C_LOG( " ProElemtreeElementAdd " );

    /*-----*/
    Populating element PRO_E_STD_SECTION
    /*-----*/

    C_PRINT( " *** Processing Element PRO_E_STD_SECTION *** " );
    status = ProElementAlloc ( PRO_E_STD_SECTION, &pro_e_std_section );
    C_LOG( " ProElementAlloc " );
    status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
    pro_e_std_section );
    C_LOG( " ProElemtreeElementAdd " );

    /*-----*/
    Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    /*-----*/
    C_PRINT( " *** Processing Element PRO_E_STD_SEC_SETUP_PLANE *** " );
    status = ProElementAlloc ( PRO_E_STD_SEC_SETUP_PLANE,
    &pro_e_std_sec_setup_plane );
    C_LOG( " ProElementAlloc " );
    status = ProElemtreeElementAdd( pro_e_std_section, NULL,
    pro_e_std_sec_setup_plane );
    C_LOG( " ProElemtreeElementAdd " );

    sketch_refs = ( ProSelection *) calloc ( 2, sizeof ( ProSelection ));

    /*-----*/
    Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    -> PRO_E_STD_SEC_PLANE
    /*-----*/

```

```

C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE *** " );
status = ProMessageDisplay ( message_file, "Select Surface for sketch
                                placement");
C_LOG( " ProMessageDisplay" );
ProTKPrintf ( "Please select datum,surface,sldface,qltface_ID_5 type of
                                Modelitem\n");
status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL, NULL,
NULL, NULL, &p_select, &n_select );
C_LOG( " ProSelect" );
if ( n_select <= 0 ) return -1;
else
{
    status = ProSelectionCopy( p_select[0], &sketch_refs[0]);
    C_LOG ( " ProSelectionCopy " );
}
status = ProElementAlloc ( PRO_E_STD_SEC_PLANE, &pro_e_std_sec_plane );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_sec_plane, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    -> PRO_E_STD_SEC_PLANE_VIEW_DIR
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_VIEW_DIR *** " );
status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_VIEW_DIR,
&pro_e_std_sec_plane_view_dir );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_SEC_VIEW_DIR_SIDE_ONE /* PRO_SEC_VIEW_DIR_SIDE_TWO
*/;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_sec_plane_view_dir, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_view_dir );
C_LOG( " ProElemtreeElementAdd" );

```



```

/*-----*\
    Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    -> PRO_E_STD_SEC_PLANE_ORIENT_DIR
/*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_ORIENT_DIR *** "
);
    status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_ORIENT_DIR,
&pro_e_std_sec_plane_orient_dir );
    C_LOG( " ProElementAlloc " );
    value_data.type = PRO_VALUE_TYPE_INT;
    value_data.v.i = PRO_SEC_ORIENT_DIR_UP /* PRO_SEC_ORIENT_DIR_DOWN */
/* PRO_SEC_ORIENT_DIR_LEFT *//* PRO_SEC_ORIENT_DIR_RIGHT */ ;
    status = ProValueAlloc ( &value );
    C_LOG( " ProValueAlloc " );
    status = ProValueDataSet ( value, &value_data );
    C_LOG( " ProValueDataSet " );
    status = ProElementValueSet ( pro_e_std_sec_plane_orient_dir, value );
    C_LOG( " ProElementValueSet " );
    status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_orient_dir );
    C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    -> PRO_E_STD_SEC_PLANE_ORIENT_REF
/*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_ORIENT_REF *** "
);
    status = ProMessageDisplay ( message_file,
                                "Select Surface for sketch orientation");
    C_LOG( " ProMessageDisplay " );
    ProTKPrintf ( "Select datum,surface,sldface,qltface_ID_5 type of
                                Modelitem\n");
    status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL,
NULL, NULL, NULL, &p_select, &n_select );
    C_LOG( " ProSelect " );
    if ( n_select <= 0 ) return -1;
    else
    {
        status = ProSelectionCopy( p_select[0], &sketch_refs[1]);
        C_LOG ( " ProSelectionCopy " );
    }

    status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_ORIENT_REF,
&pro_e_std_sec_plane_orient_ref );
    C_LOG( " ProElementAlloc " );
    value_data.type = PRO_VALUE_TYPE_SELECTION;
    value_data.v.r = p_select[0];

```

```

status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_sec_plane_orient_ref, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_orient_ref );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Creating incomplete feature in the current model.
\*-----*/
status = ProMdlToModelitem( model, &model_item );
status = ProSelectionAlloc (p_comp_path, &model_item,
&model_sel);

opts[0] = PRO_FEAT_CR_INCOMPLETE_FEAT;
status = ProFeatureCreate (model_sel, pro_e_feature_tree, opts, 1,
&feature, &errors);
C_LOG ( " ProFeatureCreate");

/* Using the element tree from created feature */
status = ProFeatureElemtreeCreate ( &feature, &created_elemtree );
C_LOG ( " ProFeatureElemtreeCreate");

/*-----*\
    Getting the initialized section element from the database.
\*-----*/
/* path to PRO_E_SKETCHER element */
path_items[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[0].path_item.elem_id = PRO_E_STD_SECTION;
path_items[1].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[1].path_item.elem_id = PRO_E_SKETCHER;
status = ProElempathAlloc (&path);
status = ProElempathDataSet (path, path_items, 2);

status = ProElemtreeElementGet ( created_elemtree, path,
&sketch_element);
C_LOG ( " ProElemtreeElementGet PRO_E_SKETCHER ");

status = ProElementValueGet ( sketch_element, &value);
C_LOG ( " ProElementValueGet PRO_E_SKETCHER ");

status = ProValueDataGet (value, &value_data);
C_LOG ( " ProValueDataGet PRO_E_SKETCHER ");
section = (ProSection)value_data.v.p;

/*-----*\
    Creating a 3-D section
\*-----*/

```

```

status = UserSectionBuild (( ProSection ) (value_data.v.p),
    sketch_refs );
C_LOG ("UserSectionBuild");

ProElempathFree (&path);

/*-----*\
    Redefining the feature to make it complete.
\*-----*/
opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMS;
status = ProSelectionAsmcomppathGet (model_sel, &comp_path);
C_LOG (" ProSelectionAsmcomppathGet");

status = ProFeatureRedefine (&comp_path, &feature, created_elemtree,
    opts, 1, &errors);
C_LOG (" ProFeatureRedefine");

/*-----*\
    Free up the allocated memory.
\*-----*/
status = ProFeatureElemtreeFree (&feature, created_elemtree );
C_LOG (" ProFeatureElemtreeFree");

status = ProElementFree (&pro_e_feature_tree );
C_LOG (" ProElementFree");

free(sketch_refs);

return (status);
}

```

Example 7: To Create a Revolved Thin Cut

The following example shows how to create a revolved thin cut, with independent angular dimensions for both sides.

```

/*=====*\
FILE      : UserSktRevolveThinCut.c
PURPOSE   : Creates a revolved thin cut.
\*=====*/

/*----- Pro/Toolkit Includes -----*/
#include "ProToolkit.h"
#include "ProFeature.h"
#include "ProElemId.h"
#include "ProExtrude.h"
#include "ProModFeat.h"
#include "ProStdSection.h"
#include "ProElement.h"

```

```

#include "ProElempath.h"
#include "ProFeatType.h"
#include "ProFeatForm.h"
#include "ProSelection.h"
#include "ProSection.h"

#include "ProRevolve.h"

static ProFileName message_file;

#define C_LOG(a) ProTKPrintf ("Status for %s is = %d\n", a, status ); \
ERROR_CHECK( a, "UgSktRevolveCreate.c", status );

#define C_PRINT(a) ProTKPrintf ( "%s\n", a);

/*=====*\
FUNCTION : UserSktRevolveThinCut()
PURPOSE  : Template function for creation of extrude features.
\*=====*/
ProError UserSktRevolveThinCut()
{
    ProErrorlist          errors;
    ProMdl                model;
    ProModelitem          model_item;
    ProSelection          model_sel;
    ProFeature            feature;
    ProFeatureCreateOptions opts[1];
    ProElempath           path;
    ProElempathItem       path_items[2];
    ProSection            section;
    ProAsmcomppath        comp_path;
    ProAsmcomppath        *p_comp_path = NULL;
    ProValue              value;

    ProElement sketch_element;
    ProElement created_elemtree;

    ProElement pro_e_feature_tree;
    ProElement pro_e_feature_type;
    ProElement pro_e_feature_form;
    ProElement pro_e_ext_surf_cut_solid_type;
    ProElement pro_e_remove_material;
    ProElement pro_e_feat_form_is_thin;
    ProElement pro_e_thickness;
    ProElement pro_e_std_matrlside;
    ProElement pro_e_std_direction;

    ProElement pro_e_rev_angle;
    ProElement pro_e_rev_angle_from;
    ProElement pro_e_rev_angle_from_type;
    ProElement pro_e_rev_angle_from_val;

```

```

ProElement pro_e_rev_angle_to;
ProElement pro_e_rev_angle_to_type;
ProElement pro_e_rev_angle_to_val;

ProElement pro_e_std_section;
ProElement pro_e_std_sec_method;
ProElement pro_e_std_sec_setup_plane;
ProElement pro_e_std_sec_plane;
ProElement pro_e_std_sec_plane_view_dir;
ProElement pro_e_std_sec_plane_orient_dir;
ProElement pro_e_std_sec_plane_orient_ref;

ProSelection *sketch_refs;
ProError status;
ProValueData value_data;
ProSelection * p_select;
int n_select;

ProStringToWstring ( message_file, "utilities.txt" );

log_file = PTApplsUnicodeFopen ( "ug_sketched_curve.log", "w" );

status = ProMdlCurrentGet (&model);
if ( status != PRO_TK_NO_ERROR ) return ( status );

/*-----*\
   Populating root element PRO_E_FEATURE_TREE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TREE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TREE, &pro_e_feature_tree );
C_LOG( " ProElementAlloc " );

/*-----*\
   Populating element PRO_E_FEATURE_FORM
\*-----*/

C_PRINT( " *** Processing Element PRO_E_FEATURE_FORM *** " );
status = ProElementAlloc ( PRO_E_FEATURE_FORM, &pro_e_feature_form );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REVOLVE;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_feature_form, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_feature_form );
C_LOG( " ProElemtreeElementAdd" );

```

```

/*-----*\
    Populating element PRO_E_EXT_SURF_CUT_SOLID_TYPE
\*-----*/

C_PRINT( " *** Processing Element PRO_E_EXT_SURF_CUT_SOLID_TYPE *** " );
status = ProElementAlloc ( PRO_E_EXT_SURF_CUT_SOLID_TYPE,
                          &pro_e_ext_surf_cut_solid_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_FEAT_TYPE_SOLID;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_ext_surf_cut_solid_type, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_ext_surf_cut_solid_type );
C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Populating element PRO_E_REMOVE_MATERIAL
\*-----*/

C_PRINT( " *** Processing Element PRO_E_REMOVE_MATERIAL *** " );
status = ProElementAlloc ( PRO_E_REMOVE_MATERIAL, &pro_e_remove_material
);
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_MATERIAL_REMOVE;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_remove_material, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_remove_material );
C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Populating element PRO_E_FEAT_FORM_IS_THIN
\*-----*/

C_PRINT( " *** Processing Element PRO_E_FEAT_FORM_IS_THIN *** " );
status = ProElementAlloc ( PRO_E_FEAT_FORM_IS_THIN,
                          &pro_e_feat_form_is_thin );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_FEAT_FORM_THIN;

```

```

status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_feat_form_is_thin, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_feat_form_is_thin );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_THICKNESS (thin only)
\*-----*/

C_PRINT( " *** Processing Element PRO_E_THICKNESS *** " );
status = ProElementAlloc ( PRO_E_THICKNESS, &pro_e_thickness );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 1.00;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_thickness, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_thickness );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating compound element PRO_E_STD_REV_ANGLE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_REV_ANGLE *** " );
status = ProElementAlloc ( PRO_E_REV_ANGLE, &pro_e_rev_angle );
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_rev_angle );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_REV_ANGLE
-> PRO_E_REV_ANGLE_FROM
\*-----*/
C_PRINT( " *** Processing Element PRO_E_REV_ANGLE_FROM *** " );
status = ProElementAlloc ( PRO_E_REV_ANGLE_FROM, &pro_e_rev_angle_from
);
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_rev_angle, NULL,
                                pro_e_rev_angle_from );
C_LOG( " ProElemtreeElementAdd" );

```

```

/*-----*\
    Populating element PRO_E_REV_ANGLE
-> PRO_E_REV_ANGLE_FROM
    -> PRO_E_REV_ANGLE_FROM_TYPE
/*-----*/
C_PRINT( " *** Processing Element PRO_E_REV_ANGLE_FROM_TYPE *** " );
status = ProElementAlloc ( PRO_E_REV_ANGLE_FROM_TYPE,
                          &pro_e_rev_angle_from_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_ANG_FROM_ANGLE;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_rev_angle_from_type, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_rev_angle_from, NULL,
                                pro_e_rev_angle_from_type );
C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Populating element PRO_E_REV_ANGLE
-> PRO_E_REV_ANGLE_FROM
    -> PRO_E_REV_ANGLE_FROM_VAL
/*-----*/
C_PRINT( " *** Processing Element PRO_E_REV_ANGLE_FROM_VAL *** " );
status = ProElementAlloc ( PRO_E_REV_ANGLE_FROM_VAL,
                          &pro_e_rev_angle_from_val );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 60.000000;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_rev_angle_from_val, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_rev_angle_from, NULL,
                                pro_e_rev_angle_from_val );
C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Populating element PRO_E_REV_ANGLE
-> PRO_E_REV_ANGLE_TO
/*-----*/
C_PRINT( " *** Processing Element PRO_E_REV_ANGLE_TO *** " );
status = ProElementAlloc ( PRO_E_REV_ANGLE_TO, &pro_e_rev_angle_to );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_rev_angle, NULL,
                                pro_e_rev_angle_to );

```



```

C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_REV_ANGLE
-> PRO_E_REV_ANGLE_TO
    -> PRO_E_REV_ANGLE_TO_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_REV_ANGLE_TO_TYPE *** " );
status = ProElementAlloc ( PRO_E_REV_ANGLE_TO_TYPE,
                           &pro_e_rev_angle_to_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_ANG_TO_ANGLE;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_rev_angle_to_type, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_rev_angle_to, NULL,
                                pro_e_rev_angle_to_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_REV_ANGLE
-> PRO_E_REV_ANGLE_TO
    -> PRO_E_REV_ANGLE_TO_VAL
\*-----*/
C_PRINT( " *** Processing Element PRO_E_REV_ANGLE_TO_VAL *** " );
status = ProElementAlloc ( PRO_E_REV_ANGLE_TO_VAL,
                           &pro_e_rev_angle_to_val );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 135.000;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_rev_angle_to_val, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_rev_angle_to, NULL,
                                pro_e_rev_angle_to_val );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_SECTION
\*-----*/

C_PRINT( " *** Processing Element PRO_E_STD_SECTION *** " );
status = ProElementAlloc ( PRO_E_STD_SECTION, &pro_e_std_section );
C_LOG( " ProElementAlloc" );

```

```

    status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_std_section );
    C_LOG( " ProElemtreeElementAdd" );

    /*-----*\
        Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    \*-----*/
    C_PRINT( " *** Processing Element PRO_E_STD_SEC_SETUP_PLANE *** " );
    status = ProElementAlloc ( PRO_E_STD_SEC_SETUP_PLANE,
&pro_e_std_sec_setup_plane );
    C_LOG( " ProElementAlloc" );
    status = ProElemtreeElementAdd ( pro_e_std_section, NULL,
pro_e_std_sec_setup_plane );
    C_LOG( " ProElemtreeElementAdd" );

    sketch_refs = ( ProSelection *) calloc ( 2, sizeof ( ProSelection ));

    /*-----*\
        Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    -> PRO_E_STD_SEC_PLANE
    \*-----*/

    C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE *** " );
    status = ProMessageDisplay ( message_file, "Select Surface for sketch
        placement");
    C_LOG( " ProMessageDisplay" );
    ProTKPrintf ( "Please select datum,surface,sldface,qltface_ID_5 type of
        Modelitem\n");
    status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL, NULL,
NULL, NULL, &p_select, &n_select );
    C_LOG( " ProSelect" );
    if ( n_select <= 0 ) return -1;
    else
    {
        status = ProSelectionCopy( p_select[0], &sketch_refs[0]);
        C_LOG ( " ProSelectionCopy " );
    }
    status = ProElementAlloc ( PRO_E_STD_SEC_PLANE, &pro_e_std_sec_plane );
    C_LOG( " ProElementAlloc " );
    value_data.type = PRO_VALUE_TYPE_SELECTION;
    value_data.v.r = p_select[0];
    status = ProValueAlloc ( &value );
    C_LOG( " ProValueAlloc" );
    status = ProValueDataSet ( value, &value_data );
    C_LOG( " ProValueDataSet" );
    status = ProElementValueSet ( pro_e_std_sec_plane, value );
    C_LOG( " ProElementValueSet" );
    status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane );

```

```

C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    -> PRO_E_STD_SEC_PLANE_VIEW_DIR
    \*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_VIEW_DIR *** " );
status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_VIEW_DIR,
&pro_e_std_sec_plane_view_dir );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_SEC_VIEW_DIR_SIDE_ONE /* PRO_SEC_VIEW_DIR_SIDE_TWO
*/;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_std_sec_plane_view_dir, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_view_dir );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    -> PRO_E_STD_SEC_PLANE_ORIENT_DIR
    \*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_ORIENT_DIR *** "
);
status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_ORIENT_DIR,
&pro_e_std_sec_plane_orient_dir );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_SEC_ORIENT_DIR_UP /* PRO_SEC_ORIENT_DIR_DOWN */
/* PRO_SEC_ORIENT_DIR_LEFT */ /* PRO_SEC_ORIENT_DIR_RIGHT */ ;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_std_sec_plane_orient_dir, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_orient_dir );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    -> PRO_E_STD_SEC_PLANE_ORIENT_REF

```

```

/*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_ORIENT_REF *** "
);
status = ProMessageDisplay ( message_file,
"Select Surface for sketch orientation");
C_LOG( " ProMessageDisplay" );
ProTKPrintf ( "Select datum,surface,sldface,qltface_ID_5 type of
Modelitem\n");
status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL,
NULL, NULL, NULL, &p_select, &n_select );
C_LOG( " ProSelect" );
if ( n_select <= 0 ) return -1;
else
{
status = ProSelectionCopy( p_select[0], &sketch_refs[1]);
C_LOG ( " ProSelectionCopy ");
}

status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_ORIENT_REF,
&pro_e_std_sec_plane_orient_ref );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_sec_plane_orient_ref, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_orient_ref );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
Creating incomplete feature in the current model.
*\-----*/
status = ProMdlToModelitem( model, &model_item );
status = ProSelectionAlloc (p_comp_path, &model_item,
&model_sel);

opts[0] = PRO_FEAT_CR_INCOMPLETE_FEAT;
status = ProFeatureCreate (model_sel, pro_e_feature_tree, opts, 1,
&feature, &errors);
C_LOG ( " ProFeatureCreate");

/* Using the element tree from created feature */
status = ProFeatureElemtreeCreate ( &feature, &created_elemtree );
C_LOG ( " ProFeatureElemtreeCreate");

/*-----*\
Getting the initialized section element from the database.

```

```

\*-----*/
/* path to PRO_E_SKETCHER element */
path_items[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[0].path_item.elem_id = PRO_E_STD_SECTION;
path_items[1].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[1].path_item.elem_id = PRO_E_SKETCHER;
status = ProElempathAlloc (&path);
status = ProElempathDataSet (path, path_items, 2);

status = ProElemtreeElementGet ( created_elemtree, path,
                                &sketch_element);
C_LOG (" ProElemtreeElementGet PRO_E_SKETCHER ");

status = ProElementValueGet ( sketch_element, &value);
C_LOG (" ProElementValueGet PRO_E_SKETCHER ");

status = ProValueDataGet (value, &value_data);
C_LOG (" ProValueDataGet PRO_E_SKETCHER ");
section = (ProSection)value_data.v.p;

/*-----*\
    Creating a 3-D section
\*-----*/

status = UserSectionBuild (( ProSection )(value_data.v.p),
    sketch_refs );
C_LOG ("UserSectionBuild");

ProElempathFree (&path);

/*-----*\
    Populating element PRO_E_STD_MATRLSIDE (must be done once section is
set)
\*-----*/

C_PRINT( " *** Processing Element PRO_E_STD_MATRLSIDE *** " );
path_items[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[0].path_item.elem_id = PRO_E_STD_MATRLSIDE;

status = ProElempathAlloc (&path);
status = ProElempathDataSet (path, path_items, 1);

status = ProElemtreeElementGet ( created_elemtree, path,
                                &pro_e_std_matrlside);
C_LOG (" ProElemtreeElementGet PRO_E_SKETCHER ");

status = ProElementValueGet ( pro_e_std_matrlside, &value );
C_LOG( " ProElementValueGet" );

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_MATERIAL_SIDE_ONE;

```

```

status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_matrlside, value );
C_LOG( " ProElementValueSet" );

ProElempathFree (&path);

/*-----*\
   Populating element PRO_E_STD_DIRECTION (must be done once section is
set)
\*-----*/

C_PRINT( " *** Processing Element PRO_E_STD_DIRECTION *** " );
path_items[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[0].path_item.elem_id = PRO_E_STD_DIRECTION;

status = ProElempathAlloc (&path);
status = ProElempathDataSet (path, path_items, 1);

status = ProElemtreeElementGet ( created_elemtree, path,
                                &pro_e_std_direction);
C_LOG ( " ProElemtreeElementGet PRO_E_SKETCHER " );

status = ProElementValueGet ( pro_e_std_direction, &value );
C_LOG( " ProElementValueGet" );

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_CR_IN_SIDE_TWO;
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_direction, value );
C_LOG( " ProElementValueSet" );

ProElempathFree (&path);

/*-----*\
   Redefining the feature to make it complete.
\*-----*/

opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMS;
status = ProSelectionAsmcomppathGet (model_sel, &comp_path);
C_LOG ( " ProSelectionAsmcomppathGet");

status = ProFeatureRedefine (&comp_path, &feature, created_elemtree,
                             opts, 1, &errors);
C_LOG ( " ProFeatureRedefine");

/*-----*\
   Free up the allocated memory.
\*-----*/

status = ProFeatureElemtreeFree (&feature, created_elemtree );
C_LOG ( " ProFeatureElemtreeFree");

```

```

    status = ProElementFree (&pro_e_feature_tree );
    C_LOG ( " ProElementFree");

    free(sketch_refs);

    return (status);
}

```

Example 8: To Create a Revolved Surface

The example shows how to create a revolved surface. It includes all possible element assignments. By following the instructions for the feature you want to create, it is possible to remove element settings not appropriate for your use.

```

/*=====*\
FILE      : UgSktRevolveSurface.c
PURPOSE   : Includes all possible element assignments for revolved features.
\*=====*/

/*----- Pro/Toolkit Includes -----*/
#include "ProToolkit.h"
#include "ProFeature.h"
#include "ProElemId.h"
#include "ProExtrude.h"
#include "ProModFeat.h"
#include "ProStdSection.h"
#include "ProElement.h"
#include "ProElemPath.h"
#include "ProFeatType.h"
#include "ProFeatForm.h"
#include "ProSelection.h"
#include "ProSection.h"

#include "ProRevolve.h"

static ProFileName message_file;

#define C_LOG(a) ProTKPrintf ("Status for %s is = %d\n", a, status ); \
ERROR_CHECK( a, "UgSktRevolveCreate.c", status );

#define C_PRINT(a) ProTKPrintf ( "%s\n", a);

/*=====*\
FUNCTION  : UserSktRevolveSurface()
PURPOSE   : Template function for creation of extrude features.
\*=====*/
ProError UserSktRevolveSurface()
{
    ProErrorlist      errors;

```

```

ProMdl          model;
ProModelitem    model_item;
ProSelection    model_sel;
ProFeature      feature;
ProFeatureCreateOptions opts[1];
ProElempath     path;
ProElempathItem path_items[2];
ProSection      section;
ProAsmcomppath  comp_path;
ProAsmcomppath  *p_comp_path = NULL;
ProValue        value;

ProElement sketch_element;
ProElement created_elemtree;

ProElement pro_e_feature_tree;
ProElement pro_e_feature_form;
ProElement pro_e_ext_surf_cut_solid_type;
ProElement pro_e_remove_material;
ProElement pro_e_feat_form_is_thin;
ProElement pro_e_srf_end_attributes;
ProElement pro_e_std_direction;

ProElement pro_e_rev_angle;
ProElement pro_e_rev_angle_from;
ProElement pro_e_rev_angle_from_type;
ProElement pro_e_rev_angle_to;
ProElement pro_e_rev_angle_to_type;
ProElement pro_e_rev_angle_to_ref;
int to_ref_id, to_ref_type;
ProModelitem to_ref_item;

ProElement pro_e_std_section;
ProElement pro_e_std_sec_method;
ProElement pro_e_std_sec_setup_plane;
ProElement pro_e_std_sec_plane;
ProElement pro_e_std_sec_plane_view_dir;
ProElement pro_e_std_sec_plane_orient_dir;
ProElement pro_e_std_sec_plane_orient_ref;

ProSelection *sketch_refs;
ProError status;
ProValueData value_data;
ProSelection * p_select;
int n_select;

ProStringToWstring ( message_file, "utilities.txt" );

log_file = PTApplsUnicodeFopen ( "ug_sketched_curve.log", "w" );

```



```

status = ProMdlCurrentGet (&model);
if ( status != PRO_TK_NO_ERROR ) return ( status );

/*-----*\
    Populating root element PRO_E_FEATURE_TREE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TREE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TREE, &pro_e_feature_tree );
C_LOG( " ProElementAlloc " );

/*-----*\
    Populating element PRO_E_FEATURE_FORM
\*-----*/

C_PRINT( " *** Processing Element PRO_E_FEATURE_FORM *** " );
status = ProElementAlloc ( PRO_E_FEATURE_FORM, &pro_e_feature_form );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REVOLVE;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_feature_form, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_feature_form );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_EXT_SURF_CUT_SOLID_TYPE
\*-----*/

C_PRINT( " *** Processing Element PRO_E_EXT_SURF_CUT_SOLID_TYPE *** " );
status = ProElementAlloc ( PRO_E_EXT_SURF_CUT_SOLID_TYPE,
                          &pro_e_ext_surf_cut_solid_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_FEAT_TYPE_SURFACE;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_ext_surf_cut_solid_type, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_ext_surf_cut_solid_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\

```

```

    Populating element PRO_E_REMOVE_MATERIAL
/*-----*/

C_PRINT( " *** Processing Element PRO_E_REMOVE_MATERIAL *** " );
status = ProElementAlloc ( PRO_E_REMOVE_MATERIAL, &pro_e_remove_material
);
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_MATERIAL_ADD;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_remove_material, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_remove_material );
C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Populating element PRO_E_FEAT_FORM_IS_THIN
/*-----*/

C_PRINT( " *** Processing Element PRO_E_FEAT_FORM_IS_THIN *** " );
status = ProElementAlloc ( PRO_E_FEAT_FORM_IS_THIN,
                            &pro_e_feat_form_is_thin );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_FEAT_FORM_NO_THIN;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_feat_form_is_thin, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_feat_form_is_thin );
C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Populating compound element PRO_E_STD_REV_ANGLE
/*-----*/
C_PRINT( " *** Processing Element PRO_E_REV_ANGLE *** " );
status = ProElementAlloc ( PRO_E_REV_ANGLE, &pro_e_rev_angle );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_rev_angle );
C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Populating element PRO_E_REV_ANGLE

```

```

-> PRO_E_REV_ANGLE_FROM
  \*-----*/
  C_PRINT( " *** Processing Element PRO_E_REV_ANGLE_FROM *** " );
  status = ProElementAlloc ( PRO_E_REV_ANGLE_FROM, &pro_e_rev_angle_from
);
  C_LOG( " ProElementAlloc" );
  status = ProElemtreeElementAdd ( pro_e_rev_angle, NULL,
                                pro_e_rev_angle_from );
  C_LOG( " ProElemtreeElementAdd" );

  \*-----*/
  Populating element PRO_E_REV_ANGLE
-> PRO_E_REV_ANGLE_FROM
  -> PRO_E_REV_ANGLE_FROM_TYPE
  \*-----*/
  C_PRINT( " *** Processing Element PRO_E_REV_ANGLE_FROM_TYPE *** " );
  status = ProElementAlloc ( PRO_E_REV_ANGLE_FROM_TYPE,
                            &pro_e_rev_angle_from_type );
  C_LOG( " ProElementAlloc " );
  value_data.type = PRO_VALUE_TYPE_INT;
  value_data.v.i = PRO_REV_ANG_FROM_NONE;
  status = ProValueAlloc ( &value );
  C_LOG( " ProValueAlloc" );
  status = ProValueDataSet ( value, &value_data );
  C_LOG( " ProValueDataSet" );
  status = ProElementValueSet ( pro_e_rev_angle_from_type, value );
  C_LOG( " ProElementValueSet" );
  status = ProElemtreeElementAdd ( pro_e_rev_angle_from, NULL,
                                pro_e_rev_angle_from_type );
  C_LOG( " ProElemtreeElementAdd" );

  \*-----*/
  Populating element PRO_E_REV_ANGLE
-> PRO_E_REV_ANGLE_TO
  \*-----*/
  C_PRINT( " *** Processing Element PRO_E_REV_ANGLE_TO *** " );
  status = ProElementAlloc ( PRO_E_REV_ANGLE_TO, &pro_e_rev_angle_to );
  C_LOG( " ProElementAlloc" );
  status = ProElemtreeElementAdd ( pro_e_rev_angle, NULL,
                                pro_e_rev_angle_to );
  C_LOG( " ProElemtreeElementAdd" );

  \*-----*/
  Populating element PRO_E_REV_ANGLE
-> PRO_E_REV_ANGLE_TO
  -> PRO_E_REV_ANGLE_TO_TYPE
  \*-----*/
  C_PRINT( " *** Processing Element PRO_E_REV_ANGLE_TO_TYPE *** " );
  status = ProElementAlloc ( PRO_E_REV_ANGLE_TO_TYPE,
                            &pro_e_rev_angle_to_type );
  C_LOG( " ProElementAlloc " );

```

```

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_ANG_TO_REF;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_rev_angle_to_type, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_rev_angle_to, NULL,
                                pro_e_rev_angle_to_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_REV_ANGLE
-> PRO_E_REV_ANGLE_TO
   -> PRO_E_REV_ANGLE_TO_REF
\*-----*/
C_PRINT( " *** Processing Element PRO_E_REV_ANGLE_TO_REF *** " );
status = ProElementAlloc ( PRO_E_REV_ANGLE_TO_REF,
                          &pro_e_rev_angle_to_ref );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_SELECTION;

/* PRO_SURFACE(plane), PRO_POINT, PRO_EDGE_START, PRO_EDGE_END,
   PRO_CRV_START, PRO_CRV_END */

if (UserUtilItemSelect ( "surf,datum,point,edge_end,curve_end", "USER
                        Select reference.", &to_ref_id, (ProType *)
                        &to_ref_type) != PRO_TK_NO_ERROR)
return PRO_TK_GENERAL_ERROR;
status = ProModelitemInit ( model, to_ref_id, to_ref_type,
                          &to_ref_item);
C_LOG ( " ProModelitemInit" );

status = ProSelectionAlloc (NULL, &to_ref_item, &value_data.v.r);
C_LOG ( " ProSelectionAlloc" );

status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_rev_angle_to_ref, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_rev_angle_to, NULL,
                                pro_e_rev_angle_to_ref );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_SECTION
\*-----*/

```

```

C_PRINT( " *** Processing Element PRO_E_STD_SECTION *** " );
status = ProElementAlloc ( PRO_E_STD_SECTION, &pro_e_std_section );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_std_section );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SEC_SETUP_PLANE *** " );
status = ProElementAlloc ( PRO_E_STD_SEC_SETUP_PLANE,
&pro_e_std_sec_setup_plane );
C_LOG( " ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_std_section, NULL,
pro_e_std_sec_setup_plane );
C_LOG( " ProElemtreeElementAdd" );

sketch_refs = ( ProSelection *) calloc ( 2, sizeof ( ProSelection ) );

/*-----*\
    Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    -> PRO_E_STD_SEC_PLANE
\*-----*/

C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE *** " );
status = ProMessageDisplay ( message_file, "Select Surface for sketch
    placement");
C_LOG( " ProMessageDisplay" );
ProTKPrintf ( "Please select datum,surface,sldface,qltface_ID_5 type of
    Modelitem\n");
status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL, NULL,
    NULL, NULL, &p_select, &n_select );
C_LOG( " ProSelect" );
if ( n_select <= 0 ) return -1;
else
{
    status = ProSelectionCopy( p_select[0], &sketch_refs[0]);
    C_LOG ( " ProSelectionCopy " );
}
status = ProElementAlloc ( PRO_E_STD_SEC_PLANE, &pro_e_std_sec_plane );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_sec_plane, value );

```

```

    C_LOG( " ProElementValueSet" );
    status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane );
    C_LOG( " ProElemtreeElementAdd" );

    /*-----*\
        Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    -> PRO_E_STD_SEC_PLANE_VIEW_DIR
    \*-----*/
    C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_VIEW_DIR *** " );
    status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_VIEW_DIR,
&pro_e_std_sec_plane_view_dir );
    C_LOG( " ProElementAlloc " );
    value_data.type = PRO_VALUE_TYPE_INT;
    value_data.v.i = PRO_SEC_VIEW_DIR_SIDE_ONE /* PRO_SEC_VIEW_DIR_SIDE_TWO
*/;
    status = ProValueAlloc ( &value );
    C_LOG( " ProValueAlloc" );
    status = ProValueDataSet ( value, &value_data );
    C_LOG( " ProValueDataSet" );
    status = ProElementValueSet ( pro_e_std_sec_plane_view_dir, value );
    C_LOG( " ProElementValueSet" );
    status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_view_dir );
    C_LOG( " ProElemtreeElementAdd" );

    /*-----*\
        Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    -> PRO_E_STD_SEC_PLANE_ORIENT_DIR
    \*-----*/
    C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_ORIENT_DIR *** "
);
    status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_ORIENT_DIR,
&pro_e_std_sec_plane_orient_dir );
    C_LOG( " ProElementAlloc " );
    value_data.type = PRO_VALUE_TYPE_INT;
    value_data.v.i = PRO_SEC_ORIENT_DIR_UP /* PRO_SEC_ORIENT_DIR_DOWN */
/* PRO_SEC_ORIENT_DIR_LEFT */ /* PRO_SEC_ORIENT_DIR_RIGHT */ ;
    status = ProValueAlloc ( &value );
    C_LOG( " ProValueAlloc" );
    status = ProValueDataSet ( value, &value_data );
    C_LOG( " ProValueDataSet" );
    status = ProElementValueSet ( pro_e_std_sec_plane_orient_dir, value );
    C_LOG( " ProElementValueSet" );
    status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_orient_dir );
    C_LOG( " ProElemtreeElementAdd" );

    /*-----*\

```

```

    Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
    -> PRO_E_STD_SEC_PLANE_ORIENT_REF
    /*-----*/
    C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_ORIENT_REF *** "
);
    status = ProMessageDisplay ( message_file,
"Select Surface for sketch orientation");
    C_LOG( " ProMessageDisplay" );
    ProTKPrintf ( "Select datum,surface,sldface,qltface_ID_5 type of
        Modelitem\n");
    status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL,
        NULL, NULL, NULL, &p_select, &n_select );
    C_LOG( " ProSelect" );
    if ( n_select <= 0 ) return -1;
    else
    {
        status = ProSelectionCopy( p_select[0], &sketch_refs[1]);
        C_LOG ( " ProSelectionCopy ");
    }

    status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_ORIENT_REF,
&pro_e_std_sec_plane_orient_ref );
    C_LOG( " ProElementAlloc " );
    value_data.type = PRO_VALUE_TYPE_SELECTION;
    value_data.v.r = p_select[0];
    status = ProValueAlloc ( &value );
    C_LOG( " ProValueAlloc" );
    status = ProValueDataSet ( value, &value_data );
    C_LOG( " ProValueDataSet" );
    status = ProElementValueSet ( pro_e_std_sec_plane_orient_ref, value );
    C_LOG( " ProElementValueSet" );
    status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_orient_ref );
    C_LOG( " ProElemtreeElementAdd" );

    /*-----*\
    Creating incomplete feature in the current model.
    /*-----*/
    status = ProMdlToModelitem( model, &model_item );
    status = ProSelectionAlloc ( p_comp_path, &model_item,
        &model_sel);

    opts[0] = PRO_FEAT_CR_INCOMPLETE_FEAT;
    status = ProFeatureCreate (model_sel, pro_e_feature_tree, opts, 1,
&feature, &errors);
    C_LOG ( " ProFeatureCreate");

    /* Using the element tree from created feature */
    status = ProFeatureElemtreeCreate ( &feature, &created_elemtree );
    C_LOG ( " ProFeatureElemtreeCreate");

```

```

/*-----*\
    Getting the initialized section element from the database.
\*-----*/
/* path to PRO_E_SKETCHER element */
path_items[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[0].path_item.elem_id = PRO_E_STD_SECTION;
path_items[1].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[1].path_item.elem_id = PRO_E_SKETCHER;
status = ProElempathAlloc (&path);
status = ProElempathDataSet (path, path_items, 2);

status = ProElemtreeElementGet ( created_elemtree, path,
                                &sketch_element);
C_LOG (" ProElemtreeElementGet PRO_E_SKETCHER ");

status = ProElementValueGet ( sketch_element, &value);
C_LOG (" ProElementValueGet PRO_E_SKETCHER ");

status = ProValueDataGet (value, &value_data);
C_LOG (" ProValueDataGet PRO_E_SKETCHER ");
section = (ProSection)value_data.v.p;

/*-----*\
    Creating a 3-D section
\*-----*/

status = UserSectionBuild (( ProSection ) (value_data.v.p),
    sketch_refs );
C_LOG ("UserSectionBuild");

ProElempathFree (&path);

/*-----*\
    Populating element PRO_E_STD_DIRECTION (must be done once section is
set)
\*-----*/

C_PRINT( " *** Processing Element PRO_E_STD_DIRECTION *** " );
path_items[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[0].path_item.elem_id = PRO_E_STD_DIRECTION;

status = ProElempathAlloc (&path);
status = ProElempathDataSet (path, path_items, 1);

status = ProElemtreeElementGet ( created_elemtree, path,
                                &pro_e_std_direction);
C_LOG (" ProElemtreeElementGet PRO_E_SKETCHER ");

status = ProElementValueGet ( pro_e_std_direction, &value );
C_LOG( " ProElementValueGet" );

```



```

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_CR_IN_SIDE_ONE;
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_direction, value );
C_LOG( " ProElementValueSet" );

ProElempathFree (&path);

/*-----*\
    Populating element PRO_E_SRF_END_ATTRIBUTES (surface features only -
    must be done once section is set)
\*-----*/

C_PRINT( " *** Processing Element PRO_E_SRF_END_ATTRIBUTES *** " );
path_items[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[0].path_item.elem_id = PRO_E_SRF_END_ATTRIBUTES;

status = ProElempathAlloc (&path);
status = ProElempathDataSet (path, path_items, 1);

status = ProElemtreeElementGet ( created_elemtree, path,
                                &pro_e_srf_end_attributes);
C_LOG ( " ProElemtreeElementGet PRO_E_SRF_END_ATTRIBUTES " );

status = ProElementValueGet ( pro_e_srf_end_attributes, &value );
C_LOG( " ProElementValueGet" );

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_SURF_END_ATTR_OPEN;
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_srf_end_attributes, value );
C_LOG( " ProElementValueSet" );

ProElempathFree (&path);

/*-----*\
    Redefining the feature to make it complete.
\*-----*/

opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMS;
status = ProSelectionAsmcomppathGet (model_sel, &comp_path);
C_LOG ( " ProSelectionAsmcomppathGet" );

status = ProFeatureRedefine (&comp_path, &feature, created_elemtree,
                             opts, 1, &errors);
C_LOG ( " ProFeatureRedefine" );

/*-----*\
    Free up the allocated memory.
\*-----*/

```

```

\*-----*/
status = ProFeatureElemtreeFree (&feature, created_elemtree );
C_LOG (" ProFeatureElemtreeFree");

status = ProElementFree (&pro_e_feature_tree );
C_LOG (" ProElementFree");

free(sketch_refs);

return (status);
}

```

The Element Tree for First Features

First features (extrude and revolve solids created as the first feature in a part) require a subset of the standard element tree and some special handling for the section pointer.

The following table lists the elements applicable to first feature creation (extrude or revolve):

Element ID	Value
PRO_E_FEATURE_TYPE	Feature type: PRO_FEAT_FIRST
PRO_E_FEATURE_FORM	PRO_EXTRUDE / PRO_REVOLVE
PRO_E_FEAT_FORM_IS_THIN	Feature Form Of Type ProRevFeatForm = PRO_REV_FEAT_FORM_NO_THIN for a Solid feature = PRO_REV_FEAT_FORM_THIN for a Thin feature
PRO_E_SKETCHER	Sketcher pointer. Used because the standard section requires selected references not available in an empty model.
PRO_E_STD_MATRLSIDE*	Mandatory if thin Of type ProExtMatlSide (Extrude) Of type ProRevMatlSide (Revolve)
PRO_E_THICKNESS	Mandatory >= 0.0 if thin Of type PRO_VALUE_TYPE_DOUBLE
PRO_E_EXT_DEPTH_FROM	Compound Element (Extrude only)
PRO_E_EXT_DEPTH_FROM_VAL	Depth dimension (of type PRO_VALUE_TYPE_DOUBLE) (Extrude only)

Element ID	Value
PRO_E_REV_ANGLE_FROM	Compound Element (Revolve only)
PRO_E_REV_ANGLE_FROM_VAL	Angular dimension (of type PRO_VALUE_TYPE_DOUBLE) (Revolve only)
PRO_E_STD_FEATURE_NAME	Default given by application depending on the feature type. Can be modified by the user.

Elements identified with “*” depend on the definition of the standard section. These elements may not be assigned values until the standard section has been completely allocated (which typically happens during redefine of the feature). Values assigned to these elements while the section is not complete are ignored.

Example 9: Creating the First Extruded Protrusion Feature by Conventional Approach

This example code shows how to create the first extruded protrusion using the approach for the sketched features.

The following example:

- Creates an incomplete feature using **ProFeatureCreate()**,
- Extracts the section from the element tree of the incomplete feature,
- Builds the section on the section handle obtained, and,
- Completes the feature using **ProFeatureRedefine()**.

Following is the change in the approach for Pro/ENGINEER Wildfire release:

1. Level of PRO_E_SKETCHER in an element tree is changed.

For any Pro/ENGINEER release previous to Wildfire:

```
PRO_E_FEATURE_TREE -> PRO_E_STD_SECTION -> PRO_E_SKETCHER
```

For Pro/ENGINEER Wildfire release:

```
PRO_E_FEATURE_TREE -> PRO_E_SKETCHER
```

2. Value of PRO_E_SKETCHER—A new ProValue is to be allocated and then assigned to the element (rather than the old approach of reusing the value extracted from the element tree).

```

/*=====*\
FILE      : UgSktFirstFeatureCreate.c
PURPOSE   : Creating the First Extruded Protrusion Feature by conventional
            approach (ProFeatureCreate -> incomplete feature
            Extract Section handle -> Build the section
            ProFeatureRedefine -> complete feature)
\*=====*/
/*----- Pro/Toolkit Includes -----*/
#include "ProToolkit.h"
#include "ProFeature.h"
#include "ProElemId.h"
#include "ProExtrude.h"
#include "ProModFeat.h"
#include "ProStdSection.h"
#include "ProElement.h"
#include "ProElempath.h"
#include "ProFeatType.h"
#include "ProFeatForm.h"
#include "ProSelection.h"
#include "ProSection.h"

#include "ProExtrude.h"

static ProFileName message_file;

#define C_LOG(a) ProTKPrintf ("Status for %s is = %d\n", a, status ); \
ERROR_CHECK( a, "UgSktFirstFeatureCreate.c", status );

#define C_PRINT(a) ProTKPrintf ( "%s\n", a);

ProError UserSectionFirstFeatureBuild ( ProSection * section );

/*=====*\
FUNCTION   : UgSktFirstFeatureCreate
PURPOSE    : Demonstrates the creation of the first extruded protrusion
            feature
\*=====*/
ProError UgSktFirstFeatureCreate()
{
    ProErrorlist      errors;
    ProMdl            model;
    ProModelitem      model_item;
    ProSelection       model_sel;
    ProFeature         feature;
    ProFeatureCreateOptions opts[1];
    ProElempath        path;
    ProElempathItem    path_items[2];

```

```

ProSection          section;
ProAsmcomppath      comp_path;
ProAsmcomppath      *p_comp_path = NULL;
ProValue            value;
ProValue            new_value;

ProElement sketch_element;
ProElement created_elemtree;

ProElement pro_e_feature_tree;
ProElement pro_e_feature_type;
ProElement pro_e_feature_form;

ProElement pro_e_feat_form_is_thin;
ProElement pro_e_thickness;

ProElement pro_e_std_ext_depth;
ProElement pro_e_ext_depth_from;
ProElement pro_e_ext_depth_from_type;
ProElement pro_e_ext_depth_from_value;

ProElement pro_e_ext_depth_to;
ProElement pro_e_ext_depth_to_type;
ProElement pro_e_ext_depth_to_value;

ProElement pro_e_sketcher;

ProError status;
ProValueData value_data;
ProSelection * p_select;
int n_select;

ProStringToWstring ( message_file, "utilities.txt" );

log_file = PTApplsUnicodeFopen ( "ug_sketched_curve.log", "w" );

/*-----*\
   Populating root element PRO_E_FEATURE_TREE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TREE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TREE, &pro_e_feature_tree );
C_LOG( " ProElementAlloc " );

/*-----*\
   Populating element PRO_E_FEATURE_TYPE
\*-----*/

C_PRINT( " *** Processing Element PRO_E_FEATURE_TYPE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TYPE, &pro_e_feature_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;

```

```

value_data.v.i = PRO_FEAT_FIRST_FEAT;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_feature_type, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_feature_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
   Populating element PRO_E_FEATURE_FORM
\*-----*/

C_PRINT( " *** Processing Element PRO_E_FEATURE_FORM *** " );
status = ProElementAlloc ( PRO_E_FEATURE_FORM, &pro_e_feature_form );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXTRUDE;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_feature_form, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_feature_form );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
   Populating compound element PRO_E_STD_EXT_DEPTH
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_EXT_DEPTH *** " );
status = ProElementAlloc ( PRO_E_STD_EXT_DEPTH, &pro_e_std_ext_depth );
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_std_ext_depth );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
   Populating element PRO_E_STD_EXT_DEPTH
-> PRO_E_EXT_DEPTH_FROM
\*-----*/
C_PRINT( " *** Processing Element PRO_E_EXT_DEPTH_FROM *** " );
status = ProElementAlloc ( PRO_E_EXT_DEPTH_FROM, &pro_e_ext_depth_from
);
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_std_ext_depth, NULL,
                                pro_e_ext_depth_from );
C_LOG( " ProElemtreeElementAdd" );

```

```

/*-----*\
    Populating element PRO_E_STD_EXT_DEPTH
        -> PRO_E_EXT_DEPTH_FROM
        -> PRO_E_EXT_DEPTH_FROM_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_EXT_DEPTH_FROM_TYPE *** " );
status = ProElementAlloc ( PRO_E_EXT_DEPTH_FROM_TYPE,
                          &pro_e_ext_depth_from_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_DEPTH_FROM_BLIND;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_ext_depth_from_type, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_ext_depth_from, NULL,
                                pro_e_ext_depth_from_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_EXT_DEPTH
-> PRO_E_EXT_DEPTH_FROM
    -> PRO_E_EXT_DEPTH_FROM_VALUE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_EXT_DEPTH_FROM_VALUE *** " );
status = ProElementAlloc ( PRO_E_EXT_DEPTH_FROM_VALUE,
                          &pro_e_ext_depth_from_value );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 120.000000;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_ext_depth_from_value, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_ext_depth_from, NULL,
                                pro_e_ext_depth_from_value );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_EXT_DEPTH
        -> PRO_E_EXT_DEPTH_TO
\*-----*/
C_PRINT( " *** Processing Element PRO_E_EXT_DEPTH_TO *** " );
status = ProElementAlloc ( PRO_E_EXT_DEPTH_TO, &pro_e_ext_depth_to );
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_std_ext_depth, NULL,

```

```

                                pro_e_ext_depth_to );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_EXT_DEPTH
                                -> PRO_E_EXT_DEPTH_TO
                                -> PRO_E_EXT_DEPTH_TO_TYPE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_EXT_DEPTH_TO_TYPE *** " );
status = ProElementAlloc ( PRO_E_EXT_DEPTH_TO_TYPE,
                            &pro_e_ext_depth_to_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_DEPTH_TO_BLIND;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_ext_depth_to_type, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_ext_depth_to, NULL,
                                pro_e_ext_depth_to_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_STD_EXT_DEPTH
                                -> PRO_E_EXT_DEPTH_TO
                                -> PRO_E_EXT_DEPTH_TO_VALUE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_EXT_DEPTH_TO_VALUE *** " );
status = ProElementAlloc ( PRO_E_EXT_DEPTH_TO_VALUE,
                            &pro_e_ext_depth_to_value );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 0.0;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_ext_depth_to_value, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_ext_depth_to, NULL,
                                pro_e_ext_depth_to_value );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Creating incomplete feature in the current model.
\*-----*/
status = ProMdlCurrentGet (&model);
if ( status != PRO_TK_NO_ERROR ) return ( status );
status = ProMdlToModelitem( model, &model_item );

```



```

status = ProSelectionAlloc (p_comp_path, &model_item,
    &model_sel);

opts[0] = PRO_FEAT_CR_INCOMPLETE_FEAT;
status = ProFeatureCreate (model_sel, pro_e_feature_tree, opts, 1,
    &feature, &errors);
C_LOG (" ProFeatureCreate");

/* Using the element tree from created feature */
status = ProFeatureElemtreeCreate ( &feature, &created_elemtree );
C_LOG (" ProFeatureElemtreeCreate");

/*-----*\
    Getting the initialized section element from the database.
\*-----*/
/* path to PRO_E_SKETCHER element */
path_items[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[0].path_item.elem_id = PRO_E_SKETCHER;

status = ProElemppathAlloc (&path);
status = ProElemppathDataSet (path, path_items, 1);

status = ProElemtreeElementGet ( created_elemtree, path,
    &sketch_element);
C_LOG (" ProElemtreeElementGet PRO_E_SKETCHER ");

status = ProElementValueGet ( sketch_element, &value);
C_LOG (" ProElementValueGet PRO_E_SKETCHER ");

status = ProValueDataGet (value, &value_data);
C_LOG (" ProValueDataGet PRO_E_SKETCHER ");

/*-----*\
    Creating a 3-D section
\*-----*/

status = UserSectionFirstFeatureBuild (( ProSection * )(&value_data.v.p)
);
C_LOG ("UserSectionFirstFeatureBuild");

/*-----*\
    Allocating and assigning new ProValue to sketcher element
    :: Wildfire requirement
\*-----*/

status = ProValueAlloc ( &new_value );
C_LOG (" ProValueAlloc ");

status = ProValueDataSet ( new_value, &value_data);
C_LOG (" ProValueDataSet ");

```

```

    status = ProElementValueSet ( sketch_element, new_value);
    C_LOG ( " ProElementValueSet PRO_E_SKETCHER ");

    ProElempathFree (&path);

/*-----*\
    Redefining the feature to make it complete.
\*-----*/
    opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMS;
    status = ProSelectionAsmcomppathGet (model_sel, &comp_path);
    C_LOG ( " ProSelectionAsmcomppathGet");

    status = ProFeatureRedefine (&comp_path, &feature, created_elemtree,
        opts, 1, &errors);
    C_LOG ( " ProFeatureRedefine");

/*-----*\
    Free up the allocated memory.
\*-----*/
    status = ProFeatureElemtreeFree (&feature, created_elemtree );
    C_LOG ( " ProFeatureElemtreeFree");

    status = ProElementFree (&pro_e_feature_tree );
    C_LOG ( " ProElementFree");

    return (status);
}

ProError UserSectionFirstFeatureBuild ( ProSection * section_here )
{
    Pro2dLinedef line;
    ProError status;
    int line_id[4];

    ProName section_name;

    ProWSecerror section_errors1;
    ProWSecerror section_errors2;
    ProWSecerror section_errors3;
    int n_sec;
    ProMsg error_message;
    char error_message_s[100];

    int i, error_id;

    status = ProSecerrorAlloc ( &section_errors1 );
    C_LOG ( " ProSecerrorAlloc");

    status = ProSecerrorAlloc ( &section_errors2 );
    C_LOG ( " ProSecerrorAlloc");

```

```

status = ProSecerrorAlloc ( &section_errors3 );
C_LOG ( " ProSecerrorAlloc");

status = ProSection2DAlloc ( section_here );
C_LOG ( " ProSection2DAlloc");

ProTKPrintf("The status for ProSection2DAlloc is = %d\n", status );

line.type = PRO_2D_LINE;
line.end1[0] = 0;
line.end1[1] = 0;
line.end2[0] = 100;
line.end2[1] = 0;

status = ProSectionEntityAdd ( *section_here, (Pro2dEntdef*)&line,
                               &line_id[0] );
C_LOG ( " ProSectionEntityAdd");

line.type = PRO_2D_LINE;
line.end1[0] = 100;
line.end1[1] = 0;
line.end2[0] = 100;
line.end2[1] = 100;

status = ProSectionEntityAdd ( *section_here, (Pro2dEntdef*)&line,
                               &line_id[1] );
C_LOG ( " ProSectionEntityAdd");

line.type = PRO_2D_LINE;
line.end1[0] = 100;
line.end1[1] = 100;
line.end2[0] = 0;
line.end2[1] = 100;

status = ProSectionEntityAdd ( *section_here, (Pro2dEntdef*)&line,
                               &line_id[2] );
C_LOG ( " ProSectionEntityAdd");

line.type = PRO_2D_LINE;
line.end1[0] = 0;
line.end1[1] = 100;
line.end2[0] = 0;
line.end2[1] = 0;

status = ProSectionEntityAdd ( *section_here, (Pro2dEntdef*)&line,
                               &line_id[3] );
C_LOG ( " ProSectionEntityAdd");

status = ProSectionEpsilonSet( *section_here, 0.1 );
C_LOG ( " ProSectionEpsilonSet(");

```

```

status = ProSectionAutodim ( *section_here, &section_errors1 );
C_LOG ( " ProSectionAutodim");

if ( status != PRO_TK_NO_ERROR )
{
    status = ProSecerrorCount ( &section_errors1, &n_sec );
    ProTKPrintf("The status for ProSecerrorCount is = %d\n", status );
    ProTKPrintf("Number of errors in the ProSectionAutodim = %d
                \n",n_sec );
    for ( i = 0; i < n_sec; i++ )
{
    status = ProSecerrorMsgGet (section_errors1, i, error_message);
    ProTKPrintf("The status for ProSecerrorMsgGet is = %d\n", status );
    ProWstringToString (error_message_s, error_message);
    status = ProSecerrorItemGet ( section_errors1, i, &error_id);
    ProTKPrintf("The status for ProSecerrorMsgGet is = %d\n", status );
    ProTKPrintf(" %s : Problem ID : %d \n", error_message_s, error_id );
}
}

status = ProSectionRegenerate ( *section_here, &section_errors3 );
C_LOG ( " ProSectionRegenerate");

if ( status != PRO_TK_NO_ERROR )
{
    status = ProSecerrorCount ( &section_errors3, &n_sec );
    ProTKPrintf("The status for ProSecerrorCount is = %d\n", status );
    ProTKPrintf("Number of errors in ProSectionRegenerate = %d \n",n_sec
);
}

return ( status );
}

```

Example 10: Creating the First Thin Revolve Protrusion Feature by Conventional Approach

This example code shows how to create a thin first revolve Protrusion using the approach for sketched features.

The following example:

- Creates an incomplete feature using **ProFeatureCreate()**,
- Extracts the section from the element tree of the incomplete feature,
- Builds the section on the section handle obtained, and,

- Completes the feature using **ProFeatureRedefine()**.

Following is the change in the approach for Pro/ENGINEER Wildfire release:

1. Level of PRO_E_SKETCHER in an element tree is changed.

For any Pro/ENGINEER release previous to Wildfire:

```
PRO_E_FEATURE_TREE -> PRO_E_STD_SECTION -> PRO_E_SKETCHER
```

For Pro/ENGINEER Wildfire release:

```
PRO_E_FEATURE_TREE -> PRO_E_SKETCHER
```

2. Value of PRO_E_SKETCHER—A new ProValue is to be allocated and then assigned to the element (rather than the old approach of reusing the value extracted from the element tree).

```
/*=====*\
FILE      : UgSktFirstFeatureRevCreate.c
PURPOSE   : Creating the First thin revolve Protrusion Feature by
conventional approach ( ProFeatureCreate -> incomplete feature
Extract Section handle -> Build the section ProFeatureRedefine ->
complete feature )
\*=====*/
/*----- Pro/Toolkit Includes -----*/
#include "ProToolkit.h"
#include "ProFeature.h"
#include "ProElemId.h"
#include "ProExtrude.h"
#include "ProModFeat.h"
#include "ProStdSection.h"
#include "ProElement.h"
#include "ProElempath.h"
#include "ProFeatType.h"
#include "ProFeatForm.h"
#include "ProSelection.h"
#include "ProSection.h"

#include "ProRevolve.h"

static ProFileName message_file;

#define C_LOG(a) ProTKPrintf ("Status for %s is = %d\n", a, status ); \
ERROR_CHECK( a, "UgSktFirstFeatureRevCreate.c", status );

#define C_PRINT(a) ProTKPrintf ( "%s\n", a);

ProError UserSectionFirstFeatureRevBuild ( ProSection * section );
```

```

/*=====*\
FUNCTION : UgSktFirstFeatureRevCreate
PURPOSE  : Demonstrates the creation of the first thin revolve protrusion
           feature
\*=====*/
ProError UgSktFirstFeatureRevCreate()
{
    ProErrorlist          errors;
    ProMdl                model;
    ProModelitem          model_item;
    ProSelection          model_sel;
    ProFeature            feature;
    ProFeatureCreateOptions opts[1];
    ProElempath           path;
    ProElempathItem       path_items[2];
    ProSection            section;
    ProAsmcomppath        comp_path;
    ProAsmcomppath        *p_comp_path = NULL;
    ProValue              value;
    ProValue              new_value;

    ProElement sketch_element;
    ProElement created_elemtree;

    ProElement pro_e_feature_tree;
    ProElement pro_e_feature_type;
    ProElement pro_e_feature_form;

    ProElement pro_e_feat_form_is_thin;
    ProElement pro_e_thickness;
    ProElement pro_e_std_matrlside;

    ProElement pro_e_rev_angle;
    ProElement pro_e_rev_angle_from;
    ProElement pro_e_rev_angle_from_type;
    ProElement pro_e_rev_angle_from_val;

    ProElement pro_e_sketcher;

    ProError status;
    ProValueData value_data;
    ProSelection * p_select;
    int n_select;

    ProStringToWstring ( message_file, "utilities.txt" );

    log_file = PTApplsUnicodeFopen ( "ug_sketched_curve.log", "w" );

/*-----*\
    Populating root element PRO_E_FEATURE_TREE

```

```

\*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TREE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TREE, &pro_e_feature_tree );
C_LOG( " ProElementAlloc " );

/*-----*\
    Populating element PRO_E_FEATURE_TYPE
\*-----*/

C_PRINT( " *** Processing Element PRO_E_FEATURE_TYPE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TYPE, &pro_e_feature_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_FEAT_FIRST_FEAT;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_feature_type, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_feature_type );
C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Populating element PRO_E_FEATURE_FORM
\*-----*/

C_PRINT( " *** Processing Element PRO_E_FEATURE_FORM *** " );
status = ProElementAlloc ( PRO_E_FEATURE_FORM, &pro_e_feature_form );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REVOLVE;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_feature_form, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_feature_form );
C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Populating element PRO_E_FEAT_FORM_IS_THIN
\*-----*/

C_PRINT( " *** Processing Element PRO_E_FEAT_FORM_IS_THIN *** " );
status = ProElementAlloc ( PRO_E_FEATURE_FORM, &pro_e_feat_form_is_thin
);
C_LOG( " ProElementAlloc " );

```

```

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_FEAT_FORM_THIN; /* 128 */
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_feat_form_is_thin, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_feat_form_is_thin );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
   Populating compound element PRO_E_REV_ANGLE
\*-----*/

C_PRINT( " *** Processing Element PRO_E_REV_ANGLE *** " );
status = ProElementAlloc ( PRO_E_REV_ANGLE, &pro_e_rev_angle );
C_LOG( " ProElementAlloc" );

status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_rev_angle );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
   Populating element PRO_E_REV_ANGLE
-> PRO_E_REV_ANGLE_FROM
\*-----*/

C_PRINT( " *** Processing Element PRO_E_REV_ANGLE_FROM *** " );
status = ProElementAlloc ( PRO_E_REV_ANGLE_FROM, &pro_e_rev_angle_from
);
C_LOG( " ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_rev_angle, NULL,
                                pro_e_rev_angle_from );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
   Populating element PRO_E_REV_ANGLE
-> PRO_E_REV_ANGLE_FROM
   -> PRO_E_REV_ANGLE_FROM_TYPE
\*-----*/

C_PRINT( " *** Processing Element PRO_E_REV_ANGLE_FROM_TYPE *** " );
status = ProElementAlloc ( PRO_E_REV_ANGLE_FROM_TYPE,
                          &pro_e_rev_angle_from_type );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_ANG_FROM_ANGLE;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );

```



```

status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_rev_angle_from_type, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_rev_angle_from, NULL,
                                pro_e_rev_angle_from_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_REV_ANGLE
-> PRO_E_REV_ANGLE_FROM
    -> PRO_E_REV_ANGLE_FROM_VAL
\*-----*/

C_PRINT( " *** Processing Element PRO_E_REV_ANGLE_FROM_VAL *** " );
status = ProElementAlloc ( PRO_E_REV_ANGLE_FROM_VAL,
                          &pro_e_rev_angle_from_val );
C_LOG( " ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 120.000000;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_rev_angle_from_val, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_rev_angle_from, NULL,
                                pro_e_rev_angle_from_val );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Creating incomplete feature in the current model.
\*-----*/

status = ProMdlCurrentGet (&model);
if ( status != PRO_TK_NO_ERROR ) return ( status );
status = ProMdlToModelitem( model, &model_item );
status = ProSelectionAlloc (p_comp_path, &model_item,
                            &model_sel);

opts[0] = PRO_FEAT_CR_INCOMPLETE_FEAT;
status = ProFeatureCreate (model_sel, pro_e_feature_tree, opts, 1,
                            &feature, &errors);
C_LOG ( " ProFeatureCreate");

/* Using the element tree from created feature */
status = ProFeatureElemtreeCreate ( &feature, &created_elemtree );
C_LOG ( " ProFeatureElemtreeCreate");

/*-----*\
    Getting the initialized section element from the database.

```

```

\*-----*/

/* path to PRO_E_SKETCHER element */
path_items[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[0].path_item.elem_id = PRO_E_SKETCHER;

status = ProElempathAlloc (&path);
status = ProElempathDataSet (path, path_items, 1);

status = ProElemtreeElementGet ( created_elemtree, path,
                                &sketch_element);
C_LOG (" ProElemtreeElementGet PRO_E_SKETCHER ");

status = ProElementValueGet ( sketch_element, &value);
C_LOG (" ProElementValueGet PRO_E_SKETCHER ");

status = ProValueDataGet (value, &value_data);
C_LOG (" ProValueDataGet PRO_E_SKETCHER ");

/*-----*\
    Creating a 3-D section
\*-----*/

status = UserSectionFirstFeatureRevBuild (( ProSection *
)(&value_data.v.p) );
C_LOG ("UserSectionFirstFeatureRevBuild");

/*-----*\
    Allocating and assigning new ProValue to sketcher element
    :: Wildfire requirement
\*-----*/

status = ProValueAlloc ( &new_value );
C_LOG (" ProValueAlloc ");

status = ProValueDataSet ( new_value, &value_data);
C_LOG (" ProValueDataSet ");

status = ProElementValueSet ( sketch_element, new_value);
C_LOG (" ProElementValueSet PRO_E_SKETCHER ");

ProElempathFree (&path);

/*-----*\
    Populating element PRO_E_STD_MATRLSIDE (must be done once section is
set)
\*-----*/

C_PRINT( " *** Processing Element PRO_E_STD_MATRLSIDE *** " );
path_items[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[0].path_item.elem_id = PRO_E_STD_MATRLSIDE;

```

```

status = ProElemPathAlloc (&path);
status = ProElemPathDataSet (path, path_items, 1);

status = ProElemtreeElementGet ( created_elemtree, path,
                                &pro_e_std_matrlside);

C_LOG (" ProElemtreeElementGet PRO_E_STD_MATRLSIDE ");

status = ProElementValueGet ( pro_e_std_matrlside, &value );
C_LOG( " ProElementValueGet" );

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_MATERIAL_SIDE_ONE;
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_matrlside, value );
C_LOG( " ProElementValueSet" );

ProElemPathFree (&path);

/*-----*\
   Populating element PRO_E_THICKNESS (must be done once section is set)
\*-----*/

C_PRINT( " *** Processing Element PRO_E_THICKNESS *** " );
path_items[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
path_items[0].path_item.elem_id = PRO_E_THICKNESS;

status = ProElemPathAlloc (&path);
status = ProElemPathDataSet (path, path_items, 1);

status = ProElemtreeElementGet ( created_elemtree, path,
                                &pro_e_thickness);
C_LOG (" ProElemtreeElementGet PRO_E_THICKNESS ");

status = ProElementValueGet ( pro_e_thickness, &value );
C_LOG( " ProElementValueGet" );

value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 10.0;
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_thickness, value );
C_LOG( " ProElementValueSet" );

ProElemPathFree (&path);

/*-----*\
   Redefining the feature to make it complete.
\*-----*/

```

```

    opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMS;
    status = ProSelectionAsmcomppathGet (model_sel, &comp_path);
    C_LOG (" ProSelectionAsmcomppathGet");

    status = ProFeatureRedefine (&comp_path, &feature, created_elemtree,
        opts, 1, &errors);
    C_LOG (" ProFeatureRedefine");

/*-----*\
    Free up the allocated memory.
\*-----*/
    status = ProFeatureElemtreeFree (&feature, created_elemtree );
    C_LOG (" ProFeatureElemtreeFree");

    status = ProElementFree (&pro_e_feature_tree );
    C_LOG (" ProElementFree");

    return (status);
}

ProError UserSectionFirstFeatureRevBuild ( ProSection * section_here )
{
    Pro2dLinedef line;
    ProError status;
    int c_line_id;
    int line_id[4];

    ProName section_name;

    ProWSecerror section_errors1;
    ProWSecerror section_errors2;
    ProWSecerror section_errors3;
    int n_sec;
    ProMsg error_message;
    char error_message_s[100];

    int i, error_id;

    status = ProSecerrorAlloc ( &section_errors1 );
    C_LOG (" ProSecerrorAlloc");

    status = ProSecerrorAlloc ( &section_errors2 );
    C_LOG (" ProSecerrorAlloc");

    status = ProSecerrorAlloc ( &section_errors3 );
    C_LOG (" ProSecerrorAlloc");

    status = ProSection2DAlloc ( section_here );
    C_LOG (" ProSection2DAlloc");

```

```

ProTKPrintf("The status for ProSection2DAlloc is = %d\n", status );

line.type = PRO_2D_CENTER_LINE;
line.end1[0] = 0;
line.end1[1] = 0;
line.end2[0] = 100;
line.end2[1] = 0;

status = ProSectionEntityAdd ( *section_here, (Pro2dEntdef*)&line,
                                &c_line_id );
C_LOG (" ProSectionEntityAdd");

status = ProSectionEntityConstructionSet( *section_here, c_line_id,
                                           PRO_B_FALSE );
C_LOG (" ProSectionEntityConstructionSet");

line.type = PRO_2D_LINE;
line.end1[0] = 0;
line.end1[1] = 0;
line.end2[0] = 100;
line.end2[1] = 0;

status = ProSectionEntityAdd ( *section_here, (Pro2dEntdef*)&line,
                                &line_id[0] );
C_LOG (" ProSectionEntityAdd");

line.type = PRO_2D_LINE;
line.end1[0] = 100;
line.end1[1] = 0;
line.end2[0] = 100;
line.end2[1] = 100;

status = ProSectionEntityAdd ( *section_here, (Pro2dEntdef*)&line,
                                &line_id[1] );
C_LOG (" ProSectionEntityAdd");

line.type = PRO_2D_LINE;
line.end1[0] = 100;
line.end1[1] = 100;
line.end2[0] = 0;
line.end2[1] = 100;

status = ProSectionEntityAdd ( *section_here, (Pro2dEntdef*)&line,
                                &line_id[2] );
C_LOG (" ProSectionEntityAdd");

line.type = PRO_2D_LINE;
line.end1[0] = 0;
line.end1[1] = 100;
line.end2[0] = 0;
line.end2[1] = 0;

```

```

status = ProSectionEntityAdd ( *section_here, (Pro2dEntdef*)&line,
                                &line_id[3] );
C_LOG (" ProSectionEntityAdd");

status = ProSectionEpsilonSet( *section_here, 0.1 );
C_LOG (" ProSectionEpsilonSet(");

status = ProSectionAutodim ( *section_here, &section_errors1 );
C_LOG (" ProSectionAutodim");

if ( status != PRO_TK_NO_ERROR )
{
    status = ProSecerrorCount ( &section_errors1, &n_sec );
    ProTKPrintf("The status for ProSecerrorCount is = %d\n", status );
    ProTKPrintf("Number of errors in the ProSectionAutodim = %d
                \n",n_sec );
    for ( i = 0; i < n_sec; i++ )
    {
        status = ProSecerrorMsgGet (section_errors1, i, error_message);
        ProTKPrintf("The status for ProSecerrorMsgGet is = %d\n", status );
        ProWstringToString (error_message_s, error_message);
        status = ProSecerrorItemGet ( section_errors1, i, &error_id);
        ProTKPrintf("The status for ProSecerrorMsgGet is = %d\n", status );
        ProTKPrintf(" %s : Problem ID : %d \n", error_message_s, error_id );
    }
}

status = ProSectionRegenerate ( *section_here, &section_errors3 );
C_LOG (" ProSectionRegenerate");

if ( status != PRO_TK_NO_ERROR )
{
    status = ProSecerrorCount ( &section_errors3, &n_sec );
    ProTKPrintf("The status for ProSecerrorCount is = %d\n", status );
    ProTKPrintf("Number of errors in ProSectionRegenerate = %d \n",n_sec
);
}

return ( status );
}

```

Element Trees: Basic Sweep

This chapter describes the basic principles of creating a swept feature. The chapters 'Element Trees: Principles of Feature Creation' and 'Element Trees: Sketched Features' provide the necessary background for this topic. Read those chapters before this one.

Topic	Page
Sweeps in Pro/ENGINEER Wildfire	43 - 2
Creating a Swept Feature	43 - 5

Sweeps in Pro/ENGINEER Wildfire

In Pro/ENGINEER Wildfire, the Sweep tool allows creation of complex variable section sweeps with many parameters. Pro/TOOLKIT does not provide access to the complete element tree for the features created using this Sweep tool. Pro/TOOLKIT access to sweeps is limited to simple sweeps and uses the same element tree provided in Release 2001.

Consider the following points while working with sweeps:

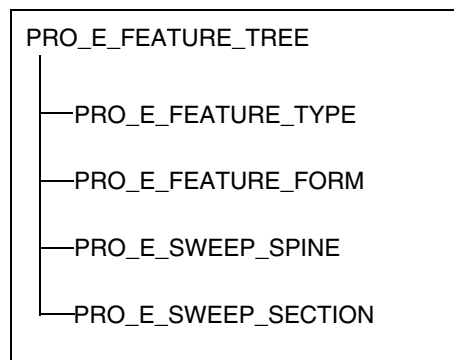
- Pro/TOOLKIT allows creation of sweeps using the Release 2001 element tree. These sweeps use the old Pro/ENGINEER interface for redefinition.
- This element tree also uses the old Release 2001 definition of PRO_E_STD_SECTION (the standard section element sub-tree). Thus, utility code used for extrude and revolve features' sections will not work for sweeps.
- To access and redefine simple sweeps, use the Release 2001 element tree.
- Pro/TOOLKIT may not currently create sweeps using the element tree used by the Pro/ENGINEER Wildfire Sweep tool.

The following descriptions and examples are relevant for the simple sweep element tree.

The Element Tree for Sweeps

The element tree for a sweep feature is documented in the header file *ProSweep.h*, and is shown in the following figure.

Figure 43-1: Element Tree for Sweep Feature



The following table describes the elements in the element tree for sweeps.

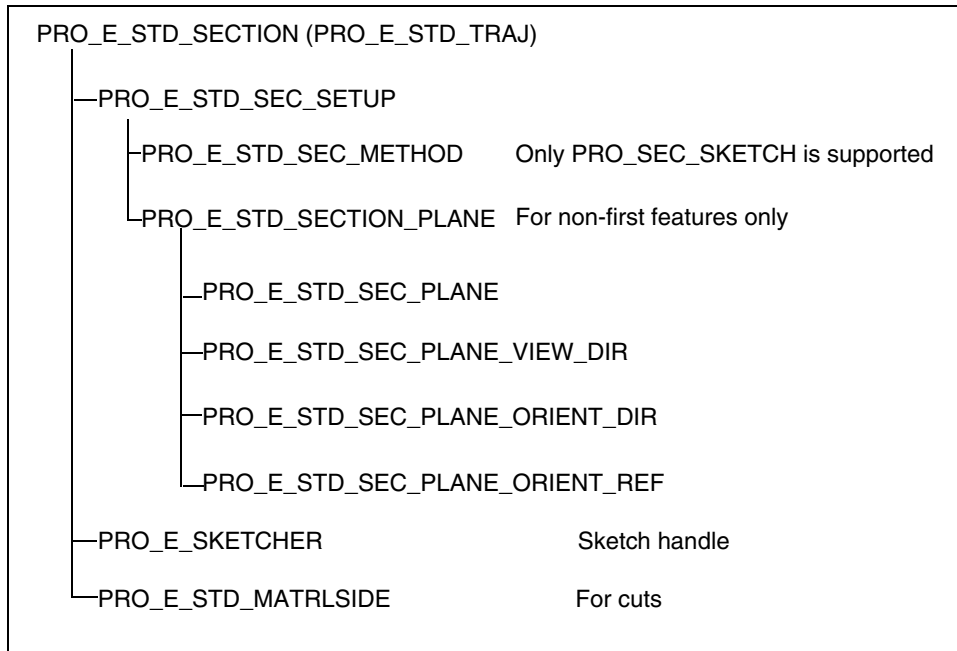
Element ID	Data Type	Description
PRO_E_FEATURE_TYPE	PRO_VALUE_TYPE_INT	Feature type
PRO_E_FEATURE_FORM	PRO_VALUE_TYPE_INT	Feature form (PRO_SWEEP)
PRO_E_SWEEP_SPINE	Compound	Trajectory (like PRO_E_STD_SECTION)
PRO_E_SWEEP_SECTION	Compound	Section (like PRO_E_STD_SECTION)

The element tree definitions of PRO_E_SWEEP_SPINE and PRO_E_SWEEP_SECTION take on the same form as the element PRO_E_STD_SECTION (documented in the header file *ProStdSection.h*.)

Note: Release 20 of Pro/TOOLKIT supports only sketched, constant cross-sectional sweeps.

The following figure shows the valid elements within this subtree.

Figure 43-2: Element Subtree for Sweep



Swept, constant, cross-sectional feature forms are supported for the following feature types:

- PRO_FEAT_FIRST_FEAT
- PRO_FEAT_PROTRUSION
- PRO_FEAT_CUT
- PRO_FEAT_DATUM_SURF

Creating a Swept Feature

To Create a Swept Feature

1. Create an incomplete feature with the PRO_E_FEATURE_TYPE and PRO_E_FEATURE_FORM elements defined. Also define the compound elements PRO_E_SWEEP_SPINE and PRO_E_SWEEP_SECTION, down to the PRO_E_STD_SEC_METHOD element (see the subtree for details).
2. Call **ProFeatureElemtreeExtract()** with the feature handle to get the new feature tree. This results in an initialized PRO_E_SWEEP_SPINE subtree and sketch handle.
3. Create the spine section with the initialized sketch handle.
4. Call **ProFeatureRedefine()** as an incomplete feature to put the spine section in the Pro/ENGINEER database.
5. Call **ProFeatureElemtreeExtract()** to get the new feature tree. This results in an initialized PRO_E_SWEEP_SECTION subtree and sketch handle. This step is necessary because the sweep section is dependent on the spine section.
6. Create the sweep section with the initialized sketch handle. This section automatically contains the centerline cross hairs of the sweep section.

The cross hairs can be used to locate and dimension the section.
7. Call **ProFeatureRedefine()** with any option except incomplete to complete the swept feature.

Example 1: Creating a Swept First Feature Protrusion

The following example shows how to create a swept first feature protrusion using the Pro/TOOLKIT functions.

```
/*----- Pro/Toolkit Includes -----*/
#include <ProToolkit.h>
#include <ProMdl.h>
#include <ProElement.h>
#include <ProFeatType.h>
#include <ProFeatForm.h>
#include <ProSweep.h>
#include <ProFeature.h>
#include <ProSection.h>
#include <ProSecdim.h>

/*----- Pro/Develop Includes -----*/
```

```

/*----- Application Includes -----*/
#include <TestError.h>

/*----- Application Typedefs -----*/
typedef struct element_data
{
    ProElement parent_element;
    ProElemId elem_id;
    ProValueData value_data;
} UgElemdata;

/*----- Prototypes -----*/
ProError UgFeatElemAdd(UgElemdata *elem, ProElement *added_elem);
ProError UgFeatElemValueGet(ProElement tree, ProElempathItem *path,
                             int size, ProValueData *elem_val);
ProError UgSectionRegenerate(ProSection section);
ProError UgSweepSpineSktCreate(ProSection section);
ProError UgSweepSectSktCreate(ProSection section);
ProError UgFeatElemRemove(ProElement tree, ProElempathItem *path,
                             int size);
ProError UgFeatElemAddSecMethod(ProElement std_sec_elem);

/*=====*\
Function: UgBaseSweepProtrCreate
Purpose: Example to demonstrate creation of first feature sweep
         protrusion
\*=====*/
int UgBaseSweepProtrCreate(ProPart part )
{
    ProError err;
    UgElemdata elem_data;
    ProFeature feature;
    ProErrorlist ftcrr_errs;
    ProElement elem, elem_tree;
    ProSelection part_sel;
    ProModelitem part_mdllitem;
    ProFeatureCreateOptions ftcropts[] = {PRO_FEAT_CR_INCOMPLETE_FEAT};
    ProElempathItem sect_path[] = {{PRO_ELEM_PATH_ITEM_TYPE_ID,
                                     PRO_E_SWEEP_SECTION}};
    ProElempathItem spine_skt_path[] = {{PRO_ELEM_PATH_ITEM_TYPE_ID,
                                           PRO_E_SWEEP_SPINE},
                                          {PRO_ELEM_PATH_ITEM_TYPE_ID,
                                           PRO_E_SKETCHER}};
    ProElempathItem sect_skt_path[] = {{PRO_ELEM_PATH_ITEM_TYPE_ID,
                                           PRO_E_SWEEP_SECTION},
                                          {PRO_ELEM_PATH_ITEM_TYPE_ID,
                                           PRO_E_SKETCHER}};
    ProValueData spine_skt_val, sect_skt_val;

    if (part == NULL)

```

```

    {
        err = ProMdlCurrentGet((ProMdl *)&part);
        ERROR_CHECK("ProMdlCurrentGet", "UgBaseSweepProtrCreate", err);
    }
/*-----*\
Initialize element tree
\*-----*/
    err = ProElementAlloc(PRO_E_FEATURE_TREE, &elem_tree);
    ERROR_CHECK("ProElementAlloc", "UgBaseSweepProtrCreate", err);

/*-----*\
Add feature type element to the tree
\*-----*/
    elem_data.parent_element = elem_tree;
    elem_data.elem_id = PRO_E_FEATURE_TYPE;
    elem_data.value_data.type = PRO_VALUE_TYPE_INT;
    elem_data.value_data.v.i = PRO_FEAT_FIRST_FEAT;
    err = UgFeatElemAdd(&elem_data, &elem);

/*-----*\
Add feature form element to the tree
\*-----*/
    elem_data.parent_element = elem_tree;
    elem_data.elem_id = PRO_E_FEATURE_FORM;
    elem_data.value_data.type = PRO_VALUE_TYPE_INT;
    elem_data.value_data.v.i = PRO_SWEEP;
    err = UgFeatElemAdd(&elem_data, &elem);

/*-----*\
Add section method for the spine section
\*-----*/
    elem_data.parent_element = elem_tree;
    elem_data.elem_id = PRO_E_SWEEP_SPINE;
    elem_data.value_data.type = -1;
    elem_data.value_data.v.i = 0;
    err = UgFeatElemAdd(&elem_data, &elem);
    err = UgFeatElemAddSecMethod(elem);

/*-----*\
Add section method for the sweep section
\*-----*/
    elem_data.parent_element = elem_tree;
    elem_data.elem_id = PRO_E_SWEEP_SECTION;
    elem_data.value_data.type = -1;
    elem_data.value_data.v.i = 0;
    err = UgFeatElemAdd(&elem_data, &elem);
    err = UgFeatElemAddSecMethod(elem);

/*-----*\
Create incomplete feature to initialize spine sketch handle
\*-----*/

```

```

err = ProMdlToModelitem((ProMdl)part,&part_mdllitem);
ERROR_CHECK("ProMdlToModelitem","UgBaseSweepProtrCreate",err);
err = ProSelectionAlloc(NULL,&part_mdllitem,&part_sel);
ERROR_CHECK("ProSelectionAlloc","UgBaseSweepProtrCreate",err);
err = ProFeatureCreate(part_sel,elem_tree,ftcropts,1,&feature,
                        &ftcr_errs);
ERROR_CHECK("ProFeatureCreate","UgBaseSweepProtrCreate",err);
err = ProElementFree(&elem_tree);
ERROR_CHECK("ProElementFree","UgBaseSweepProtrCreate",err);

/*-----*\
Retrieve initialized element tree from the proe data base
\*-----*/
err = ProFeatureElemtreeExtract (&feature,NULL,
                                PRO_FEAT_EXTRACT_NO_OPTS,
                                &elem_tree);
ERROR_CHECK ("ProFeatureElemtreeExtract()",
            "UgBaseSweepProtrCreate",err);

/*-----*\
Get the spine sketch handle from the element tree
\*-----*/
err = UgFeatElemValueGet(elem_tree,spine_skt_path,2,&spine_skt_val);

/*-----*\
Create sweep spine
\*-----*/
err = UgSweepSpineSktCreate((ProSection)spine_skt_val.v.p);

/*-----*\
Remove sweep section element tree to avoid uninitialized data
\*-----*/
err = UgFeatElemRemove(elem_tree,sect_path,1);

/*-----*\
Add section method for the sweep x-section
\*-----*/
elem_data.parent_element = elem_tree;
elem_data.elem_id = PRO_E_SWEEP_SECTION;
elem_data.value_data.type = -1;
elem_data.value_data.v.i = 0;
err = UgFeatElemAdd(&elem_data,&elem);
err = UgFeatElemAddSecMethod(elem);

/*-----*\
Redefine feature as incomplete to add spine and initialize section sketch
\*-----*/
err = ProFeatureRedefine(NULL,&feature,elem_tree,ftcropts,1,
                        &ftcr_errs);
ERROR_CHECK("ProFeatureRedefine","UgBaseSweepProtrCreate",err);
err = ProFeatureElemtreeFree(&feature, elem_tree);
ERROR_CHECK("ProFeatureElemtreeFree","UgBaseSweepProtrCreate",err);

```

```

/*-----*\
Retrieve initialized element tree from the proe data base
\*-----*/
    err = ProFeatureElemtreeExtract (&feature, NULL,
                                     PRO_FEAT_EXTRACT_NO_OPTS,
                                     &elem_tree);
    ERROR_CHECK ("ProFeatureElemtreeExtract",
                "UgBaseSweepProtrCreate",err);

/*-----*\
Get the section sketch handle from the element tree
\*-----*/
    err = UgFeatElemValueGet (elem_tree,sect_skt_path,2,&sect_skt_val);

/*-----*\
Create sweep section sketch
\*-----*/
    err = UgSweepSectSktCreate((ProSection)sect_skt_val.v.p);

/*-----*\
Redefine feature to create complete feature
\*-----*/
    ftcropts[0] = PRO_FEAT_CR_NO_OPTS;
    err = ProFeatureRedefine(NULL,&feature,elem_tree,ftcropts,1,
                             &ftcr_errs);
    ERROR_CHECK("ProFeatureRedefine","UgBaseSweepProtrCreate",err);
    err = ProFeatureElemtreeFree(&feature, elem_tree);
    ERROR_CHECK("ProFeatureElemtreeFree", "UgBaseSweepProtrCreate",err);

    return(err);
}
/*=====*\
Function: UgFeatElemAdd
Purpose: Adds a generic feature element to the element tree
\*=====*/
ProError UgFeatElemAdd(UgElemdata *elem,ProElement *added_elem)
{
    ProValue value;
    ProElement element;
    ProError err;

    if (elem->value_data.type != -1)
    {
        err = ProValueAlloc(&value);
        ERROR_CHECK("ProValueAlloc","UgFeatElemAdd", err);
        err = ProValueDataSet (value,&elem->value_data);
        ERROR_CHECK("ProValueDataSet", "UgFeatElemAdd",err);
    }

    err = ProElementAlloc(elem->elem_id,&element);

```

```

ERROR_CHECK("ProElementAlloc", "UgFeatElemAdd", err);
if (elem->value_data.type != -1)
{
    err = ProElementValueSet(element, value);
    ERROR_CHECK("ProElementValueSet", "UgFeatElemAdd", err);
}

err = ProElementtreeElementAdd(elem->parent_element, NULL, element);
ERROR_CHECK("ProElementtreeElementAdd", "UgFeatElemAdd", err);

*added_elem = element;

return(err);
}
/*=====*\
Function: UgFeatElemAddSecMethod
Purpose: adds necessary elements below PRO_E_STD_SECTION elem to
        specify sec method element
\*=====*/
ProError UgFeatElemAddSecMethod(ProElement std_sec_elem)
{
    ProElement elem;
    UgElemdata elem_data;
    ProError err;

    elem_data.parent_element = std_sec_elem;
    elem_data.elem_id = PRO_E_STD_SEC_SETUP;
    elem_data.value_data.type = -1;
    elem_data.value_data.v.i = 0;
    err = UgFeatElemAdd(&elem_data, &elem);

    elem_data.parent_element = elem;
    elem_data.elem_id = PRO_E_STD_SEC_METHOD;
    elem_data.value_data.type = PRO_VALUE_TYPE_INT;
    elem_data.value_data.v.i = PRO_SEC_SKETCH;
    err = UgFeatElemAdd(&elem_data, &elem);

    return(0);
}

/*=====*\
Function: UgFeatElemValueGet
Purpose: Gets a feature element from the element tree
\*=====*/
ProError UgFeatElemValueGet(ProElement tree, ProElempathItem *path,
                           int size, ProValueData *elem_val)
{
    ProElempath elem_path;
    ProElement elem;
    ProValue value;
    ProError err;

```



```

err = ProElempathAlloc(&elem_path);
ERROR_CHECK("ProElempathAlloc", "UgFeatElemGet", err);

err = ProElempathDataSet(elem_path, path, size);
ERROR_CHECK("ProElempathDataSet", "UgFeatElemGet", err);

err = ProElemtreeElementGet(tree, elem_path, &elem);
ERROR_CHECK("ProElemtreeElementGet", "UgFeatElemValueGet", err);

err = ProElementValueGet(elem, &value);
ERROR_CHECK("ProElementValueGet", "UgFeatElemValueGet", err);

err = ProValueDataGet(value, elem_val);
ERROR_CHECK("ProValueDataGet", "UgFeatElemValueGet", err);

return(err);
}

/*=====*\
Function: UgFeatElemRemove
Purpose: Removes a feature element from the element tree
\*=====*/
ProError UgFeatElemRemove(ProElement tree, ProElempathItem *path,
                          int size)
{
    ProElempath elem_path;
    ProElement elem;
    ProError err;

    err = ProElempathAlloc(&elem_path);
    ERROR_CHECK("ProElempathAlloc", "UgFeatElemGet", err);

    err = ProElempathDataSet(elem_path, path, size);
    ERROR_CHECK("ProElempathDataSet", "UgFeatElemGet", err);

    err = ProElemtreeElementGet(tree, elem_path, &elem);
    ERROR_CHECK("ProElemtreeElementGet", "UgFeatElemValueGet", err);

    err = ProElemtreeElementRemove(tree, elem_path, &elem);
    ERROR_CHECK("ProElemtreeElementRemove", "UgFeatElemRemove", err);

    err = ProElementFree(&elem);
    ERROR_CHECK("ProElementFree", "UgFeatElemRemove", err);

    return(err);
}

/*=====*\
Function: UgSweepSpineSktCreate
Purpose: Creates a spine sketch for a sweep first feature

```

```

\*=====*/
ProError UgSweepSpineSktCreate(ProSection section)
{
    Pro2dSplinedef spline;
    Pro2dPnt pnt_ar[] = {{0.0,0.0},{5.0,5.0},{10.0,-5.0},{15.0,2.0}};
    int spl_id, dim_id;
    Pro2dPnt place_pnt = {7.5,0.0};
    ProSectionPointType pnt_types[] = {PRO_ENT_START,PRO_ENT_END};
    int ent_ids[2];
    ProError err;

    spline.type = PRO_2D_SPLINE;
    spline.tangency_type = PRO_2D_SPLINE_TAN_NONE;
    spline.n_points = 4;
    spline.point_arr = pnt_ar;
    spline.start_tang_angle = 0.0;
    spline.end_tang_angle = 0.0;

    err = ProSectionEntityAdd(section, (Pro2dEntdef *)&spline,&spl_id);
    ERROR_CHECK("ProSecitonEntityAdd", "UgSweepSpineSktCreate",err);

    ent_ids[0] = ent_ids[1] = spl_id;
    err = ProSecdimCreate(section,ent_ids,pnt_types,2,
        PRO_TK_DIM_PNT_PNT_HORIZ,place_pnt,&dim_id);
    ERROR_CHECK("ProSecdimCreate", "UgSweepSpineSktCreate",err);

    ent_ids[0] = ent_ids[1] = spl_id;
    err = ProSecdimCreate(section,ent_ids,pnt_types,2,
        PRO_TK_DIM_PNT_PNT_VERT,place_pnt,&dim_id);
    ERROR_CHECK("ProSecdimCreate", "UgSweepSpineSktCreate",err);

    err = UgSectionRegenerate(section);

    return(err);
}

\*=====*/
Function: UgSweepSectSktCreate
Purpose: Creates a section sketch for a constant section sweep
\*=====*/
ProError UgSweepSectSktCreate(ProSection section)
{
    ProError err;
    Pro2dCircleddef circle;
    int circle_id;
    ProSectionPointType pnt_types[] = {PRO_ENT_WHOLE};
    Pro2dPnt place_pnt = {0.0,0.0};
    int dim_id;
    ProIntlist ids;
    int n_ids;

```

```

err = ProSectionEntityIdsGet(section,&ids,&n_ids);
ERROR_CHECK("ProSectionEntityIdsGet", "UgSweepSectSktCreate",err);

ProTKPrintf("Number of section entities: %d\n",n_ids);

circle.type = PRO_2D_CIRCLE;
circle.center[0] = circle.center[1] = 0.0;
circle.radius = 1.0;
err = ProSectionEntityAdd(section, (Pro2dEntdef *)&circle,&circle_id);
ERROR_CHECK("ProSecitonEntityAdd", "UgSweepSectSktCreate",err);

/*-----*\
Dimension section and add location dimensions if necessary; Section can
be dimensioned with respect to the construction entities created by
default (i.e. xy centerlines) although not needed here since the circle is
placed at the origin.
\*-----*/
err = ProSecdimCreate(section,&circle_id,pnt_types,1,
                      PRO_TK_DIM_DIA,place_pnt,&dim_id);
ERROR_CHECK("ProSecdimCreate", "UgSweepSectSktCreate",err);

err = UgSectionRegenerate(section);

return(err);
}
/*=====*\
Function: UgSectionRegenerate
Purpose: Regenerate a section
\*=====*/
ProError UgSectionRegenerate(ProSection section)
{
    ProWSecerror sec_errs;
    ProError err;

    err = ProSecerrorAlloc(&sec_errs);
    ERROR_CHECK("ProSecerrorAlloc", "UgSectionRegenerate",err);

    err = ProSectionSolve(section,&sec_errs);
    ERROR_CHECK("ProSectionSolve", "UgSectionRegenerate",err);

    err = ProSecerrorFree(&sec_errs);
    ERROR_CHECK("ProSecerrorFree", "UgSectionRegenerate",err);

    err = ProSecerrorAlloc(&sec_errs);
    ERROR_CHECK("ProSecerrorAlloc", "UgSectionRegenerate",err);

    err = ProSectionRegenerate(section,&sec_errs);
    ERROR_CHECK("ProSectionRegenerate", "UgSectionRegenerate",err);

    err = ProSecerrorAlloc(&sec_errs);
    ERROR_CHECK("ProSecerrorAlloc", "UgSectionRegenerate",err);

```

```

    return(err);
}

```

Example 2: Creating a Sweep Protrusion Feature by Conventional Approach

The example shows how to create a Sweep Protrusion feature by the conventional approach for the sketched features.

The user is prompted to select the following:

- Sketching plane
- Orientation plane
- Orthogonal edge for the dimensioning of the spine (trajectory - PRO_E_SWEEP_SPINE) section
- Orthogonal edge for the sweep section (PRO_E_SWEEP_SECTION)

```

/*=====*\
FILE      : Ug3DSweepCreate.c
PURPOSE   : Creating a Sweep Protrusion Feature by conventional approach
\*=====*/
#include "ProToolkit.h"
#include "ProFeature.h"
#include "ProElemId.h"
#include "ProStdSection.h"
#include "ProFeatType.h"
#include "ProFeatForm.h"
#include "ProSelection.h"
#include "ProSection.h"
#include "ProSweep.h"

static ProFileName message_file;
static ProSelection *references1;
static ProError status;

#define C_LOG(a) ProTKPrintf ("Status for %s is = %d\n", a, status ); \
                  ERROR_CHECK( a, "UgSweepCreate.c", status );
#define C_PRINT(a) ProTKPrintf ( "%s\n", a);

ProError UserSweepSectionAdd ( ProSection );
ProError UserSweepSpineAdd ( ProSection , ProSelection * );
ProError UserSecerrorPrint ( ProWSecerror *section_errors );

/*=====*\
FUNCTION  : UgSweepProtrCreate
PURPOSE   : Demonstrates the creation of the Simple Sweep Protrusion
\*=====*/

```

```

ProError UgSweepProtrCreate()
{
    ProElement pro_e_feature_tree;
    ProElement pro_e_feature_type;
    ProElement pro_e_feature_form;
    ProElement pro_e_sweep_spine;
    ProElement pro_e_std_sec_setup;
    ProElement pro_e_std_sec_method;
    ProElement pro_e_std_section_plane;
    ProElement pro_e_std_sec_plane;
    ProElement pro_e_std_sec_plane_view_dir;
    ProElement pro_e_std_sec_plane_orient_dir;
    ProElement pro_e_std_sec_plane_orient_ref;
    ProElement pro_e_sweep_section;
    ProValue value;
    ProValueData value_data;
    ProSelection *p_select;
    int n_select;

    ProMdl      model_current;
    ProModelitem model_item;
    ProSelection model_selection;
    ProFeature   created_feature;
    ProErrorlist feat_errors;
    ProElement   created_elemtree;
    ProElement   created_elemtree_after1st;
    ProElemppath element_path;
    ProElemppathItem element_path_item[2];
    ProElement   sketch_element;

    ProFeatureCreateOptions create_options[] =
                                {PRO_FEAT_CR_INCOMPLETE_FEAT};
    ProFeatureCreateOptions redefine_options[] =
                                {PRO_FEAT_CR_DEFINE_MISS_ELEMS};

    ProStringToWstring ( message_file, "msg_ug3dsketch.txt" );

    references1 = ( ProSelection *) calloc (2, sizeof (ProSelection));

    /*-----*\
        Populating root element PRO_E_FEATURE_TREE
    \*-----*/
    C_PRINT( " *** Processing Element PRO_E_FEATURE_TREE *** " );
    status = ProElementAlloc ( PRO_E_FEATURE_TREE, &pro_e_feature_tree);
    C_LOG( " ProElementAlloc " );

    /*-----*\
        Populating element PRO_E_FEATURE_TYPE
    \*-----*/
    C_PRINT( " *** Processing Element PRO_E_FEATURE_TYPE *** " );
    status = ProElementAlloc ( PRO_E_FEATURE_TYPE, &pro_e_feature_type);

```

```

C_LOG( " PRO_E_FEATURE_TYPE ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_FEAT_PROTRUSION;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_feature_type, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_feature_type );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_FEATURE_FORM
\*-----*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_FORM *** " );
status = ProElementAlloc ( PRO_E_FEATURE_FORM, &pro_e_feature_form );
C_LOG( " PRO_E_FEATURE_FORM ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_SWEEP;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_feature_form, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_feature_form );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_SWEEP_SPINE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_SWEEP_SPINE *** " );
status = ProElementAlloc ( PRO_E_SWEEP_SPINE, &pro_e_sweep_spine );
C_LOG( " PRO_E_SWEEP_SPINE ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_sweep_spine );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
    Populating element PRO_E_SWEEP_SPINE -> PRO_E_STD_SEC_SETUP
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SEC_SETUP *** " );
status = ProElementAlloc ( PRO_E_STD_SEC_SETUP, &pro_e_std_sec_setup );
C_LOG( " PRO_E_STD_SEC_SETUP ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_sweep_spine, NULL,
                                pro_e_std_sec_setup );
C_LOG( " ProElemtreeElementAdd" );

```

```

/*-----*\
    Populating element PRO_E_STD_SEC_SETUP -> PRO_E_STD_SEC_METHOD
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SEC_METHOD *** " );
status = ProElementAlloc ( PRO_E_STD_SEC_METHOD,
                          &pro_e_std_sec_method );
C_LOG( " PRO_E_STD_SEC_METHOD ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_SEC_SKETCH;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc " );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet " );
status = ProElementValueSet ( pro_e_std_sec_method, value );
C_LOG( " ProElementValueSet " );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup, NULL,
                                pro_e_std_sec_method );
C_LOG( " ProElemtreeElementAdd " );

/* 1st section */
/*-----*\
    Populating element PRO_E_STD_SEC_SETUP -> PRO_E_STD_SECTION_PLANE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SECTION_PLANE *** " );
status = ProElementAlloc ( PRO_E_STD_SECTION_PLANE,
                          &pro_e_std_section_plane );
C_LOG( " PRO_E_STD_SECTION_PLANE ProElementAlloc " );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup, NULL,
                                pro_e_std_section_plane );
C_LOG( " ProElemtreeElementAdd " );

/*-----*\
    Populating element PRO_E_STD_SEC_SETUP -> PRO_E_STD_SEC_PLANE
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE *** " );
status = ProMessageDisplay ( message_file, "Select Surface for sketch
                                placement");
C_LOG( " ProMessageDisplay" );

status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL, NULL,
                    NULL, NULL, &p_select, &n_select );
C_LOG( " ProSelect " );
if ( n_select <= 0 ) return -1;
else
{
    status = ProSelectionCopy ( p_select[0], &references1[0]);
    C_LOG( " ProSelectionCopy " );
}

status = ProElementAlloc ( PRO_E_STD_SEC_PLANE, &pro_e_std_sec_plane );
C_LOG( " PRO_E_STD_SEC_PLANE ProElementAlloc " );

```

```

value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_sec_plane, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_std_section_plane, NULL,
                                pro_e_std_sec_plane );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
   Populating element PRO_E_STD_SEC_SETUP -> PRO_E_STD_SEC_PLANE_VIEW_DIR
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_VIEW_DIR *** " );
status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_VIEW_DIR,
                          &pro_e_std_sec_plane_view_dir );
C_LOG( " PRO_E_STD_SEC_PLANE_VIEW_DIR ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = 1;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_sec_plane_view_dir, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_std_section_plane, NULL,
                                pro_e_std_sec_plane_view_dir );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
   Populating element PRO_E_STD_SEC_SETUP ->
                                PRO_E_STD_SEC_PLANE_ORIENT_DIR
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_ORIENT_DIR *** "
);
status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_ORIENT_DIR,
                          &pro_e_std_sec_plane_orient_dir );
C_LOG( " PRO_E_STD_SEC_PLANE_ORIENT_DIR ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = 1;
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_sec_plane_orient_dir, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_std_section_plane, NULL,
                                pro_e_std_sec_plane_orient_dir );
C_LOG( " ProElemtreeElementAdd" );

```



```

/*-----*\
    Populating element PRO_E_STD_SEC_SETUP ->
                                PRO_E_STD_SEC_PLANE_ORIENT_REF
\*-----*/
C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_ORIENT_REF *** ");
status = ProMessageDisplay ( message_file, "Select Surface for sketch
                                orientation");
C_LOG( " ProMessageDisplay" );
status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL, NULL,
                    NULL, NULL, &p_select, &n_select );
C_LOG( " ProSelect" );
if ( n_select <= 0 ) return -1;
else
{
    status = ProSelectionCopy ( p_select[0], &references1[1]);
    C_LOG( " ProSelectionCopy" );
}

status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_ORIENT_REF,
                            &pro_e_std_sec_plane_orient_ref );
C_LOG( " PRO_E_STD_SEC_PLANE_ORIENT_REF ProElementAlloc " );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
C_LOG( " ProValueAlloc" );
status = ProValueDataSet ( value, &value_data );
C_LOG( " ProValueDataSet" );
status = ProElementValueSet ( pro_e_std_sec_plane_orient_ref, value );
C_LOG( " ProElementValueSet" );
status = ProElemtreeElementAdd ( pro_e_std_section_plane, NULL,
                                pro_e_std_sec_plane_orient_ref );
C_LOG( " ProElemtreeElementAdd" );

/*-----*\
Populating element PRO_E_SWEEP_SECTION
\*-----*/
C_PRINT( " *** Processing Element PRO_E_SWEEP_SECTION *** " );
status = ProElementAlloc ( PRO_E_SWEEP_SECTION, &pro_e_sweep_section );
C_LOG( " PRO_E_SWEEP_SECTION ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_sweep_section );
C_LOG( " ProElemtreeElementAdd" );

C_PRINT( " *** Processing Element PRO_E_STD_SEC_SETUP *** " );
status = ProElementAlloc ( PRO_E_STD_SEC_SETUP, &pro_e_std_sec_setup );
C_LOG( " PRO_E_STD_SEC_SETUP ProElementAlloc" );
status = ProElemtreeElementAdd ( pro_e_sweep_section, NULL,
                                pro_e_std_sec_setup );
C_LOG( " ProElemtreeElementAdd" );

```

```

/*-----*\
Create the incomplete feature so Pro/ENGINEER allocates the section
handles.
\*-----*/
    status = ProMdlCurrentGet( &model_current );
    C_LOG( " ProMdlCurrentGet " );

    status = ProMdlToModelitem( model_current, &model_item );
    C_LOG( " ProMdlToModelitem " );

    status = ProSelectionAlloc( NULL, &model_item, &model_selection );
    C_LOG( " ProSelectionAlloc " );

    status = ProFeatureCreate ( model_selection, pro_e_feature_tree,
                                create_options, 1, &created_feature, &feat_errors );
    C_LOG( " ProFeatureCreate " );

    if ( status != PRO_TK_NO_ERROR )
        return status;

/*-----*\
Extract the current feature element tree
\*-----*/
    status = ProFeatureElemtreeExtract ( &created_feature, NULL,
                                         PRO_FEAT_EXTRACT_NO_OPTS,
                                         &created_elemtree);
    C_LOG ( "ProFeatureElemtreeExtract()" );

    element_path_item[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
    element_path_item[0].path_item.elem_id = PRO_E_SWEEP_SPINE;
    element_path_item[1].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
    element_path_item[1].path_item.elem_id = PRO_E_SKETCHER;

    status = ProElempathAlloc ( &element_path );
    C_LOG( " ProElempathAlloc " );

    status = ProElempathDataSet ( element_path, element_path_item, 2 );
    C_LOG( " ProElempathDataSet " );

    status = ProElemtreeElementGet ( created_elemtree,
                                     element_path, &sketch_element );
    C_LOG( " ProElemtreeElementGet " );

    ProElempathFree ( &element_path );

    status = ProElementValueGet ( sketch_element, &value );
    C_LOG( " ProElementValueGet " );

    status = ProValueDataGet ( value, &value_data );
    C_LOG( " ProValueDataGet " );

```

```

/*-----*\
Add the trajectory sketch and partially redefine the feature
\*-----*/
status = UserSweepSpineAdd ( ( ProSection ) (value_data.v.p),
                             ( ProSelection * )references1);
C_LOG( " UserSweepSpineAdd " );

status = ProFeatureRedefine ( NULL, &created_feature,
                             created_elemtree, redefine_options, 1, &feat_errors );
C_LOG( " ProFeatureRedefine Spine " );

/*-----*\
Extract the new feature element tree. Now the sweep section sketcher
pointer is available for use.
\*-----*/
status = ProFeatureElemtreeExtract (&created_feature, NULL,
                                    PRO_FEAT_EXTRACT_NO_OPTS,
                                    &created_elemtree_after1st);
C_LOG ( "ProFeatureElemtreeExtract()" );

element_path_item[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
element_path_item[0].path_item.elem_id = PRO_E_SWEEP_SECTION;
element_path_item[1].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
element_path_item[1].path_item.elem_id = PRO_E_SKETCHER;

status = ProElemppathAlloc ( &element_path );
C_LOG( " ProElemppathAlloc " );

status = ProElemppathDataSet ( element_path, element_path_item, 2 );
C_LOG( " ProElemppathDataSet " );

status = ProElemtreeElementGet ( created_elemtree_after1st,
                                element_path, &sketch_element );
C_LOG( " ProElemtreeElementGet " );

ProElemppathFree ( &element_path );

status = ProElementValueGet ( sketch_element, &value );
C_LOG( " ProElementValueGet " );

status = ProValueDataGet ( value, &value_data );
C_LOG( " ProValueDataGet " );

/*-----*\
Add the section sketch and finish the feature.
\*-----*/
status = UserSweepSectionAdd ( value_data.v.p );
C_LOG( " PTTestSection2DAdd " );

status = ProFeatureRedefine ( NULL, &created_feature,
                             created_elemtree_after1st, redefine_options,

```

```

        1, &feat_errors );
C_LOG( " ProFeatureRedefine " );

/*-----*\
Free memory allocated by this function.
\*-----*/
status = ProElementFree ( &pro_e_feature_tree );
C_LOG( " ProElementFree " );

status = ProFeatureElemtreeFree ( &created_feature, created_elemtree);
C_LOG( " ProFeatureElemtreeFree " );

status = ProFeatureElemtreeFree ( &created_feature,
created_elemtree_after1st);
C_LOG( " ProFeatureElemtreeFree " );

free (references1);

return status;
}

#undef C_LOG
#undef C_PRINT

```

Element Trees: ECAD Area Feature

This chapter describes how to access ECAD Area features through Pro/TOOLKIT.

Topic	Page
Introduction to ECAD Area Feature	44 - 2
Feature Element Tree for the ECAD Area	44 - 2

Introduction to ECAD Area Feature

An ECAD Area specifies where you can place electrical components or cannot place them to avoid interference with other electric components or electrical routing. You can create an ECAD area as a sketched cosmetic feature of an electrical board part. However, since this area is sketched, you cannot dimension or regenerate it.

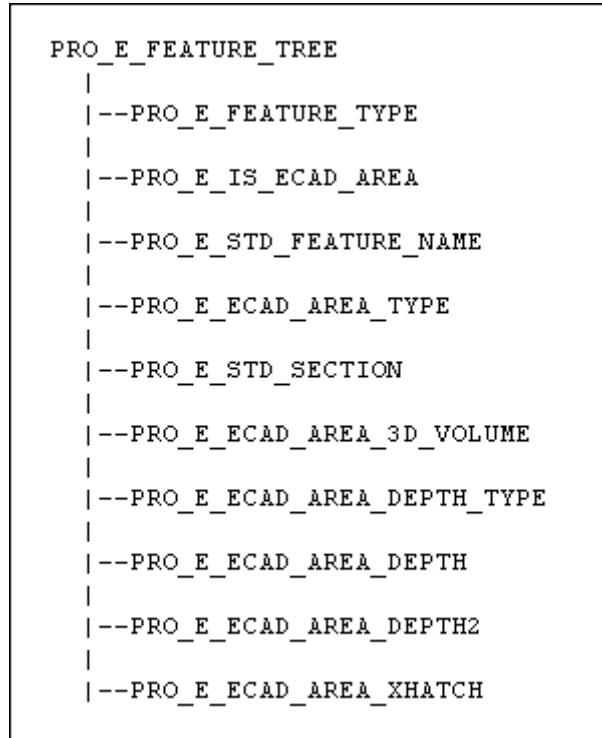
The ECAD area can have a closed 3D volume represented by a quilt. Use these quilts to perform a clearance and interference check on the neighboring electric components. You can access the geometry of the ECAD area feature by using standard Pro/TOOLKIT functions such as **ProFeatureGeomitemVisit()** and **ProSolidQuiltVisit()**.

Feature Element Tree for the ECAD Area

The element tree for the ECAD Area feature is documented in the header file *ProEcadArea.h*.

The following figure demonstrates the element tree structure:

Figure 44-1: Feature Element Tree for an ECAD Area



Element Trees:
ECAD Area Feature

The information about the elements in this tree is as follows:

- PRO_E_FEATURE_TYPE—Specifies the feature type; should always have the value PRO_FEAT_COSMETIC.
- PRO_E_IS_ECAD_AREA—Specifies whether the created cosmetic feature is an ECAD area. This element must have the value PRO_B_TRUE to distinguish this feature from other cosmetic features.
- PRO_E_STD_FEATURE_NAME—Specifies the name of the ECAD area feature.
- PRO_E_ECAD_AREA_TYPE—Specifies the type of ECAD area in the form of the enumerated type *ProEcadAreaType*. This element can take the following values:
 - PRO_ECAD_AREA_TYPE_PLACE_KEEPIN—Specifies a keepin region. You can specify the depth of the area above and below the electric board while creating this region. The default depth value is zero.

- PRO_ECAD_AREA_TYPE_PLACE_KEEPOUT—Specifies a keepout region. You can specify the depth of the area above and below the electric board while creating this region. The default depth value is zero.
- PRO_ECAD_AREA_TYPE_PLACE_REGION—Specifies a region.
- PRO_ECAD_AREA_TYPE_ROUTE_KEEPIN—Specifies a routing keepin region where routing actions are permitted.
- PRO_ECAD_AREA_TYPE_ROUTE_KEEPOUT—Specifies a routing keepout region where routing actions are not permitted.
- PRO_ECAD_AREA_TYPE_VIA_KEEPOUT—Specifies an area where you cannot create vias.
- PRO_E_STD_SECTION—Specifies the sketched region for the ECAD area. For more information on how to populate the section elements of the sketched region, refer to the 'Element Trees: Sketched Features' chapter.
- PRO_E_ECAD_AREA_3D_VOLUME—A ProBoolean element that specifies whether the ECAD area appears with a 3D quilt. Only PRO_ECAD_AREA_TYPE_PLACE_KEEPIN and PRO_ECAD_AREA_TYPE_PLACE_KEEPOUT types of ECAD areas accept a 3D volume on one or two sides.
- PRO_E_ECAD_AREA_DEPTH_TYPE—Specifies the depth type in the form the enumerated type *ProEcadAreaDepthType*. The depth options are applicable only if the element PRO_E_ECAD_AREA_3D_VOLUME is set to PRO_B_TRUE. These options are as follows:
 - PRO_ECAD_AREA_DEPTH_ONE_SIDE—Creates the ECAD area with a 3D volume on one side of the electrical board.
 - PRO_ECAD_AREA_DEPTH_TWO_SIDES_SYM—Places the created ECAD area with a 3D volume at the top and bottom of the electrical board symmetrically.
 - PRO_ECAD_AREA_DEPTH_TWO_SIDES_NOT_SYM—Places the created ECAD area with a 3D volume at the top and bottom of the electrical board asymmetrically.
- PRO_E_ECAD_AREA_DEPTH—Specifies the depth for the ECAD area.

- **PRO_E_ECAD_AREA_DEPTH2**—Specifies the other depth value, which is applicable only if the element **PRO_E_ECAD_AREA_DEPTH_TYPE** is set to **PRO_ECAD_AREA_DEPTH_TWO_SIDES_NOT_SYM**.
- **PRO_E_ECAD_AREA_XHATCH**—Specifies whether the ECAD area is created as a meshed region.

Assembly: Basic Assembly Access

This chapter describes the Pro/TOOLKIT functions that access the contents of a Pro/ENGINEER assembly. Before you read this chapter, you should be familiar with the following documentation:

- User Interface: Selection
- Core: Coordinate Systems and Transformations
- The Core: 3D Geometry section

Topic	Page
Structure of Assemblies and Assembly Objects	45 - 2
Visiting Assembly Components	45 - 5
Locations of Assembly Components	45 - 11
Locations of Assembly Components	45 - 11
Assembling Components	45 - 13
Redefining and Rerouting Components	45 - 13
Deleting Components	45 - 13
Merge and Cutout	45 - 15
Automatic Interchange	45 - 15

Structure of Assemblies and Assembly Objects

The object *ProAssembly* is an instance of *ProSolid* and shares the same declaration. The *ProAssembly* object can therefore be used as input to any of the *ProSolid* and *ProMdl* functions applicable to assemblies. In particular, because you can use the function **ProSolidFeatVisit()** to traverse features, you can extract the assembly datum features and their geometry in the same way as for parts (described in detail in the chapter on Core: 3D Geometry).

However, assemblies do not contain active geometry items other than those in datums—that is, no “solid” geometry as described in the “Core: 3D Geometry” and “Element Trees: Principles of Feature Creation” chapters. Therefore, the function **ProSolidSurfaceVisit()** will not find any surfaces, and solid assembly features such as holes and slots will not contain active surfaces or edges.

The solid geometry of an assembly is contained entirely in its components. Each component is a feature of type PRO_FEAT_COMPONENT, which is a reference to a part or another assembly, plus a set of parametric constraints for determining its geometric location within the parent assembly.

Assembly features that are solid, such as holes and slots, and therefore affect the solid geometry of parts in the assembly hierarchy, do not themselves contain the geometry items that describe those modifications. These items are always contained in the parts whose geometry is modified, within local features created for that purpose of type PRO_FEAT_ASSEM_CUT.

The most important Pro/TOOLKIT functions for assemblies are those that operate on the components of an assembly. The object *ProAsmcomp*, which is an instance of *ProFeature* and shares its DHandle declaration, is defined for that purpose. Each assembly component is treated as a variety of feature, and the integer identifier of the component is also the feature identifier.

An assembly component can be another assembly. In general, therefore, an assembly can contain a hierarchy of assemblies and parts at many levels, in which some assemblies and parts may appear more than once. To identify the role of any database item in the context of the root assembly, it is not enough to have the integer identifier of the item and the handle to its owning part or assembly, as would be provided by its *ProFeature* or *ProGeomitem* description.

It is also necessary to give the full path of the assembly-component references down from the root assembly to the part or assembly that owns the database item. The object *ProAsmcomppath*, which is used as the input to Pro/TOOLKIT assembly functions, accomplishes this purpose.

The declaration of *ProAsmcomppath* is as follows:

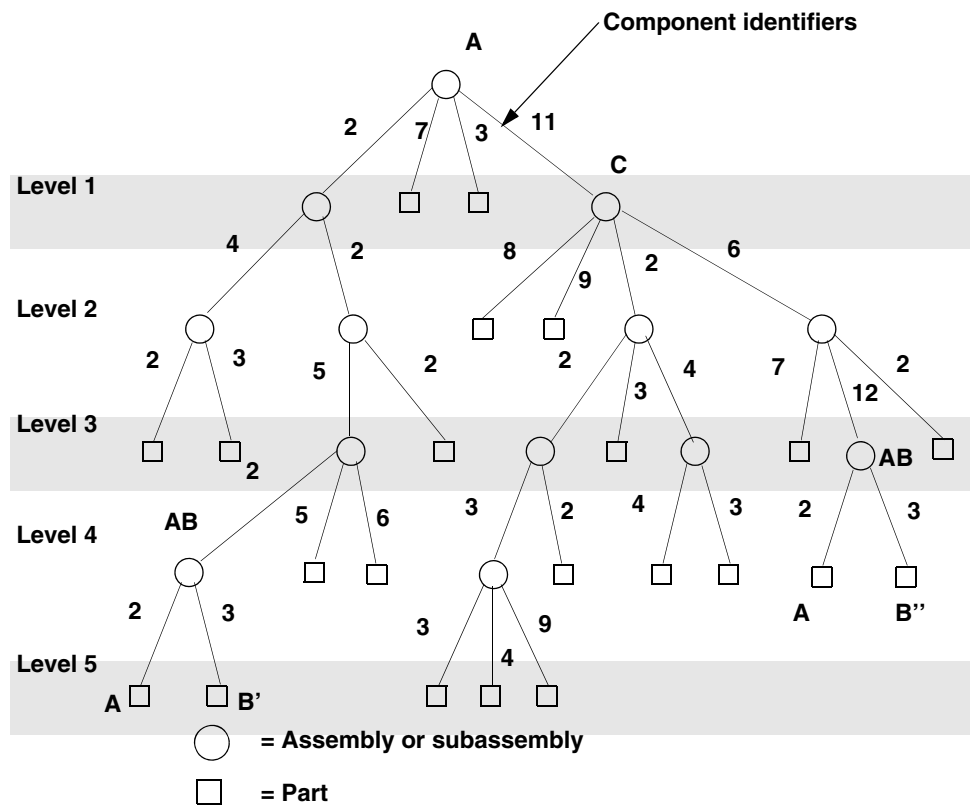
```
typedef struct pro_comp_path
{
    ProSolid    owner;
    ProIdTable  comp_id_table;
    int         table_num;
} ProAsmcomppath;
```

The data structure fields are as follows:

- *owner*—Identifies the root assembly
- *comp_id_table* (the component identifier table)—An integer array that contains the identifiers of the components that form the path from the root assembly down to the component part or assembly being referred to
- *table_num*—Specifies the number of component identifiers in the *comp_id_table* array

The following figure Sample Assembly Hierarchy shows an assembly hierarchy with two examples of the contents of a *ProAsmcomppath* object.

Figure 45-1: Sample Assembly Hierarchy



In the assembly shown in Figure 12-1: Sample Assembly Hierarchy, subassembly C is component identifier 11 within assembly A, Part B is component identifier 3 within assembly AB, and so on. The subassembly AB occurs twice. To refer to the two occurrences of part B, use the following:

Component B'

Component B''

```
table_num = 5
```

```
comp_id_tab[0] = 2
```

```
comp_id_tab[1] = 2
```

```
comp_id_tab[2] = 5
```

```
comp_id_tab[3] = 2
```

```
comp_id_tab[4] = 3
```

```
table_num = 4
```

```
comp_id_tab[0] = 11
```

```
comp_id_tab[1] = 6
```

```
comp_id_tab[2] = 12
```

```
comp_id_tab[3] = 3
```

A *ProAsmcomppath* structure in which *table_num* is set to 1 contains the same information as a *ProAsmcomp* object.

The object *ProAsmcomppath* is one of the main ingredients in the *ProSelection* object, as described in The Selection Object.

Visiting Assembly Components

Functions Introduced:

- **ProSolidFeatVisit()**
- **ProFeatureTypeGet()**

Each component of an assembly is also a feature of that assembly. Therefore, to visit the components, visit the features using **ProSolidFeatVisit()** and find those features whose type is PRO_FEAT_COMPONENT using the function **ProFeatureTypeGet()**. You can convert the *ProFeature* object for each component to the *ProAsmcomp* object by casting.

Properties Related to Component Purpose

Functions Introduced:

- **ProAsmcomppathInit()**
- **ProAsmcompMdlNameGet()**
- **ProAsmcompMdlGet()**
- **ProAsmcomppathMdlGet()**
- **ProAsmcompTypeGet()**

To create a *ProAsmcomppath* object for the component, use the function **ProAsmcomppathInit()** and set the component identifier table to contain only a single component identifier.

The function **ProAsmcompMdlNameGet()** retrieves the model name and type for the component. If an assembly component is missing on retrieval, the function **ProAsmcompMdlNameGet()** still provides information about the component while the function **ProAsmcompMdlGet()** fails to retrieve a valid model handle.

The function **ProAsmcompMdlGet()** provides the *ProMdl* handle to the part or assembly being referenced by the component. To traverse the components at all levels in the assembly hierarchy, make a recursive function to perform these steps:

1. Call **ProAsmcompMdlGet()** for each component of the root assembly to find the model for the component.
2. Call **ProMdlTypeGet()** to find out if the model is a part or an assembly.
3. If the model is an assembly, traverse each component by calling **ProSolidFeatVisit()** again.

The function **ProAsmcomppathMdlGet()** retrieves a model specified by *ProAsmcomppath* and is useful when analyzing a *ProSelection* object that refers to an assembly.

The function **ProAsmcompTypeGet()** yields the type of the assembly component. Examples of the possible types are as follows:

- PRO_ASM_COMP_TYPE_WORKPIECE—Workpiece
- PRO_ASM_COMP_TYPE_REF_MODEL—Reference model
- PRO_ASM_COMP_TYPE_FIXTURE—Fixture
- PRO_ASM_COMP_TYPE_MOLD_BASE—Mold base
- PRO_ASM_COMP_TYPE_MOLD_COMP—Mold component
- PRO_ASM_COMP_TYPE_MOLD_ASSEM—Mold assembly
- PRO_ASM_COMP_TYPE_GEN_ASSEM—General assembly
- PRO_ASM_COMP_TYPE_CAST_ASSEM—Cast assembly
- PRO_ASM_COMP_TYPE_DIE_BLOCK—Die block
- PRO_ASM_COMP_TYPE_DIE_COMP—Die component
- PRO_ASM_COMP_TYPE_SAND_CORE—Sand core
- PRO_ASM_COMP_TYPE_CAST_RESULT—Cast result
- PRO_ASM_COMP_TYPE_FROM_MOTION—Component for use by Pro/MECHANICA.
- PRO_ASM_COMP_TYPE_NO_DEF_ASSUM—Component for which Pro/ENGINEER cannot apply default assumptions.

Component Placement

- **ProAsmcomplsBulkitem()**
- **ProAsmcomplsPackaged()**

The function **ProAsmcompIsBulkitem()** reports whether an assembly component is a bulk item. A bulk item is a non-geometric assembly feature that should appear in an assembly bill of materials.

Use the function **ProAsmcompIsPackaged()** to determine whether the specified component is packaged.

Simplified Representations

- **ProAsmcomplsUnderconstrained()**
- **ProAsmcomplsFrozen()**
- **ProAsmcomplsUnplaced()**
- **ProAsmcomplsPlaced()**
- **ProAsmcomplsSubstitute()**
- **ProAsmcompVisibilityGet()**

The function **ProAsmcompIsUnderconstrained()** determines whether the specified component is underconstrained, that is, it has one or more constraints but they are not sufficient to fully constraint the component location.

The function **ProAsmcompIsFrozen()** determines whether the specified component is frozen. The frozen component behaves similar to the packaged component and does not follow the constraints that you specify.

The functions **ProAsmcompIsUnplaced()** and **ProAsmcompIsPlaced()** determine whether the specified component is unplaced or placed respectively. Unplaced components belong to an assembly without being assembled or packaged.

The function **ProAsmcompIsSubstitute()** determines whether the specified component is substituted. When you substitute a component in a simplified representation, you temporarily exclude the substituted component and superimpose the substituting component in its place.

The function **ProAsmcompVisibilityGet()** enables you to skip components of the master representation that are not shown in the representation when you traverse the assembly components of a simplified representation.

Modifying Component Properties

- **ProAsmcompTypeSet()**
- **ProAsmcompFillFromMdl()**
- **ProAsmcompMakeUniqueSubasm()**
- **ProAsmcompRmvUniqueSubasm()**
- **ProAsmcompSetPlaced()**

The function **ProAsmcompTypeSet()** enables you to set the type of a component.

The function **ProAsmcompFillFromMdl()** copies the template model into the model of the component.

Use the function **ProAsmcompMakeUniqueSubasm()** to create a unique instance of a sub-assembly by specifying the path of the sub-assembly. The function **ProAsmcompRmvUniqueSubasm()** removes the instance of the sub-assembly.

The function **ProAsmcompSetPlaced()** forces Pro/ENGINEER to consider a particular component to be placed or unplaced.

Example 1: Listing the Members of an Assembly

This example recursively lists the components of an assembly and writes the name of the solid for each component.

```
/*-----*\
Pro/TOOLKIT includes
\*-----*/
#include <ProToolkit.h>
#include <ProMdl.h>
#include <ProFeature.h>
#include <ProObjects.h>
#include <ProFeatType.h>
#include <ProUtil.h>
#include <PTApplsUnicodeUtils.h>
#include <ProAsmcomp.h>
#include <ProSolid.h>
/*-----*\
Application includes
\*-----*/
#include <TestError.h>
```

```

/*-----*\
Global definitions
/*-----*/
#define FILENAME "assembly.lst"

typedef struct user_appdata
{
    FILE *fp;
    int level;
} UserAppdata;
/*-----*\
Function prototypes
/*-----*/
int UserAsmCompVisit(void *dummy, int dummy2);
ProError UserAsmCompFilter(ProFeature *feature, ProAppData app_data);
ProError user_action(ProFeature *feature, ProError status,
                    ProAppData appdata);
/*-----*\
Write the information to a file
/*-----*/
ProError user_action(
    ProFeature *feature,
    ProError status,
    ProAppData appdata)
{
    FILE *fp;
    int l, level;
    ProError err;
    ProMdl mdl;
    char name[PRO_NAME_SIZE];
    char type[PRO_TYPE_SIZE];
    wchar_t wname[PRO_NAME_SIZE];
    UserAppdata *appd, appd1;
    ProMdldata mdldata;

    appd = (UserAppdata *)appdata;
    fp = appd->fp;
    level = appd->level;
    err = ProAsmcompMdlGet(feature, &mdl);
    ERROR_CHECK("user_action", "ProAsmcompMdlGet", err);
    err = ProMdlDataGet(mdl, &mdldata);
    ERROR_CHECK("user_action", "ProModelitemNameGet", err);
    err = ProWstringToString(name, mdldata.name);
    err = ProWstringToString(type, mdldata.type);
    for(l=0; l<level; l++)
        ProTKFprintf(fp, "    ");
    ProTKFprintf(fp, "%s %s\n", name, type);
    if (strncmp(type, "ASM", 3) == 0)
    {
        appd1.fp = appd->fp;
        appd1.level = appd->level+1;
    }
}

```

```

        err = ProSolidFeatVisit(mdl, user_action, UserAsmCompFilter, &appd1);
        ERROR_CHECK( "user_action", "ProSolidFeatVisit", err );
    }
    if (feature != NULL) return(PRO_TK_NO_ERROR);
    return(PRO_TK_CONTINUE);
}
/*-----*\
Function to write out the members of the current assembly, and
display the result in an information window.
\*-----*/
int UserAsmCompVisit(void *dummy, int dummy2)
{
    char name[PRO_NAME_SIZE];
    char type[PRO_TYPE_SIZE];
    wchar_t wname[PRO_NAME_SIZE];
    ProMdldata mdldata;
    ProError err;
    UserAppdata appdata;
    FILE *fp;
    ProMdl p_asm;

    err = ProMdlCurrentGet(&p_asm);
    ERROR_CHECK( "UserAsmCompVisit", "ProMdlCurrentGet", err );
/*-----*\
    Open the text file.
\*-----*/
    strcpy(name,FILENAME);
    fp = PTApplsUnicodeFopen(name,"w");
    err = ProMdlDataGet(p_asm, &mdldata);
    ERROR_CHECK( "UserAsmCompVisit", "ProMdlDataGet", err );
    ProWstringToString(name, mdldata.name);
    ProWstringToString(type, mdldata.type);
    ProTKFprintf(fp, "%s%s\n",name,type);
    appdata.fp = fp;
    appdata.level = 1;
/*-----*\
    List the assembly members.
\*-----*/
    err = ProSolidFeatVisit(p_asm, user_action, UserAsmCompFilter,
                            &appdata);
    ERROR_CHECK( "UserAsmCompVisit", "ProSolidFeatVisit", err );
/*-----*\
    Close the file and display it.
\*-----*/
    fclose(fp);
    ProStringToWstring(wname, FILENAME);
    err = ProInfoWindowDisplay(wname, NULL, NULL);
    ERROR_CHECK( "UserAsmCompVisit", "ProInfoWindowDisplay", err );
    return(PRO_TK_NO_ERROR);
}
/*=====*\

```

```

FUNCTION:  UserAsmCompFilter()
PURPOSE:   A filter used by ProSolidFeatVisit() to visit
           features that are assembly components
\*=====*/
ProError UserAsmCompFilter (
    ProFeature *feature,
    ProAppData app_data)
{
    ProError    status;
    ProFeattype ftype;
/*-----*\
    Get the feature type
\*-----*\
    status = ProFeatureTypeGet (feature, &ftype);
/*-----*\
    If the feature is an assembly component,
    return NO ERROR,
    else
    return CONTINUE
\*-----*\
    if(ftype == PRO_FEAT_COMPONENT)
        return(PRO_TK_NO_ERROR);
    return(PRO_TK_CONTINUE);
}

```

Locations of Assembly Components

Functions Introduced:

- **ProAsmcomppathTrfGet()**
- **ProAsmcomppathTrfSet()**
- **ProAssemblyDynPosGet()**
- **ProAssemblyDynPosSet()**

The function **ProAsmcomppathTrfGet()** provides the transformation matrix that describes the coordinate transformation between the coordinate system of an assembly component and that of the root assembly. As its name implies, its input is a *ProAsmcomppath* object, so it can be applied to a component at any level within an assembly hierarchy. It has an option to provide the transformation from bottom to top, or from top to bottom. (To apply the transformation, use the function **ProPntTrfEval()** or **ProVectorTrfEval()**, described in the section Coordinate Systems.)

In effect, this function describes the current position and orientation of the assembly component in the root assembly.

Example 2: Finding the Position of a Component

This example shows a function that finds the matrix that describes the position of an assembly component in its parent assembly.

```
/*=====*\
Function : UserAsmcompTransfGet()
Purpose  : Find the transformation matrix of the specified component
\*=====*/
int UserAsmcompTransfGet()
{
    ProError status;
    ProMdl model;
    int nb_sel, i;
    ProMatrix matrix;
    ProAsmcomppath comp_path;
    ProIdTable comp_id_table;
    ProSelection *sel_list, selection;
    ProModelitem sel_obj;
    ProFeature feature;
    ProFeattypeftype;
    FILE *fp_out=NULL;
    ProName w_name, wfile_name;
    ProCharName name;

    status = ProSelect ("prt_or_asm",1,NULL,NULL,NULL,NULL,
                        &sel_list,&nb_sel);
    ERROR_CHECK("UserAsmcompTransfGet", "ProSelect", status);

    if (status == PRO_TK_NO_ERROR)
    {
        if (nb_sel > 0)
        {
            status = ProSelectionModelitemGet (sel_list[0],&sel_obj);
            ERROR_CHECK("UserAsmcompTransfGet", "ProSelectionModelitemGet",
                        status);
            status = ProSelectionHighlight (sel_list[0],PRO_COLOR_ERROR);
            ERROR_CHECK("UserAsmcompTransfGet", "ProSelectionHighlight",
                        status);
            status = ProSelectionAsmcomppathGet(sel_list[0],&comp_path);
            ERROR_CHECK("UserAsmcompTransfGet", "ProSelectionAsmcomppathGet",
                        status);
            if (status != PRO_TK_NO_ERROR) return (PRO_TK_GENERAL_ERROR);
            status = ProAsmcomppathTrfGet (&comp_path,PRO_B_TRUE,matrix);
            ERROR_CHECK("UserAsmcompTransfGet", "ProAsmcomppathTrfGet",
                        status);
            if (status != PRO_TK_NO_ERROR) return (PRO_TK_GENERAL_ERROR);
            status = ProAsmcomppathMdlGet(&comp_path,&model);
            ERROR_CHECK("UserAsmcompTransfGet", "ProAsmcomppathMdlGet",
                        status);
            if (status != PRO_TK_NO_ERROR) return (PRO_TK_GENERAL_ERROR);
        }
    }
}
```

```

ProMdlNameGet(model,w_name);
ProWstringToString(name,w_name);
fp_out = PTApplsUnicodeFopen("rptfile.txt", "w");
ProTKFprintf(fp_out, " ----- Component Location ----- \n\n");
ProTKFprintf(fp_out, "Component name : %s\n\n",name);
ProTKFprintf(fp_out, "Matrix :\n\n");
for (i=0 ; i<4 ; i++)
    ProTKFprintf(fp_out, "%f\t%f\t%f\t%f\n",
matrix[i][0],matrix[i][1],
matrix[i][2],matrix[i][3]);
fclose(fp_out);
ProStringToWstring(w_name,"rptfile.txt");
status = ProInfoWindowDisplay (w_name,NULL,NULL);
ERROR_CHECK("UserAsmcompTransfGet","ProInfoWindowDisplay",
            status);
if (status != PRO_TK_NO_ERROR) return (-1);
    }
}
return (PRO_TK_NO_ERROR);
}

```

Assembling Components

To assemble components into an assembly, use the methods of feature creation. These methods are described in detail in the chapter Assembly: Assembling Components.

Redefining and Rerouting Components

The functions used to redefine and reroute components are described in the chapter Assembly: Assembling Components.

Deleting Components

Function Introduced:

- **ProFeatureDelete()**

The function **ProFeatureDelete()** deletes components. It has the same options as described in the chapter Core: Features.

Exploded Assemblies

Functions Introduced:

- **ProAssemblyExplode()**
- **ProAssemblyUnexplode()**
- **ProAssemblyIsExploded()**

The functions **ProAssemblyExplode()** and **ProAssemblyUnexplode()** enable you to explode and unexplode an assembly. The function **ProAssemblyIsExploded()** reports whether the specified assembly is exploded.

Note: These functions explode the assembly using the default exploded state of the assembly.

Explode State Objects

The object *ProExpldstate* describes the contents and ownership of an explode state. The object *ProExpldstate* shares the same declaration as *ProModelitem*, *ProGeomitem*, and *ProFeature*, which is as follows:

```
typedef struct pro_model_item
{
    ProType  type;
    int      id;
    ProMdl   owner;
} ProExpldstate;
```

Visiting Explode States

Function Introduced:

- **ProSolidExpldstateVisit()**

The function **ProSolidExpldstateVisit()** enables you to visit all the explode states in the specified solid, *except* for the default explode state. For a detailed explanation of visiting functions, see the section Visit Functions in the Fundamentals chapter.

Explode State Access

Functions Introduced:

- **ProExpldstateActivate()**
- **ProExpldstateSelect()**
- **ProExpldstateActiveGet()**

These functions enable you to access the explode states of a solid.

The function **ProExpldstateActivate()** activates the specified explode state representation of a solid.

The function **ProExpldstateSelect()** prompts the user to select an explode state from the list of explode states that are currently available.

The function **ProExpldstateActiveGet()** retrieves the current active explode state for the specified solid.

Merge and Cutout

Function Introduced:

- **prodb_merge_members()**

The function **prodb_merge_members()** performs the two Pro/ENGINEER operations **Component, Merge**, and **Component, Cutout**. As with the Pro/ENGINEER options, you can choose to make the resultant feature be new and independent (“by copy”) or reference the originating features (“by reference”).

To find out whether two assembly members interfere, use the function **pro_compute_clearance()**, described in the Core: 3D Geometry chapter.

Automatic Interchange

Functions Introduced:

- **prodb_auto_interchange()**
- **prodb_interchange_domain()**

In Pro/ENGINEER, it is possible to interchange an assembly component with another model that contains equivalent assembly constraints. The Pro/TOOLKIT function that performs this action is **prodb_auto_interchange()**. Depending on the type of component interchange, the assembly constraints may need to be respecified for the replacement model.

Instances in a family table share the same assembly constraints. Consequently, you can automatically replace an assembly component with another instance in the component's family table without respecifying any assembly constraints. Simply retrieve the handle for the replacement instance and pass this handle to **prodb_auto_interchange()**.

If the assembly component and replacement model are not instances in the same family table, you can define the necessary relationships between them interactively and save them in an Interchange Assembly. (See the *Assembly Modeling User's Guide* for details.) To perform an interchange using models in an interchange assembly, first retrieve the interchange assembly (using the function **ProMdlRetrieve()**), and then pass the handle of the replacement model to function **prodb_auto_interchange()**. Note that the interchange assembly must be in memory before the call to **prodb_auto_interchange()**.

An interchange assembly is *not* the same as an interchange domain. Interchange domains (*.int* files) contain interchange information, but they can no longer be created using Pro/ENGINEER. However, it is possible to use **prodb_auto_interchange()** to interchange models using an existing interchange domain. Before the call to **prodb_auto_interchange()**, you must read the interchange domain into memory using the function **prodb_interchange_domain()**.

Assembly: Top-down Design

This chapter describes the Pro/TOOLKIT Design Manager functions. For more information on Design Intent, Top-Down Design, and other Design Manager issues, refer to the Assembly portion of the Pro/ENGINEER help data, or the Top-Down Design Task Guide.

Topic	Page
Overview	46 - 2
Skeleton Model Functions	46 - 5
Assembly Component Functions	46 - 5
External Reference Control Functions	46 - 6
Feature and CopyGeom Feature Functions	46 - 7
External Reference Data Gathering	46 - 9

Overview

Pro/ENGINEER supports a design concept called Top-Down Design. Top-Down Design is a method of designing a product by specifying top-level design criteria and passing those criteria down from the top level of the product's structure to all affected subsystems. The Pro/TOOLKIT Design Manager functions support this design concept. The next sections contain a brief summary of the six steps of Top-Down Design.

Defining Design Intent

Before building parts and assemblies, it is important that you define the intent of your design. Doing this means defining:

- Purpose or function of the product
- Major systems and subsystems required
- Incorporation of subsystems into the overall product
- Dependence (if any) on any existing design or product

Design criteria and parameters you specify in this process can be shared globally among all components of the assembly, and can be used to drive design parts, assemblies, and skeleton models.

Defining Preliminary Product Structure

The preliminary product structure consists of a list of components and their hierarchy within the assembly design. This structure allows creation of subassemblies and parts without requiring creation of geometry and without having to assemble parts. You can add existing subassemblies and parts to this structure. You can also define non-geometric information for the entire design and capture design parameters including description, part number, and part type. Pro/TOOLKIT Design Manager manages the assembly structure with assembly component functions.

Introducing Skeleton Models

Skeleton models are a 3-dimensional layout of the assembly. These models are holders or distributors of critical design information, and can represent space requirements, important mounting locations, and motion.

Skeleton models can contain the master definition of the design information, which can be propagated to specific design models. You can also use skeleton models to communicate design information between components. Pro/TOOLKIT Design Manager uses skeleton model functions to manipulate these models.

Communicating Design Intent Throughout the Assembly Structure

Designers can distribute top-level design information to dependent skeleton models in the assembly. Design modification becomes a matter of changing certain distributed properties. This propagation of information first occurs from skeleton to skeleton, and then from skeleton to part until all necessary part- or subassembly-specific references have been distributed. Designers can then work on a small subsystem without retrieving or regenerating the entire top-level assembly.

This distribution lets designers reference the same information instead of recreating it for each subassembly. Pro/TOOLKIT Design Manager handles the assembly structure with functions for assembly components, features, and copy geometry features.

Assembly: Top-down
Design

Continued Population of the Assembly

Populate the assembly with detailed parts and subassemblies in one of two ways:

- Create new components in the context of the assembly
- Model components individually and then bring them into the assembly

Relate individual parts to each other with assembly relations, skeleton models, layouts, and merge features. Pro/TOOLKIT Design Manager functions manage the assembly structure with functions for assembly components, features, and copy geometry features.

Managing Part Interdependencies

Associativity allows you to modify design intent to cause automatic updating of the appropriate objects in your assembly. Associativity is accomplished through external relationships, also known as dependencies or references.

Part interdependencies allow for communication of design criteria from components on one level of the design to components on lower levels. Associativity and part dependencies provide a means for controlled changing or updating of an entire assembly design. Reference control manages part interdependencies by limiting undesirable ones or allowing desirable ones.

External references are dependencies between a Pro/ENGINEER object (part or subassembly) and information from another object not owned by either the referencing object or its submodels. References to “out-of-model” information are external references. Design Manager handles these references with external reference control functions.

Scope is the range of objects to which a specified object can refer. Scope control functions allow you to define objects to which other objects under development can refer. You can establish global scope settings for all objects or specific settings for individual objects.

Design Manager handles scope issues with external reference control functions. The enumerated type *ProExtRefScope* defines possible scope settings as:

- None—Allows no external references.
- Subassembly—Allows external references only to components of the same subassembly
- Skeleton Model—Allows external references to higher-level skeleton models only
- All—Allows external references to any other object anywhere in the assembly

The enumerated type *ProInvalidRefBehavior* defines two methods of handling out-of-scope references. They are as follows:

- Prohibit Out-of-Scope references—Pro/TOOLKIT reports the external reference as out of scope. You must select another reference.
- Copy Out-of-Scope Reference—Pro/TOOLKIT warns that the reference is out of scope. You must do one of the following:
 - Cancel the selection and choose a different reference
 - Confirm that you do want to use the selected reference. Pro/TOOLKIT then creates a “local backup” of the reference. The local backup reference automatically updates (only while the parent is in the current session).

Skeleton Model Functions

Functions Introduced:

- **ProAsmSkeletonCreate()**
- **ProAsmSkeletonAdd()**
- **ProAsmSkeletonGet()**
- **ProAsmSkeletonDelete()**
- **ProMdlIsSkeleton()**

Create skeleton models using function **ProAsmSkeletonCreate()**. This function creates a new skeleton model with the specified name, adds it to the specified assembly, and initializes the model handle. The input arguments are assembly handle, the skeleton model name, and the handle to the part or skeleton used as a template. If the template handle is NULL, an empty skeleton model is created.

ProAsmSkeletonAdd() adds an existing skeleton model to the specified assembly. The input arguments are a handle for the assembly to which the skeleton model will be added, and a handle to the skeleton model.

ProAsmSkeletonGet() returns a skeleton model of the specified assembly that is currently in memory, then initializes the model handle. The input argument is a handle to the specified assembly.

ProAsmSkeletonDelete() deletes a skeleton model component from the specified assembly. The input argument is a handle to the specified assembly.

ProMdlIsSkeleton() determines if the specified model is a skeleton model. The input argument is a handle to the model to be checked.

Assembly Component Functions

Functions Introduced:

- **ProAsmcompCreateCopy()**
- **ProAsmcompIsUnplaced()**
- **ProAsmcompFillFromMdl()**

Create new components in the specified assembly by copying them from a specified model using **ProAsmCompCreateCopy()**. This function creates a new component with the specified name, places it at a default location in the in the assembly, or leaves it unplaced. The input arguments are the assembly to copy from, the new component name, the new component type (either PRO_MDL_ASSEMBLY or PRO_MODEL_PART), the handle to the model used as a template, and specification of default or “unplaced” component placement. If the template handle is NULL, the component is created empty.

ProAsmCompIsUnplaced() determines whether the specified component is unplaced. The input argument is a handle to the component to be checked.

ProAsmCompFillFromMdl() copies the specified template model into a model of the specified component. The input arguments are the handle to the component, and the handle to the model used as a template for the copy.

External Reference Control Functions

Functions Introduced:

- **ProRefCtrlSolidSet()**
- **ProRefCtrlSolidGet()**
- **ProRefCtrlEnvirSet()**
- **ProRefCtrlEnvirGet()**
- **ProRefCtrlCheckScope()**

Function **ProRefCtrlSolidSet()** sets a specified external reference control setting on a solid, that is, on a part or assembly. Use **ProRefCtrlSolidGet()** to retrieve the external reference control setting for a specified solid.

ProRefCtrlEnvirSet() establishes the run-time environment setting for external reference control. Function **ProRefCtrlEnvirGet()** retrieves this data.

Function **ProRefCtrlCheckScope()** checks whether object-specific reference control settings for a specified model (either an independent object or an assembly component) allow that model to reference information belonging to a different model. The top-level assembly for the component being modified and for the component being referenced must be the same.

If **ProRefCtrlCheckScope()** finds that the owner of the component being modified is NULL and the solid (part or assembly) being referenced is not a sub-model of the solid being modified, it reports the reference as out of assembly context. If the ProMdl returned is NULL but there is a scope violation, the environment scope has been violated.

The enumerated type *ProExtRefScope* defines allowed scope settings for external references as follows:

```
typedef enum
{
    PRO_REFCTRL_ALLOW_ALL          = 0, /* all external references allowed */
    PRO_REFCTRL_ALLOW_SUBASSEMBLY = 1, /* allow only external references
                                         inside the same higher level
                                         subassembly as that
                                         of the modified object */
    PRO_REFCTRL_ALLOW_SKELETON     = 2, /* only external references to
                                         skeleton models allowed */
    PRO_REFCTRL_ALLOW_NONE         = 3  /* no external references allowed */
} ProExtRefScope;
```

Enumerated type *ProInvalidRefBehavior* defines the supported methods for handling Out-of-Scope external references as follows:

```
typedef enum
{
    PRO_REFCTRL_BACKUP_REF      = 0, /* create a local backup for
                                         out-of-scope references */
    PRO_REFCTRL_PROHIBIT_REF    = 1  /* prohibit out-of-scope external
                                         references */
} ProInvalidRefBehavior;
```

Feature and CopyGeom Feature Functions

Functions Introduced:

- **ProFeatureCopiedRefStateGet()**
- **ProFeatureHasBackup()**
- **ProFeatureCopyGeomDependSet()**
- **ProFeatureCopyGeomInDependSet()**

Function **ProFeatureCopiedRefStateGet()** retrieves the status of copied references for a specified feature. This function supports both CopyGeom features and features with local backup of references.

Note: CopyGeom features have no local backup of reference data.

ProFeatureHasBackup() determines if the specified feature has local backup of external references.

ProFeatureCopyGeomDependSet() sets copied references of the specified CopyGeom feature to be dependent on the referenced or master model. This means the specified CopyGeom feature references will update as the referenced or master model changes. The function **ProFeatureCopyGeomInDependSet()** sets copied references to be independent of the referenced or master model, that is, they do not update as this top-level model changes.

Enumerated type *ProRefCopiedState* defines possible states of local copies of external references in a CopyGeom feature or in a feature with a local backup.

```
typedef enum
{
    PRO_COPIED_REF_NOT_FOUND = 0,    /* copied references are not found */
    PRO_COPIED_REF_ACTIVE    = 1,    /* copied reference is dependent on
                                     the "master" model and is up to date */
    PRO_COPIED_REF_INDEPEND = 2,    /* copied reference is not dependent
                                     on the "master" model */
    PRO_COPIED_REF_EXCLUDED = 3,    /* copied reference is dependent on
                                     model which currently is excluded
                                     by Simp Rep or by Family Table. */
    PRO_COPIED_REF_FROZEN   = 4,    /* copied reference is dependent on
                                     the "master" model, which currently
                                     is not in the session */
    PRO_COPIED_REF_SUPPRESSED = 5,  /* copied reference is dependent on
                                     an entity which is suppressed
                                     in the "master" model */
    PRO_COPIED_REF_MISSING  = 6,    /* copied reference is dependent on
                                     an entity which is missing
                                     in the "master" model */
    PRO_COPIED_REF_FROZEN_PLACE = 7 /* copied reference cannot update its
                                     location */
    PRO_COPIED_REF_QUILT    = 10 /* copied reference solidification
                                     failed and restored to quilt */
} ProCopiedRefState;
```

External Reference Data Gathering

Functions Introduced:

- **ProFeatureExternChildrenGet()**
- **ProFeatureExternParentsGet()**
- **ProSolidExternChildrenGet()**
- **ProSolidExternChildrenGet()**
- **ProExtRefInfoFree()**
- **ProExtRefStateGet()**
- **ProExtRefTypeGet()**
- **ProExtRefAsmcompsGet()**
- **ProExtRefOwnMdlGet()**
- **ProExtRefMdlGet()**
- **ProExtRefOwnFeatGet()**
- **ProExtRefFeatGet()**
- **ProExtRefModelitemGet()**

Function **ProFeatureExternChildGet()** retrieves information about external and local children of the specified feature according to the specified reference type. **ProFeatureExternParentGet()** does the same for parents of the feature.

Function **ProSolidExternChildGet()** retrieves external and local children of the specified solid according to the specified reference type. **ProSolidExternParentGet()** does the same for parents of the solid.

ProExtRefInfoFree() releases memory allocated to the external reference data for a feature or solid.

ProExtRefStateGet() returns the external reference status of the referenced item of the specified reference. **ProExtRefTypeGet()** returns the type of the external reference.

ProExtRefAsmcompsGet() retrieves from the specified external reference a path to the component from which the reference was created. It also returns a path to the component that owns the specified external reference.

ProExtRefOwnMdlGet() retrieves from the specified external reference a solid (in a model active in the session) that uses the reference. Function **ProExtRefMdlGet()** retrieves from the specified external reference a solid (in a model active in the session) referred to by the reference.

ProExtRefOwnFeatGet() retrieves from the specified external reference a feature that uses the reference. Function **ProExtRefFeatGet()** retrieves from the specified external reference a feature referred to by the external reference.

ProExtRefModelItemGet() retrieves from the specified external reference a model item that uses that reference.

The *ProExtRefType* enumerated type defines supported external reference types as follows:

```
typedef enum
{
    PRO_EXT_GEOM_REF      = 1,      /* all out of solid references,
                                     created in assembly context, kept in
                                     plins, sections, draft sections      */
    PRO_LOC_GEOM_REF      = 2,      /* local for solid references, kept in
                                     plins, sections, draft sections      */
    PRO_MERGE_REF         = 3,      /* reference models of merge by ref feats */
    PRO_EXT_REL_REF       = 4,      /* out of solid references, kept in
                                     symbols used for relations.
                                     Can be "to solid" or feature,
                                     geometry references.                  */
    PRO_LOC_REL_REF       = 5,      /* local for solid references, kept in
                                     symbols used for relations.
                                     Can be "to solid" or feature,
                                     geometry references.                  */
    PRO_PRGM_REF          = 6,      /* out of solid references, kept in
                                     symbols used in Pro/Program.
                                     Always solid references                  */
    PRO_MOVE_COMP_REF     = 7,      /* Move Components external references.
                                     Kept in components and always "to solid" */
    PRO_SUBS_REF          = 8,      /* Substitute Component references. Kept
                                     in components and always "to solid" */
    PRO_MFG_INFO_REF      = 9,      /* Mfg Info references. Kept in
                                     mfg feat, always "to solid"          */
    PRO_INTRCH_REF        = 10,     /* Interchange Assembly references.
                                     Kept in the solid itself.
                                     Always "to solid"                      */
    PRO_HARN_REF          = 11,     /* Harness references.
                                     Kept in the solid itself.
                                     Always "to solid"                      */
    PRO_FEAT_PAT_REF      = 12,     /* Feature pattern references.
                                     Does not include pattern relation
                                     references. Always "to solid"          */
}
```

```

PRO_NON_ASSY_GEOM_REF = 13,      /* Out of solid external geometry refs,
                                   created not in assembly context,
                                   kept in plins. (used in external geom
                                   copy feature). */
PRO_ALL_EXT_REF_TYPES = 100,    /* All the types except for
                                   PRO_LOC_GEOM_REF
                                   and PRO_LOC_REL_REF */
PRO_ALL_REF_TYPES          = 101
} ProExtRefType;

```

The enumerated type *ProRefState* defines the possible states of top-level solids (part, assembly, or component) to which a lower-level solid refers:

```

typedef enum
{
    PRO_REF_NOT_FOUND    = PRO_COPIED_REF_NOT_FOUND, /* references are not
                                                         found */
    PRO_REF_ACTIVE       = PRO_COPIED_REF_ACTIVE,    /* referenced entity on the
                                                         "master" model is up to date */
    PRO_REF_EXCLUDED     = PRO_COPIED_REF_EXCLUDED /* "master" model currently
                                                         is excluded by Simp Rep or by Family Table. */
    PRO_REF_FROZEN       = PRO_COPIED_REF_FROZEN,    /* "master" model currently
                                                         not in session */
    PRO_REF_SUPPRESSED   = PRO_COPIED_REF_SUPPRESSED, /* referenced entity is
                                                         suppressed in the "master" model */
    PRO_REF_MISSING      = PRO_COPIED_REF_MISSING /* referenced entity is
                                                         missing in the "master" model */
    PRO_REF_FROZEN_PLACE = PRO_COPIED_REF_FROZEN_PLACE, /* referenced entity
                                                         on the "master" model is up to date but location
                                                         of "target" model according to "master" model can
                                                         not be updated */
    PRO_REF_INVALID = 8, /* Referenced entity is not valid. For example:
                           - Straight edge becomes non straight;
                           - Planar surface becomes non planar; */
    PRO_REF_ALTERNATE = 9,
    PRO_REF_QUILT = PRO_COPIED_REF_QUILT
} ProRefState; /* state of referenced entity in "master" model */

```

Structures *ProExtFeatRef* and *ProExtRefInfo* provide pointers to a structure containing external references for a specified feature:

```

typedef struct ext_feat_ref *ProExtFeatRef;

typedef struct
{
    ProExtRefType    type;
    ProExtFeatRef    *ext_refs;
    int              n_refs;
} ProExtRefInfo;

```


Assembly: Assembling Components

This chapter describes how to use the concepts of feature creation to assemble components into an assembly. Read the chapter 'Element Trees: Principles of Feature Creation' before this chapter.

Topic	Page
Assembling Components by Functions	47 - 2
Assembling a Component Parametrically	47 - 3
Redefining Components Interactively	47 - 16
Assembling Components by Element Tree	47 - 17
The Element Tree for an Assembly Component	47 - 17
Assembling Components Using Intent Datums	47 - 27

Assembling Components by Functions

Functions Introduced:

- **ProAsmcompCreateCopy()**
- **ProAsmcompAssemble()**
- **ProAsmcompPositionGet()**
- **ProAsmcompPositionSet()**
- **ProAsmcompConstraintsGet()**
- **ProAsmcompConstraintsSet()**
- **ProAsmcompConstrRemove()**
- **ProAsmcompRegenerate()**

Use the function **ProAsmcompCreateCopy()** to create a new component in the assembly by copying from an existing model. Specify the handle to the model to be used as a template for the copy. If a model is not specified, a component that does not have initial geometry is created. The function provides the *ProAsmcomp* handle to the new component.

The function **ProAsmcompAssemble()** assembles a component to the assembly or sub-assembly using the parametric constraints available when assembling a component in Pro/ENGINEER. The initial position of the component is a *ProMatrix* object. Specify the orientation of the three axes and the position of the origin of the component coordinate system, with respect to the target assembly coordinate system. The function provides the *ProAsmcomp* feature handle to the newly created assembly.

The function **ProAsmcompPositionGet()** retrieves the component's initial position before constraint are applied.

The function **ProAsmcompPositionSet()** specifies the initial position of the component before constraints are applied. This affects the position of the component only if the component is packaged or underconstrained.

The function **ProAsmcompConstraintsGet()** retrieves the specified constraints for the given assembly component.

The function **ProAsmcompConstraintsSet()** specifies the position of the component relative to the base component or other components and the datum features in the assembly.

The function **ProAsmcompConstrRemove()** deletes one or all constraints for the specified assembly component. As input, the function takes a *ProAsmcomp* handle to the assembly component and a constraint index. To remove all the constraints, use -1 as the value for the constraint index. You can determine the index of a particular constraint by using the function **ProAsmcompConstraintGet()**.

The function **ProAsmcompRegenerate()** regenerates the placement instructions for an assembly component, given the component handle. The function regenerates the placement instructions just as in an interactive Pro/ENGINEER session. Alternatively, you can use the visit functionality to regenerate recursively some or all of the components in the assembly.

Assembling a Component Parametrically

Functions Introduced:

- **ProAsmcompconstraintAlloc()**
- **ProAsmcompconstraintTypeGet()**
- **ProAsmcompconstraintTypeSet()**
- **ProAsmcompconstraintAsmreferenceGet()**
- **ProAsmcompconstraintAsmreferenceSet()**
- **ProAsmcompconstraintCompreferenceGet()**
- **ProAsmcompconstraintCompreferenceSet()**
- **ProAsmcompconstraintOffsetGet()**
- **ProAsmcompconstraintOffsetSet()**
- **ProAsmcompconstraintAttributesGet()**
- **ProAsmcompconstraintAttributesSet()**
- **ProAsmcompconstraintUserdataGet()**
- **ProAsmcompconstraintUserdataSet()**
- **ProAsmcompconstraintFree()**
- **ProAsmcompconstraintArrayFree()**

The function **ProAsmcompconstraintAlloc()** allocates memory for the constraint data structure. This data structure describes the types of constraints that can be applied to the assembly component.

The function **ProAsmcompconstraintTypeGet()** retrieves the constraint type of the specified constraint. The types of constraints are:

- PRO_ASM_MATE—Use this option to make two surfaces coincident with one another and facing each other.
- PRO_ASM_MATE_OFF—Use this option to make two planar surfaces parallel and facing each other.
- PRO_ASM_ALIGN—Use this option to make two planes coplanar, two axes coaxial or two points coincident. You can also align revolved surfaces or edges.
- PRO_ASM_ALIGN_OFF—Use this option to align two planar surfaces at an offset.
- PRO_ASM_INSERT—Use this option to insert a "male" revolved surface into a "female" revolved surface, making their respective axes coaxial.
- PRO_ASM_ORIENT—Use this option to make two planar surfaces to be parallel in the same direction.
- PRO_ASM_CSYS—Use this option to place a component in an assembly by aligning the coordinate system of the component with the coordinate system of the assembly.
- PRO_ASM_TANGENT—Use this option to force two surfaces to be tangent.
- PRO_ASM_PNT_ON_SRF—Use this option to align a point with a of a surface.
- PROS_ASM_EDGE_ON_SRF—Use this option to align a straight edge with a surface.
- PRO_ASM_DEF_PLACEMENT—Use this option to align the default coordinate system of the component to the default coordinate system of the assembly.
- PRO_ASM_SUBSTITUTE—This constraint type is used in simplified representations only when the component is replaced by a substitute component.
- PRO_ASM_PNT_ON_LINE—Use this option to force the intersection of a line with a point.
- PRO_ASM_FIX—Use this option to fix the current location of the component as a constraint.
- PRO_ASM_AUTO—Not for use by Pro/TOOLKIT.

- **PRO_ASM_ALIGN_ANG_OFF**—This option can only be used in conjunction with another constraint. If you have two flat surfaces and create an align edge or axis constraint where the edge or axis lies on the surface, then you can specify an angle offset constraint between the two surfaces.
- **PRO_ASM_MATE_ANG_OFF**—This option can only be used in conjunction with another constraint. If you have two flat surfaces and create a mate edge or axis constraint where the edge or axis lies on the surface, then you can specify an angle offset constraint between the two surfaces.

Use the function **ProAsmcompconstraintTypeSet()** to set the constraints for the assembly component constraint.

The function **ProAsmcompconstraintAsmreferenceGet()** retrieves the *ProSelection* handle to a reference on the assembly and the orientation of the assembly for the specified assembly component constraint. The assembly orientation can have the following values:

- **PRO_DATUM_SIDE_YELLOW**—The primary side of the datum plane which is the default direction of the arrow.
- **PRO_DATUM_SIDE_RED**—The secondary side of the datum plane which is the direction opposite to that of the arrow.
- **PRO_DATUM_SIDE_NONE**—No orientation is specified.

The function **ProAsmcompconstraintAsmreferenceSet()** selects a reference on the assembly and sets the orientation of the assembly for a specified assembly component constraint.

Note: The assembly reference selection must be assigned an assembly component path, even if the reference geometry is in the top-level assembly. In that situation the *table_num* value of the *ProAsmcomppath* structure would be 0.

The function **ProAsmcompconstraintCompreferenceGet()** retrieves the *ProSelection* handle to a reference on the placed component and the orientation of the component for the specified assembly component constraint. The component orientation can have the following values:

- **PRO_DATUM_SIDE_YELLOW**—The primary side of the datum plane which is the default direction of the arrow.
- **PRO_DATUM_SIDE_RED**—The secondary side of the datum plane which is the direction opposite to that of the arrow.
- **PRO_DATUM_SIDE_NONE**—No orientation is specified.

ProAsmcompconstraintCompferenceSet() selects a reference on the placed component and sets the orientation of the component for a specified assembly component constraint.

ProAsmcompconstraintOffsetGet() retrieves the offset value from the reference for the Mate or Align constraint type and the function **ProAsmcompconstraintOffsetSet()** defines the offset value.

The function **ProAsmcompconstraintAttributesGet()** retrieves the constraint attributes for the specified assembly component constraint. The function

ProAsmcompconstraintAttributesSet() sets the constraint attributes. The types of constraint attributes are:

- **PRO_ASM_CONSTR_ATTR_FORCE**—Force the constraint, causing strict alignment for axes, lines, and points. You can force a constraint only if the constraint type is Align.
- **PRO_ASM_CONSTR_ATTR_IGNORE**—Not for use by Pro/TOOLKIT.
- **PRO_ASM_CONSTR_ATTR_NONE**—No constraint attributes are specified. This is the default value.
- **PRO_ASM_CONSTR_ATTR_INTFC_DEPENDENT**—When set in a component interface, the constraint cannot be changed by application of settings making it coincident, offset, or reoriented.
- **PRO_ASM_CONSTR_ATTR_INACTIVE**—The constraint should not be applied to the feature. This corresponds to the **Enabled** checkbox in the component feature user interface.

The function **ProAsmcompconstraintUserdataGet()** retrieves the user data for the given constraint while the function **ProAsmcompconstraintUserdataSet()** specifies the user data for the given constraint.

Use the function **ProAsmcompconstraintFree()** to free the constraint data structure from the memory.

The function **ProAsmcompconstraintArrayFree()** provides a single function to use to free an entire *ProArray* of *ProAsmcompconstraint* structures.

Example 1: Component Constraints

This function displays each constraint of the component visually on the screen, and includes a text explanation for each constraint.

```
#include <ProToolkit.h>
```

```
#include <ProAsmcomp.h>
#include <ProMessage.h>
#include <ProSolid.h>
#include <ProArray.h>
#include <ProPopupMenu.h>
#include <ProMenuBar.h>
#include <ProSelection.h>
#include <PTApplsUnicodeUtils.h>
#include <TestError.h>

#define MSGFIL L##"msg_ugasmcomp.txt"

/*-----*\
  Prototypes
\*-----*/
ProError UserAsmcompConstraintsHighlight (ProAsmcomp* asmcomp);
ProError UserOpenComponent (ProSolid* solid);

void UserConstraintTypeToString (ProAsmcompConstrType type,
                                char* type_string);

/*=====*\
  FUNCTION: UserAsmcompConstraintsHighlight
  PURPOSE: Highlights and labels a component's constraints
\*=====*/
ProError UserAsmcompConstraintsHighlight (ProAsmcomp* asmcomp)
{
  ProError status, aref_status, cref_status;
  ProSelection comp_constr, asm_constr;
  ProDatumside dtmside;
  ProAsmcompconstraint* constr_array;
  int size, i, i_count;
  double offset;
  char offset_str [20];
  ProLine entry;
  char type_str [20];
  ProAsmcompConstrType type;
/*-----*\
  Get the constraints for the component.
\*-----*/
  status = ProAsmcompConstraintsGet (asmcomp, &constr_array);
  if (status != PRO_TK_NO_ERROR)
    return PRO_TK_E_NOT_FOUND;

  ProArraySizeGet (constr_array, &size);

  for (i = 0; i < size; i++)
  {
/*-----*\
    Highlight the assembly reference geometry
\*-----*/
```

```

    aref_status = ProAsmcompconstraintAsmreferenceGet (constr_array [i],
                                                    &asm_constr, &dtmside);

    if (aref_status == PRO_TK_NO_ERROR)
    {
        ProSelectionHighlight (asm_constr, PRO_COLOR_ERROR);
    }
/*-----*\
Highlight the component reference geometry
\*-----*/
    cref_status = ProAsmcompconstraintCompreferenceGet(constr_array [i],
                                                    &comp_constr, &dtmside);

    if (cref_status == PRO_TK_NO_ERROR)
    {
        ProSelectionHighlight (comp_constr, PRO_COLOR_WARNING);
    }
/*-----*\
Prepare and display the message text.
\*-----*/
    status = ProAsmcompconstraintOffsetGet (constr_array [i], &offset);

    if (status == PRO_TK_NO_ERROR)
    {
        ProTKSprintf (offset_str, " offset = %6.3lf", offset);
    }
    else
        strcpy (offset_str, "");

    status = ProAsmcompconstraintTypeGet (constr_array [i], &type);
    UserConstraintTypeToString (type, type_str);

    i_count = i + 1;
    ProMessageDisplay (MSGFIL, "USER Showing constraint %0d of %1d
                               %2s%3s.Hit <CR> to continue.",
                      &i_count, &size, type_str, offset_str);
    ProMessageStringRead (1, entry);

/*-----*\
Clean up the UI for the next constraint
\*-----*/
    if (aref_status == PRO_TK_NO_ERROR)
    {
        ProSelectionUnhighlight (asm_constr);
    }

    if (cref_status == PRO_TK_NO_ERROR)
    {
        ProSelectionUnhighlight (comp_constr);
    }
}

```

```

    ProArrayFree ((ProArray*)&constr_array);

    return PRO_TK_NO_ERROR;
}
/*=====*\
FUNCTION: UserAsmcompConstraintsInfo
PURPOSE: Menu action for constraint information gathering
\*=====*/
ProError UserAsmcompConstraintsInfo ()
{
    ProError status;
    ProSelection* component_arr;
    int size;
    ProAsmcomp cfeat;

    ProMessageDisplay (MSGFIL, "USER Select an assembly component:");
    status = ProSelect ("membfeat", 1, NULL, NULL, NULL, NULL,
        &component_arr, &size);
    if (status != PRO_TK_NO_ERROR)
        return status;

    ProSelectionModelitemGet (component_arr [0], (ProModelitem*)&cfeat);

    ProSelectionUnhighlight (component_arr [0]);

    ProWindowRepaint (-1);

    UserAsmcompConstraintsHighlight (&cfeat);

    return PRO_TK_NO_ERROR;
}

/*=====*\
FUNCTION: UserConstraintTypeToString
PURPOSE: Utility: convert the constraint type to a string for printing
\*=====*/
void UserConstraintTypeToString (ProAsmcompConstrType type,
    char* type_string)
{
    switch (type)
    {
        case PRO_ASM_MATE:
            strcpy (type_string, "(Mate)");
            break;

        case PRO_ASM_MATE_OFF:
            strcpy (type_string, "(Mate Offset)");
            break;
    }
}

```

```

case PRO_ASM_ALIGN:
strcpy (type_string, "(Align)");
break;

case PRO_ASM_ALIGN_OFF:
strcpy (type_string, "(Align Offset)");
break;

case PRO_ASM_INSERT:
strcpy (type_string, "(Insert)");
break;

case PRO_ASM_ORIENT:
strcpy (type_string, "(Orient)");
break;

case PRO_ASM_CSYS:
strcpy (type_string, "(Csys)");
break;

case PRO_ASM_TANGENT:
strcpy (type_string, "(Tangent)");
break;

case PRO_ASM_PNT_ON_SRF:
strcpy (type_string, "(Point on Surf)");
break;

case PRO_ASM_EDGE_ON_SRF:
strcpy (type_string, "(Edge on Surf)");
break;

case PRO_ASM_DEF_PLACEMENT:
strcpy (type_string, "(Default)");
break;

case PRO_ASM_SUBSTITUTE:
strcpy (type_string, "(Substitute)");
break;

case PRO_ASM_PNT_ON_LINE:
strcpy (type_string, "(Point on Line)");
break;

case PRO_ASM_FIX:
strcpy (type_string, "(Fix)");
break;

case PRO_ASM_AUTO:
strcpy (type_string, "(Auto)");
break;

```



```

default:
strcpy (type_string, "(Unrecognized Type)");
break;
}
}

```

Example 2: Assembling Components

The following example demonstrates how to assemble a component into an assembly, and how to constrain the component by aligning datum planes. If the complete set of datum planes is not found, the function will show the component constraint dialog to the user to allow them to adjust the placement as they wish.

```

#include <ProToolkit.h>
#include <ProAsmcomp.h>
#include <ProMessage.h>
#include <ProSolid.h>
#include <ProArray.h>
#include <ProPopupMenu.h>
#include <ProMenuBar.h>
#include <ProSelection.h>
#include <PTAppUnicodeUtils.h>
#include <ProModelitem.h>
#include <ProUtil.h>
#include <UtilFiles.h>
#include <TestError.h>
#include <ProSelbuffer.h>

#define MSGFIL L##"msg_ugasmcomp.txt"

/*-----*\
    Prototypes
\*-----*/

ProError UserAsmcompConstraintsHighlight (ProAsmcomp* asmcomp);
ProError UserOpenComponent (ProSolid* solid);

void UserConstraintTypeToString (ProAsmcompConstrType type,
                                char* type_string);
/*=====*\
    FUNCTION: UserAssembleByDatums
    PURPOSE:  Assemble a component by aligning named datums.
\*=====*/

ProError UserAssembleByDatums(ProAssembly asm_model, ProSolid
                              comp_model)
{
    ProError status;
    ProName asm_datums [3];

```

```

ProName comp_datums [3];
ProMatrix identity_matrix = {{ 1.0, 0.0, 0.0, 0.0 },
                             {0.0, 1.0, 0.0, 0.0},
                             {0.0, 0.0, 1.0, 0.0},
                             {0.0, 0.0, 0.0, 1.0}};

ProAsmcomp asmcomp;
ProAsmcompconstraint* constraints;
ProAsmcompconstraint constraint;
int i;
ProBoolean interact_flag = PRO_B_FALSE;
ProModelitem asm_datum, comp_datum;
ProSelection asm_sel, comp_sel;
ProAsmcomppath comp_path;
ProIdTable c_id_table;
c_id_table [0] = -1;

/*-----*\
Set up the arrays of datum names
\*-----*/
ProStringToWstring (asm_datums [0], "ASM_D_FRONT");
ProStringToWstring (asm_datums [1], "ASM_D_TOP");
ProStringToWstring (asm_datums [2], "ASM_D_RIGHT");

ProStringToWstring (comp_datums [0], "COMP_D_FRONT");
ProStringToWstring (comp_datums [1], "COMP_D_TOP");
ProStringToWstring (comp_datums [2], "COMP_D_RIGHT");
/*-----*\
Package the component initially
\*-----*/
ProAsmcompAssemble (asm_model, comp_model, identity_matrix,
                    &asmcomp);

/*-----*\
Prepare the constraints array
\*-----*/
ProArrayAlloc (0, sizeof (ProAsmcompconstraint), 1,
               (ProArray*)&constraints);

for (i = 0; i < 3; i++)
{
/*-----*\
Find the assembly datum
\*-----*/
status = ProModelitemByNameInit (asm_model, PRO_SURFACE,
                                asm_datums [i], &asm_datum);
if (status != PRO_TK_NO_ERROR)
{
interact_flag = PRO_B_TRUE;
continue;
}
ProTKPrintf ("Item type = %d, PRO_SURFACE = %d\n",

```

```

        asm_datum.type, PRO_SURFACE);
    fflush (stdout);

/*-----*\
    Find the component datum
\*-----*/
    status = ProModelitemByNameInit (comp_model, PRO_SURFACE,
                                     comp_datums [i], &comp_datum);

    if (status != PRO_TK_NO_ERROR)
    {
        interact_flag = PRO_B_TRUE;
        continue;
    }
    ProTKPrintf ("Item type = %d, PRO_SURFACE = %d\n",
                 comp_datum.type, PRO_SURFACE);
    fflush (stdout);

/*-----*\
    For the assembly reference, initialize a component path.
    This is necessary even if the reference geometry is in the
    assembly.
\*-----*/
    status = ProAsmcomppathInit (asm_model, c_id_table, 0,
                                &comp_path);
    ERROR_CHECK ("ProAsmcomppathInit()", "UserAssembleByDatums",
                 status);

/*-----*\
    Allocate the references
\*-----*/
    status = ProSelectionAlloc (&comp_path, &asm_datum, &asm_sel);
    status = ProSelectionAlloc (NULL, &comp_datum, &comp_sel);

/*-----*\
    Allocate and fill the constraint.
\*-----*/
    status = ProAsmcompconstraintAlloc (&constraint);

    status = ProAsmcompconstraintTypeSet (constraint, PRO_ASM_ALIGN);

    status = ProAsmcompconstraintAsmreferenceSet (constraint, asm_sel,
                                                  PRO_DATUM_SIDE_YELLOW);

    status = ProAsmcompconstraintCompreferenceSet (constraint, comp_sel,
                                                  PRO_DATUM_SIDE_YELLOW);

    status = ProArrayObjectAdd ((ProArray*)&constraints,
                                -1, 1, &constraint);
}

/*-----*\
    Set the assembly component constraints and regenerate the assembly.
\*-----*/

```

```

\*-----*/

    status = ProAsmcompConstraintsSet (NULL, &asmcomp, constraints);
    ERROR_CHECK ("ProAsmcompConstraintsSet()", "UserAssembleByDatums",
                status);

    ProSolidRegenerate (asmcomp.owner, PRO_REGEN_CAN_FIX);
/*-----*\
    If any of the expect datums was not found, prompt the user
    to constrain the new component.
\*-----*/
    if (interact_flag)
    {
        ProMessageDisplay (MSGFIL, "USER Unable to locate all standard
        datum references. New component is packaged.");

        ProAsmcompConstrRedefUI (&asmcomp);
    }
    return (PRO_TK_NO_ERROR);
}

/*=====*\
    FUNCTION: UserAssemble
    PURPOSE:  Menu action to assemble a component
\*=====*/
ProError UserAssemble ()
{
    ProAssembly top_level_asm;
    ProSolid comp_model;
    ProError status;

    ProMdlCurrentGet ((ProMdl*)&top_level_asm);

    status = UserOpenComponent (&comp_model);
    if (status != PRO_TK_NO_ERROR)
        return status;

    UserAssembleByDatums (top_level_asm, comp_model);

    ProWindowRepaint (-1);

    ProTreetoolRefresh (top_level_asm);

    return PRO_TK_NO_ERROR;
}
/*=====*\
    FUNCTION: UserOpenComponent
    PURPOSE:  Prompt the user to select a component.
\*=====*/
ProError UserOpenComponent (ProSolid* solid)
{

```

```

ProSolid comp;
ProError err, er1;
int version, win_id, sel_n, Oldwin_id;
ProCharLine line, str;
ProLine buff;
ProPath *path_arr, sel_path, def_path, cur_path;
ProName *path_lab_arr, w_name, w_type;
ProType type;

ProMessageDisplay(MSGFIL, "USER Choose new component to
                        assemble:");
/*-----*\
  Prompt the user to select the file.
\*-----*/
ProStringToWstring(buff, "*.prt,*.asm");
ProStringToWstring(def_path, ".");

err = ProArrayAlloc(0, sizeof(ProPath), 1, (ProArray*)&path_arr);

err = ProArrayAlloc(0, sizeof(ProPath), 1,
                    (ProArray*)&path_lab_arr);

err = ProFileOpen(NULL, buff, path_arr, path_lab_arr, def_path,
                  NULL, sel_path);

if (err == PRO_TK_NO_ERROR)
{
/*-----*\
  Parse the filename to locate the component model.
\*-----*/
err = ProFilenameParse(sel_path, def_path, w_name, w_type,
                      &version);

ProUtilWstrcpy(sel_path, def_path);
err = ProPathCreate(NULL, w_name, w_type, PRO_VALUE_UNUSED,
                    def_path);

ProWstringToString(str, def_path);
if( ProUtilConfirmNameType(str, line, &type) !=
    PRO_TK_BAD_INPUTS)
{
/*-----*\
  Load the component model.
\*-----*/
err = ProDirectoryCurrentGet(cur_path);

err = ProDirectoryChange(sel_path);

err = ProMdlRetrieve(w_name, (ProMdlType)type, (ProMdl*)&comp);

er1 = ProDirectoryChange(cur_path);

```

```

        if (err != PRO_TK_NO_ERROR)
        {
            ProMessageDisplay (MSGFIL, "USER Failed to retrieve %0s", str);
        }
    }
}

/*-----*\
Free allocated memory
\*-----*/
er1 = ProArrayFree((ProArray*)&path_arr);

er1 = ProArrayFree((ProArray*)&path_lab_arr);

if (err != PRO_TK_NO_ERROR)
    return (PRO_TK_BAD_CONTEXT);

*solid = comp;

return PRO_TK_NO_ERROR;
}

```

Redefining Components Interactively

The functions described in this section enable you to reroute previously assembled components, as in an interactive session of Pro/ENGINEER.

Functions Introduced:

- **ProAsmcompConstrRedefUI()**
- **ProAsmcompPackageMove()**

The function **ProAsmcompConstrRedefUI()** is intended for use in interactive Pro/TOOLKIT applications. This function displays the Pro/ENGINEER Component Placement dialog box, enabling you to redefine the constraints interactively. Control is given back to the Pro/TOOLKIT application when you select **OK** or **Cancel** and the dialog box is closed.

The function **ProAsmcompPackageMove()** supports interactive and programmatic packaging of components. The arguments to this function allow the user or the program to repackage an existing component. If the component is used for interactive purposes, the control is given back to the Pro/TOOLKIT application when you select **OK** or **Cancel** and the dialog box is closed.

Assembling Components by Element Tree

Assembly components are treated as features in Pro/ENGINEER, so it is logical to replace those dedicated functions by a feature element tree that provides the same functionality, but uses the existing Pro/TOOLKIT functions **ProFeatureCreate()**, **ProFeatureRedefine()** and **ProFeatureElemtreeExtract()**.

The Element Tree for an Assembly Component

The element tree for a component assembly is documented in the header file *ProAsmcomp.h* and is shown in the following figure.

Figure 47-1: Top-level Feature Element tree for Component Assembly

```
PRO_E_FEATURE_TREE
|
|--PRO_E_FEATURE_TYPE
|
|--PRO_E_COMPONENT_MODEL
|
|--PRO_E_COMPONENT_MISC_ATTR
|
|--PRO_E_COMPONENT_INIT_POS
|
|--PRO_E_COMP_PLACE_INTERFACE
|
|--PRO_E_COMPONENT_SETS
|
|--PRO_E_COMPONENT_CONSTRAINTS
|
|--PRO_E_COMPONENT_MOVEMENTS
```

Model

The element **PRO_E_COMPONENT_MODEL** defines the model to be assembled into the assembly.

Attributes

The element `PRO_E_MISC_ATTR` defines the type of component being assembled. You can also access this property through **ProAsmcompTypeGet()** and **ProAsmcompTypeSet()**.

Initial Position

The element `PRO_E_COMPONENT_INIT_POS` defines the absolute position of the component in the absence of any parametric constraints. Its value is a ProMatrix object describing the component position. If you supply this element but no `PRO_E_COMPONENT_CONSTRAINT` elements, the component is assembled as packaged at this location.

If you underconstrain the component, the value of this element is used in conjunction with the constraints to set the packaged position for the component.

Constraint Sets and Mechanism Connections

The element `PRO_E_COMPONENT_SETS` is an array of elements of type `PRO_E_COMPONENT_SET`. Each of these elements is a predefined set of constraints.

Although you can define a user-defined type of constraint set containing any type and combination of constraints, the set element is also important when used to create predefined types of sets in Pro/ENGINEER.

Mechanism uses predefined set types to represent connections. The connection can also contain elements describing mechanism motion axes stored for the connection.

Figure 47-2: PRO_E_COMPONENT_SET

```

PRO_E_COMPONENT_SET
|--PRO_E_COMPONENT_SET_ID
|--PRO_E_COMPONENT_SET_TYPE
|--PRO_E_COMPONENT_SET_NAME
|--PRO_E_COMPONENT_SET_MISC_ATTR
|--PRO_E_COMPONENT_SET_FLIP
|--PRO_E_COMPONENT_JAS_SETS
    |--PRO_E_COMPONENT_JAS_SET
        |--PRO_E_COMPONENT_JAS_ZERO_TYPE
        |--PRO_E_COMPONENT_JAS_REFS
            |--PRO_E_COMPONENT_JAS_ORANGE_REF
            |--PRO_E_COMPONENT_JAS_GREEN_REF
            |--PRO_E_COMPONENT_JAS_O_OFFSET_VAL
        |--PRO_E_COMPONENT_JAS_REGEN_VALUE_GROUP
            |--PRO_E_COMPONENT_JAS_REGEN_VALUE_FLAG
            |--PRO_E_COMPONENT_JAS_REGEN_VALUE
        |--PRO_E_COMPONENT_JAS_MIN_LIMIT
            |--PRO_E_COMPONENT_JAS_MIN_LIMIT_FLAG
            |--PRO_E_COMPONENT_JAS_MIN_LIMIT_VAL
            |--PRO_E_COMPONENT_JAS_MIN_LIMIT_REF
        |--PRO_E_COMPONENT_JAS_MAX_LIMIT
            |--PRO_E_COMPONENT_JAS_MAX_LIMIT_FLAG
            |--PRO_E_COMPONENT_JAS_MAX_LIMIT_VAL
            |--PRO_E_COMPONENT_JAS_MAX_LIMIT_REF
        |--PRO_E_COMPONENT_JAS_RESTITUTION
            |--PRO_E_COMPONENT_JAS_RESTITUTION_FLAG
            |--PRO_E_COMPONENT_JAS_RESTITUTION_COEF
        |--PRO_E_COMPONENT_JAS_FRICTION
            |--PRO_E_COMPONENT_JAS_FRICTION_FLAG
            |--PRO_E_COMPONENT_JAS_STATIC_FRIC_COEF
            |--PRO_E_COMPONENT_JAS_KINEM_FRIC_COEF
            |--PRO_E_COMPONENT_JAS_RADIUS_VALUE
    
```

Each component set contains the following elements:

- PRO_E_COMPONENT_SET_ID—Specifies the component set id. This value is generated automatically by Pro/ENGINEER upon creation of the set. The set ids should remain the same when redefining the component.
- PRO_E_COMPONENT_SET_TYPE—Specifies the component set type. The following table describes the valid values in details.

Component Set Type	Description
PRO_ASM_SET_TYPE_FIXED	A "Rigid" Mechanism connection: Connects two components so that they do not move relatively to each other.
PRO_ASM_SET_TYPE_PIN	A "Pin" Mechanism connection: Connects a component to a referenced axis so that the component rotates or moves along this axis with one degree of freedom.
PRO_ASM_SET_TYPE_SLIDER	A "Slider" Mechanism connection: Connects a component to a referenced axis so that the component moves along the axis with one degree of freedom.
PRO_ASM_SET_TYPE_CYLINDRICAL	A "Slider" Mechanism connection: Connects a component so that it moves along and rotates about a specific axis with two degrees of freedom.
PRO_ASM_SET_TYPE_PLANAR	A "Planar" Mechanism connection: Connects components so that they move in a plane relatively to each other with two degrees of freedom in the plane and one degree of freedom around an axis perpendicular to it.
PRO_ASM_SET_TYPE_BALL	A "Ball" Mechanism connection: Connects a component so that it can rotate in any direction with three degrees of freedom (360° rotation).

Component Set Type	Description
PRO_ASM_SET_TYPE_WELD	A "Weld" Mechanism connection: Connects a component to another so that they do not move relatively to each other.
PRO_ASM_SET_TYPE_BEARING	A "Bearing" Mechanism connection: A combination of Ball and Slider connections with four degrees of freedom.
PRO_ASM_SET_TYPE_GENERAL	A "General" Mechanism connection: Has one or two configurable constraints that are identical to those in a user-defined set.
PRO_ASM_SET_TYPE_6DOF	A "Six Degrees of Freedom" Mechanism connection: Does not affect the motion of the component in relation to the assembly because no constraints are applied.
PRO_ASM_SET_TYPE_SLOT	A "Slot" Mechanism connection: A point on a non straight trajectory. This connection has four degrees of freedom, where the point follows the trajectory in three directions.
PRO_ASM_SET_USER_DEFINED_TYPE	A user defined constraint set. Legacy components which do not have defined Mechanism connections will always use this type.

- PRO_E_COMPONENT_SET_NAME—Specifies the name of the component set.
- PRO_E_COMPONENT_SET_MISC_ATTR—Specifies the component set attributes. Currently, these attributes are limited to flags which enable or disable the set.
- PRO_E_COMPONENT_JAS_SETS is an array of compound elements of type PRO_E_COMPONENT_JAS_SET which indicate Joint Axis Set (JAS). It represents the motion axis settings for the Mechanism connection. It consists of the following elements—
 - PRO_E_COMPONENT_JAS_ZERO_TYPE—Specifies the type of motion represented by this motion axis element and the value is drawn from the following types:
 - PRO_AXIS_ZERO_TRANSLATE1

- PRO_AXIS_ZERO_TRANSLATE2
- PRO_AXIS_ZERO_TRANSLATE3
- PRO_AXIS_ZERO_ROTATION1
- PRO_AXIS_ZERO_ROTATION2
- PRO_AXIS_ZERO_ROTATION3
- PRO_AXIS_ZERO_SLOT
- PRO_E_COMPONENT_JAS_REFS—Specifies initial position references for the motion axis.
 - PRO_E_COMPONENT_JAS_ORANGE_REF—Specifies the component reference of the motion axis.
 - PRO_E_COMPONENT_JAS_GREEN_REF—Specifies the assembly reference of the motion axis.
 - PRO_E_COMPONENT_JAS_0_OFFSET_VAL—Specifies the zero offset value for the motion axis.
- PRO_E_COMPONENT_JAS_REGEN_VALUE_GROUP is a compound element specifying the motion axis regeneration options. The motion axis regeneration value is an offset value used when regenerating the model. This value determines the offset of the component being placed in the assembly from the zero reference during regeneration.
 - PRO_E_COMPONENT_JAS_REGEN_VALUE_FLAG—Specifies enabling or disabling the regeneration value. When you disable it, the motion axis becomes free of constraint.
 - PRO_E_COMPONENT_JAS_REGEN_VALUE—Specifies the value along this motion axis that will be used for regeneration of the position and orientation of the component.
- PRO_E_COMPONENT_JAS_MIN_LIMIT specifies the minimum limit for regeneration value. It consists of the following elements:
 - PRO_E_COMPONENT_JAS_MIN_LIMIT_FLAG—A boolean element indicating whether the minimum limit is applied to the component.
 - PRO_E_COMPONENT_JAS_MIN_LIMIT_VAL—The value for the minimum limit.

- PRO_E_COMPONENT_JAS_MIN_LIMIT_REF—A selected item which serves as the minimum limit instead of using an assigned value.
- PRO_E_COMPONENT_JAS_MAX_LIMIT—Specifies the maximum limit for regeneration value. It consists of the following elements:
 - PRO_E_COMPONENT_JAS_MAX_LIMIT_FLAG—A boolean element indicating if the maximum limit is applied to the component.
 - PRO_E_COMPONENT_JAS_MAX_LIMIT_VAL—The value for the maximum limit.
 - PRO_E_COMPONENT_JAS_MAX_LIMIT_REF—A selected item, which serves as the maximum limit instead of using an assigned value.
- PRO_E_COMPONENT_JAS_RESTITUTION—Specifies coefficient of restitution for the motion axis. It is a compound element and consists of the following elements:
 - PRO_E_COMPONENT_JAS_RESTITUTION_FLAG—A boolean element indicating if a coefficient of restitution is applied.
 - PRO_E_COMPONENT_JAS_RESTITUTION_COEF—Specifies the coefficient of restitution value.
- PRO_E_COMPONENT_JAS_FRICTION—Specifies coefficient of friction for the motion axis. It is a compound element and consists of the following elements:
 - PRO_E_COMPONENT_JAS_FRICTION_FLAG—A boolean element indicating if the coefficients of friction are applied.
 - PRO_E_COMPONENT_JAS_STATIC_FRIC_COEF—Specifies the coefficient of static friction value.
 - PRO_E_COMPONENT_JAS_KINEM_FRIC_COEF—Specifies the coefficient of kinetic friction value.
 - PRO_E_COMPONENT_JAS_RADIUS_VALUE—Specifies the contact radius value.

Placement Constraints

The element PRO_E_COMPONENT_CONSTRAINTS is an array of elements of type PRO_E_COMPONENT_CONSTRAINT, each representing a single component placement constraint.

Figure 47-3: PRO_E_COMPONENT_CONSTRAINT

```
PRO_E_COMPONENT_CONSTRAINT
|--PRO_E_COMPONENT_CONSTR_TYPE
|--PRO_E_COMPONENT_COMP_CONSTR_REF
|--PRO_E_COMPONENT_ASSEM_CONSTR_REF
|--PRO_E_COMPONENT_CONSTR_REF_OFFSET
|--PRO_E_COMPONENT_USER_DATA
|--PRO_E_COMPONENT_CONSTR_ATTR
|--PRO_E_COMPONENT_COMP_ORIENT
|--PRO_E_COMPONENT_ASSM_ORIENT
|--PRO_E_COMPONENT_CONSTR_SET_ID
|--PRO_E_COMPONENT_SLOT_EXTRA_CRV_REF
```

Each constraint element contains the following elements:

- PRO_E_COMPONENT_CONSTR_TYPE—See discussion regarding ProAsmcompconstraintTypeGet().
- PRO_E_COMPONENT_COMP_CONSTR_REF—Identifies the geometry item in the component referenced by the constraint. This element is of type *Selection*.
- PRO_E_COMPONENT_ASSEM_CONSTR_REF—Identifies the constraint reference in the assembly.
Note: this reference must include a component path referencing the top level assembly, even if the reference belongs directly to the top level assembly.
- PRO_E_COMPONENT_CONSTR_REF_OFFSET—Gives the offset value, if the constraint type is an offset.
- PRO_E_COMPONENT_USER_DATA—Specifies user data.
- PRO_E_COMPONENT_CONSTR_ATTR—See discussion under ProAsmcompconstraintAttributesGet().
- The elements PRO_E_COMPONENT_COMP_ORIENT and PRO_E_COMPONENT_ASSM_ORIENT indicate which side of a referenced surface to be used. These values have different meanings for user-defined constraint sets and mechanism connections. See discussion under ProAsmcompconstraintAsmreferenceGet() and ProAsmcompconstraintCompreferenceGet().

- PRO_E_COMPONENT_CONSTR_SET_ID—Specifies the index of the member of the array of PRO_E_COMPONENT_SET elements that owns the constraint.
- PRO_E_COMPONENT_SLOT_EXTRA_CRV_REF—Specifies the extra curve references used by a Slot connection only. This is a multivalued element.

Component Movement in Assembly

The element PRO_E_COMPONENT_MOVEMENTS is an array of elements of type PRO_E_COMPONENT_MOVEMENT which represent movements applied to the component being assembled.

Figure 47-4: PRO_E_COMPONENT_MOVEMENT

PRO_E_COMPONENT MOVEMENT --PRO_E_COMPONENT MOVEMENT TYPE --PRO_E_COMPONENT MOVEMENT REF --PRO_E_COMPONENT MOVEMENT VALUE

Each movement element contains the following elements:

- PRO_E_COMPONENT_MOVEMENT_TYPE specifies allowed movement types

Use the following options to move the component parallel to the reference selected and are of the following values:

- PRO_ASM_TRANSLATE_X
- PRO_ASM_TRANSLATE_Y
- PRO_ASM_TRANSLATE_Z

Use the following options to rotate the component about the selected references.

- PRO_ASM_ROTATE_X
- PRO_ASM_ROTATE_Y
- PRO_ASM_ROTATE_Z
- PRO_ASM_TWIST_FIT
- PRO_E_COMPONENT_MOVEMENT_REF specifies the translational and rotational motion references.
- PRO_E_COMPONENT_MOVEMENT_VALUE

Placement via Interface

The element PRO_E_COMP_PLACE_INTERFACE is a compound element that defines an alternative assembly technique: using component interfaces to define the placement instead of traditional constraints.

Figure 47-5: PRO_E_COMP_PLACE_INTERFACE

```
PRO_E_COMP_PLACE_INTERFACE
|--PRO_E_COMP_PLACE_INTERFACE_TYPE
|--PRO_E_COMP_PLACE_INTERFACE_COMP
|--PRO_E_COMP_PLACE_INTERFACE_ASSEMS
    |--PRO_E_COMP_PLACE_INTERFACE_ASSEM
        |--PRO_E_COMP_PLACE_INTERFACE_ASSEM_REF
```

The element contains the following elements:

- PRO_E_COMP_PLACE_INTERFACE_TYPE—Specifies the interface types as follows:
 - PRO_ASM_INTFC_TO_GEOM—Assembly of the component by matching an interface on the component to referenced geometry in the assembly.
 - PRO_ASM_INTFC_TO_INTFC—Assembly of the component by matching an interface on the component to an interface defined in the assembly.
- Note:** If this value is not set (set to 0) then component interfaces are not used to define this component.
- PRO_E_COMP_PLACE_INTERFACE_COMP—Specifies the component model interface. This should contain the component interface feature.
 - PRO_E_COMP_PLACE_INTERFACE_ASSEMS—Specifies an array of assembly references. If the placement type is PRO_ASM_INTFC_TO_GEOM this contains 1 or more geometric references from the assembly. If the placement type is PRO_ASM_INTFC_TO_INTFC this contains a single reference element containing the component interface feature.

Assembling Components Using Intent Datums

You can use Intent Datums such as Intent Axis, Intent Point, Intent Plane, and Intent Coordinate System in assembly component placement and constraints.

Example 3: Assembling Components Using Intent Datums

The following example demonstrates how to assemble a component into an assembly using intent point and intent plane, and how to set constraints after assembling a component using intents.

```
#include <ProToolkit.h>
#include <ProAsmcomp.h>
#include <ProMessage.h>
#include <ProSolid.h>
#include <ProArray.h>
#include <ProPopupMenu.h>
#include <ProMenuBar.h>
#include <ProSelection.h>
#include <PTApplsUnicodeUtils.h>
#include <ProModelitem.h>
#include <ProUtil.h>
#include <UtilFiles.h>
#include <TestError.h>
#include <ProSelbuffer.h>

#define MSGFIL L##"msg_ugasmcomp.txt"

/*-----*\
  Prototypes
\*-----*/
ProError UserAsmcompConstraintsHighlight (ProAsmcomp* asmcomp);
ProError UserOpenComponent (ProSolid* solid);

void UserConstraintTypeToString (ProAsmcompConstrType type,
  char* type_string);
ProError UgSelectAsmIntent(ProAssembly asm_model, ProSolid comp_model,
  ProType constr_type, ProAsmcompconstraint** constraints);

/*=====*\
  FUNCTION: UgSelectAsmIntent
  PURPOSE: Assemble a component using intent point and intent plane
\*=====*/
ProError UgSelectAsmIntent(ProAssembly asm_model, ProSolid comp_model,
  ProType constr_type,
  ProAsmcompconstraint** constraints)
{
  ProError status;
  ProAsmcompconstraint constraint;
```

```

int i, n_sels;
int asm_point,asm_plane,comp_point,comp_plane;
int constr_point,constr_plane;
ProModelitem asm_datum, comp_datum;
ProSelection asm_sel, comp_sel;
ProAsmcomppath comp_path;
ProSelection* sels = NULL, *sels2 = NULL;
ProName selmdl_name, compmdl_name;
ProCharName c_selmdl_name, c_compmdl_name;
ProModelitem p_asmintent, p_compintent;

/*-----*\
  Get the intent datum point for the component
  \*-----*/
while (1)
{
  /*-----*\
    Find the component intent
    \*-----*/
  if(constr_type == PRO_LOG_PNT )
    ProMessageDisplay (MSGFIL,"Select the intent point in component");

  else if(constr_type == PRO_LOG_PLANE)
    ProMessageDisplay (MSGFIL,"Select the intent plane in component");

  status = ProSelect ("membfeat", 1, NULL, NULL, NULL, NULL,
    &sels, &n_sels);
  if ((status != PRO_TK_NO_ERROR) || (n_sels < 1))
    return status;

  status = ProSelectionModelitemGet (sels[0], &comp_datum);
  if ((status != PRO_TK_NO_ERROR) || (comp_datum.type != PRO_FEATURE))
    return status;

  status = ProMdlNameGet(comp_datum.owner, selmdl_name);
  ProWstringToString (c_selmdl_name, selmdl_name);

  status = ProMdlNameGet(comp_model, compmdl_name);
  ProWstringToString (c_compmdl_name, compmdl_name);

  if(strcmp(c_compmdl_name,c_selmdl_name) != 0)
    continue;

  status = ProModelitemInit(comp_datum.owner,(comp_datum.id+1),
    constr_type,&p_compintent);
  if (status == PRO_TK_NO_ERROR)
    break;
}

while(1)

```

```

{
/*-----*\
Find the assembly intent
\*-----*/

if(constr_type == PRO_LOG_PNT )
    ProMessageDisplay (MSGFIL,"Select the intent point in assembly");

else if(constr_type == PRO_LOG_PLANE)
    ProMessageDisplay (MSGFIL,"Select the intent plane in assembly");

status = ProSelect ("membfeat", 1, NULL, NULL, NULL, NULL, &sels2,
                    &n_sels);
if ((status != PRO_TK_NO_ERROR) || (n_sels < 1))
    return status;

status = ProSelectionModelitemGet (sels2[0], &asm_datum);
if ((status != PRO_TK_NO_ERROR) || (asm_datum.type != PRO_FEATURE))
    return status;

status = ProSelectionAsmcomppathGet(sels2[0], &comp_path);
if (status != PRO_TK_NO_ERROR)
    continue;

status = ProModelitemInit(asm_datum.owner, (asm_datum.id+1),
                          constr_type, &p_asmintent);
if (status == PRO_TK_NO_ERROR)
    break;
}

/*-----*\
Allocate the references
\*-----*/
status = ProSelectionAlloc (&comp_path, &p_asmintent, &asm_sel);
status = ProSelectionAlloc (NULL, &p_compintent, &comp_sel);

/*-----*\
Allocate and fill the constraint.
\*-----*/
status = ProAsmcompconstraintAlloc (&constraint);
if(constr_type == PRO_LOG_PNT)
    ProAsmcompconstraintTypeSet (constraint, PRO_ASM_ALIGN);

else if(constr_type == PRO_LOG_PLANE)
    ProAsmcompconstraintTypeSet (constraint, PRO_ASM_ORIENT);

status = ProAsmcompconstraintAsmreferenceSet (constraint, asm_sel,
                                              PRO_DATUM_SIDE_YELLOW);

status = ProAsmcompconstraintCompferenceSet (constraint, comp_sel,

```

```

RO_DATUM_SIDE_YELLOW);

status = ProArrayObjectAdd ((ProArray*)constraints, -1, 1,
                             &constraint);
}

/*=====*\
FUNCTION: UgAssembleByIntent
PURPOSE: Set constraints after assembling a component using intents
\*=====*/
ProError UgAssembleByIntent (ProAssembly asm_model, ProSolid comp_model)
{
    ProError status;
    ProMatrix identity_matrix = {{ 1.0, 0.0, 0.0, 0.0 },
                                   {0.0, 1.0, 0.0, 0.0},
                                   {0.0, 0.0, 1.0, 0.0},
                                   {0.0, 0.0, 0.0, 1.0}};

    ProAsmcomp asmcomp;
    ProAsmcompconstraint* constraints;
    ProAsmcompconstraint constraint;

    /*-----*\
    Package the component initially
    \*-----*/
    status = ProAsmcompAssemble (asm_model, comp_model,
                                   identity_matrix, &asmcomp);

    /*-----*\
    Prepare the constraints array
    \*-----*/
    status = ProArrayAlloc (0, sizeof (ProAsmcompconstraint), 1,
                            (ProArray*)&constraints);

    /*-----*\
    Set PRO_LOG_PNT constraint in assembly
    \*-----*/
    status = UgSelectAsmIntent(asm_model, comp_model, PRO_LOG_PNT,
                                &constraints);

    /*-----*\
    Set PRO_LOG_PLANE constraint in assembly
    \*-----*/
    status = UgSelectAsmIntent(asm_model, comp_model, PRO_LOG_PLANE,
                                &constraints);

    /*-----*\
    Set the assembly component constraints and regenerate the assembly.
    \*-----*/
    status = ProAsmcompConstraintsSet (NULL, &asmcomp, constraints);

    if(status == PRO_TK_NO_ERROR)

```

```

        ProMessageDisplay (MSGFIL,"Component assembled successfully");

    else
        return status;

    ProSolidRegenerate (asmcomp.owner, PRO_REGEN_CAN_FIX);

    return (PRO_TK_NO_ERROR);
}

/*=====*\
FUNCTION: UgAssembleIntent
PURPOSE:  Menu action to assemble a component using intent feature
\*=====*/
ProError UgAssembleIntent ()
{
    ProAssembly top_level_asm = NULL;
    ProSolid comp_model = NULL;
    ProError status;

    status = ProMdlCurrentGet ((ProMdl*)&top_level_asm);
    if((status != PRO_TK_NO_ERROR) || (top_level_asm == NULL))
    {
        ProMessageDisplay (MSGFIL, "Failed to open the assembly");
        return status;
    }

    status = UserOpenComponent (&comp_model);
    if ((status != PRO_TK_NO_ERROR) || comp_model == NULL )
    {
        ProMessageDisplay (MSGFIL, "Failed to open the component");
        return status;
    }

    if(top_level_asm && comp_model)
        status = UgAssembleByIntent (top_level_asm, comp_model);

    ProWindowRepaint (-1);

    ProTreetoolRefresh (top_level_asm);

    return PRO_TK_NO_ERROR;
}

```


48

Assembly: Kinematic Dragging and Creating Snapshots

This chapter describes the Pro/TOOLKIT functions for dragging assembly entities through an allowable range to see how your assembly works in a specific configuration. These functions provide the same result as obtained by using the **Kinematic Drag** dialog box in the Pro/ENGINEER user interface.

This chapter also describes the functions to create snapshots of the assembly in different positions and orientations.

Topic	Page
Connecting to a Kinematic Drag Session	48 - 2
Performing Kinematic Drag	48 - 4
Creating and Modifying Snapshots	48 - 5
Snapshot Constraints	48 - 5
Snapshot Transforms	48 - 9
Snapshots in Drawing Views	48 - 9

Connecting to a Kinematic Drag Session

Functions Introduced:

- **ProKinDragStart()**
- **ProKinDragStop()**
- **ProKinDragSessionInquire()**
- **ProKinDragEnvironmentSet()**
- **ProKinDragReconnect()**

The function **ProKinDragStart()** starts a drag session. This function should be called before any other function for Kinematic drag or snapshots. All subsequent functions work with the snapshots of the top-level assembly. This function can be called for design assemblies in the Standard Assembly and Mechanism modes only. It cannot be used in the following cases:

- If the assembly is in the exploded state
- If the Sketcher mode is active
- When component placement is active
- When the Drag dialog box is active

The function **ProKinDragStop()** stops the drag session. Use **ProWindowRefresh()** to view the change in the graphic window scale according to the updated outline.

The function **ProKinDragSessionInquire()** determines if the drag session is active. The drag session may exist automatically in some cases, such as redefine, regeneration, window switch, representation mode change.

The function **ProKinDragEnvironmentSet()** sets the environment for dragging and reconnect operations. It requires the following input arguments:

- *snap_name*—Specifies the name of the active snapshot whose constraints are used for the drag operation. The snapshot is checked for statuses such as good, outdated, or incomplete. Pass NULL if not required.

The *outdated* status means some parts from the body have a relative transformation that is different from the current transformation. The *incomplete* status means that some parts from the body are missing in the active snapshot.

- *path*—Specifies the path in terms of the *ProAsmcomppath* object to the active snapshot of a subassembly contained within the top-level assembly. Pass `NULL` for a top-level assembly snapshot.
- *type*—Specifies the type of drag to be performed. It is given by the enumerated type *ProKinDragType* that takes the following values:
 - `PRO_KIN_POINT_DRAG`—Select a point to drag in a part within the top-level assembly. During the dragging operation, the point you selected follows the pointer's movement while maintaining connections.
 - `PRO_KIN_BODY_DRAG`—Select a part from the top-level assembly to drag. When you drag a part, its position in the graphics window changes, but its orientation remains fixed. If the assembly requires a part to be reoriented in conjunction with a change in position, the part does not move at all, as the assembly cannot reassemble in the new position.
 - `PRO_KIN_ADVANCED_TRANS_X`—Specifies the translation in the X direction of the selected coordinate system. This type is applicable only in case of an advanced drag operation. You can select a coordinate system by selecting the part to be dragged.
 - `PRO_KIN_ADVANCED_TRANS_Y`—Specifies the translation in the Y direction of the selected coordinate system. This type is applicable only in case of an advanced drag operation.
 - `PRO_KIN_ADVANCED_TRANS_Z`—Specifies the translation in the Z direction of the selected coordinate system. This type is applicable only in case of an advanced drag operation.
 - `PRO_KIN_ADVANCED_ROT_X`—Specifies the rotation around the X axis of the selected coordinate system. This type is applicable only in case of an advanced drag operation. You can select a coordinate system by selecting the part to be dragged.
 - `PRO_KIN_ADVANCED_ROT_Y`—Specifies the rotation around the Y axis of the selected coordinate system. This type is applicable only in case of an advanced drag operation.

- **PRO_KIN_ADVANCED_ROT_Z**—Specifies the rotation around the Z axis of the selected coordinate system. This type is applicable only in case of an advanced drag operation.
- *refobject*—Specifies the selection reference in the form a coordinate system (**PRO_CSYS**), part (**PRO_PART**), or mechanism body (**PRO_MDO_BODY**) for an advanced drag operation. This argument is relevant only for advanced drag types. For all other drag types, it is ignored and **NULL** should be passed.

The function **ProKinDragReconnect()** reconnects to a drag session taking into account constraints but not transforms from the active snapshot specified in **ProKinDragEnvironmentSet()**. Use **ProWindowRefresh()** to view the changes in the positions of the assembly components. Refer to the sections “Snapshot Constraints” and “Snapshot Transforms” for more information.

Performing Kinematic Drag

Function Introduced:

- **ProKinDragPerformMove()**

The function **ProKinDragPerformMove()** drags the selected geometric object to the specified X and Y screen coordinates given by *ProArray* of *Pro2dPnt* object. The geometric object can be **PRO_POINT**, **PRO_AXIS**, **PRO_EDGE**, **PRO_CURVE**, **PRO_DATUM_PLANE**, or **PRO_SURFACE** in case of a point drag, or **PRO_PART** or **PRO_MDO_BODY** for all other drag types. A hook point is displayed on the object selected for drag, which is removed at the end of the drag operation. The dragging is performed according to the environment set by the function **ProKinDragEnvironmentSet()**. Use the constraints set for the active snapshot while dragging.

Creating and Modifying Snapshots

Functions Introduced:

- **ProKinDragSnapshotsNamesGet()**
- **ProSnapshotCreate()**
- **ProSnapshotUpdate()**
- **ProSnapshotRename()**
- **ProSnapshotDelete()**
- **ProSnapshotApply()**

The function **ProKinDragSnapshotsNamesGet()** retrieves an array of names of the snapshots belonging to the top-level assembly and all its subassemblies.

The function **ProSnapshotCreate()** creates a new snapshot for the top-level assembly as per its current position in the Pro/ENGINEER window. Constraints are copied from the active snapshot. The newly created snapshot becomes active.

The function **ProSnapshotUpdate()** updates the snapshot for the top-level assembly as per its current position in the Pro/ENGINEER window. Constraints are copied from the active snapshot. The newly updated snapshot becomes active.

The function **ProSnapshotRename()** renames the active snapshot of the top-level assembly.

The function **ProSnapshotDelete()** deletes a specified snapshot. The snapshot can be of the top-level assembly or any of its subassemblies. Pass `NULL` for the argument *path* to specify the top-level assembly.

The function **ProSnapshotApply()** applies the transforms of the active snapshot. Unlike in the user interface, no attempt to reconnect is made. Call **ProKinDragReconnect()** if reconnect is necessary. Use **ProWindowRefresh()** to view the changes in the positions of the assembly components.

Snapshot Constraints

The constraints that can be applied to a snapshot are given by the structure *ProSnapshotConstraint*. The declaration for *ProSnapshotConstraint* is as follows:

```
typedef struct proSnapshotConstraint
{
ProSnapshotConstraintType  type;
ProSelection               *sel_array;
double                    value;
ProBool                   user_active;
ProBool                   valid
} ProSnapshotConstraint;
```

The fields in the above structure are described as follows:

- **type**—Specifies the type of snapshot constraint. It is given by the enumerated type *ProSnapshotConstraintType* and can take the following values:
 - **PRO_SNAP_ALIGN**—Select two points, two lines, or two planes from the top-level assembly. The two entities remain aligned during the drag operation.
 - **PRO_SNAP_MATE**—Select two planes from the top-level assembly. The planes remain mated during the drag operation.
 - **PRO_SNAP_ORIENT**—Select two planes to orient at an angle to each other.
 - **PRO_SNAP_MOTION_AXIS_POS**—Select a motion axis to specify the motion axis position.
 - **PRO_SNAP_BODY_LOCK**—Select the bodies to be locked together while dragging.
 - **PRO_SNAP_CONNECTION_DISABLE**—Select a connection that will be disabled while dragging.
 - **PRO_SNAP_PARALLEL_VIEW_PLANE**—Select a body that will move parallel only to the view plane. This constraint type is the same as the **PRO_SNAP_ALIGN** or **PRO_SNAP_MATE** types, but the second reference in this case is the view plane. This constraint type is available only via Pro/TOOLKIT. If set by the Pro/TOOLKIT application, this constraint type becomes visible in the **Kinematic Drag** dialog box in the Pro/ENGINEER user interface, but it cannot be stored.
- ***sel_array**—Specifies the *ProArray* of selections. The number of selections needed and the permitted selection types for each constraint type is given by the following table:

Constraint Type	Number of Selections needed	Permitted Selection Types
PRO_SNAP_ALIGN	2	PRO_POINT, PRO_AXIS, PRO_EDGE, PRO_CURVE, PRO_DATUM_PLANE, PRO_SURFACE
PRO_SNAP_MATE		PRO_DATUM_PLANE, PRO_SURFACE
PRO_SNAP_ORIENT		
PRO_SNAP_MOTION_AXIS_POS	1	PRO_MDO_CONN_AXIS_ROT_1/2/3, PRO_MDO_CONN_AXIS_TR_1/2/3, PRO_MDO_SLOT_AXIS
PRO_SNAP_CONNECTION_DISABLE		PRO_MDO_CONN, PRO_MDO_CAM_CONN, PRO_MDO_SLOT_CONN, PRO_MDO_GEAR_CONN
PRO_SNAP_PARALLEL_VIEW_PLANE	1	PRO_POINT, PRO_EDGE_START, PRO_EDGE_END, PRO_CRV_START, PRO_CRV_END, PRO_AXIS, PRO_EDGE, PRO_CURVE, PRO_SURFACE, PRO_DATUM_PLANE
PRO_SNAP_BODY_LOCK	2 or more	PRO_PART, PRO_MDO_BODY

- **value**—Depending upon the constraint type, this field takes the following values:
 - For the PRO_SNAP_ALIGN and PRO_SNAP_MATE constraint types, **value** specifies the linear distance between the references.
 - For the PRO_SNAP_ORIENT type, **value** specifies the angle between the references.

- For the `PRO_SNAP_MOTION_AXIS_POS` type, `value` specifies the offset of the joint axis zero position. This value is angular for the rotation axis and linear for the translational axis. The references for the joint axis zero position may be default or as specified by the user.
- `active`—Specifies the *ProBoolean* option to enable or disable a constraint.
- `valid`—Specifies if the constraint is valid or invalid in the current model context. If this *ProBoolean* option is `PRO_B_TRUE`, the constraint is valid (active), and if it is `PRO_B_FALSE`, the constraint is invalid (suppressed).

Functions Introduced:

- **ProSnapshotConstraintsGet()**
- **ProSnapshotConstraintAdd()**
- **ProSnapshotConstraintDelete()**
- **ProSnapshotConstraintUpdate()**
- **ProSnapshotConstraintEvaluate()**

The function **ProSnapshotConstraintsGet()** retrieves all the constraints of a specified snapshot. The snapshot can be of the top-level assembly or any of its subassemblies. Pass `NULL` for the argument *path* to specify the top-level assembly.

The function **ProSnapshotConstraintAdd()** adds a constraint to the snapshot of the top-level assembly.

The function **ProSnapshotConstraintDelete()** deletes a constraint from the snapshot of the top-level assembly.

The function **ProSnapshotConstraintUpdate()** updates a constraint for the snapshot of the top-level assembly.

The function **ProSnapshotConstraintEvaluate()** calculates the position of the motion axis for the constraint type `PRO_SNAP_MOTION_AXIS_POS` for the active model in the Pro/ENGINEER window. This value does not depend on the current snapshot and it is not necessary for the snapshot to contain the constraint.

Snapshot Transforms

Functions Introduced:

- **ProSnapshotTrfsGet()**
- **ProSnapshotTrfsUpdate()**

The function **ProSnapshotTrfsGet()** retrieves the transformation paths and transformation matrices saved in the snapshots for the subassemblies and their components with respect to the top-level assembly snapshot.

The function **ProSnapshotTrfsUpdate()** updates the transformation matrices saved in the snapshots for the subassemblies and their components with respect to the top-level assembly snapshot.

Assembly: Kinematic
Dragging and

Snapshots in Drawing Views

Functions Introduced:

- **ProSnapshotAllowedInDrawingSet()**
- **ProSnapshotAllowedInDrawingGet()**

The function **ProSnapshotAllowedInDrawingSet()** assigns the active snapshot of the top-level assembly to be available or unavailable in drawings. Set the argument *allow* to `PRO_B_TRUE` to make the snapshot available in drawings.

The function **ProSnapshotAllowedInDrawingGet()** determines if the active snapshot is allowed in drawings. The argument *p_is_allowed* is `PRO_B_TRUE` if the snapshot is allowed in drawings.

49

Assembly: Simplified Representations

Pro/TOOLKIT gives programmatic access to all the simplified representation functionality of Pro/ENGINEER. You can create simplified representations either permanently or at runtime, and you can save, retrieve, or modify them by adding or deleting items.

Topic	Page
Overview	49 - 2
Simplified Representations in Session	49 - 3
Retrieving Simplified Representations	49 - 5
Creating and Deleting Simplified Representations	49 - 6
Extracting Information About Simplified Representations	49 - 9
Modifying Simplified Representations	49 - 14
Gathering Components by Rule	49 - 15

Overview

Using Pro/TOOLKIT, you can create and manipulate assembly simplified representations just as you can using Pro/ENGINEER interactively.

Note: Pro/TOOLKIT supports retrieval and activation of both part and assembly simplified representations. In addition, Pro/TOOLKIT supports creation and modification of assembly simplified representations. Functions not appropriate for part mode are identified in the description.

Simplified representations are identified by the DHandle *ProSimpdataprep*. As with other DHandles such as *ProFeature* and *ProGeomitem*, the *ProSimpdataprep* handle contains just enough information to uniquely identify the object in the database—the model owner, type, and identifier.

The information required to create and modify a simplified representation is stored in a series of *ProSimpdataprepdata* structures, which are visible data structures. The data structure contains the following fields:

- *ProName name*—The name of the simplified representation
- *ProBoolean temp*—Specifies whether it is a temporary, simplified representation
- *ProSimpdataprepActionType action_type*—The rule that controls the default treatment of items in the simplified representation
- *ProSimpdataprepitem *items*—An array of assembly components and features and the actions applied to them in the simplified representation

A *ProSimpdataprepitem* is identified by the *ProIdTable* that defines the assembly component path to that item. (Even if the ID table path is only one level, use the *ProIdTable* and not the feature id for assemblies). Each *ProSimpdataprepitem* has its own *ProSimpdataprepAction* assigned to it. *ProSimpdataprepAction* is a visible data structure that includes a variable of type *ProSimpdataprepActionType*.

ProSimpredActionType is an enumerated type that specifies the possible treatment of items in a simplified representation. The declaration is as follows:

```
typedef enum pro_simpred_actions
{
    PRO_SIMPREP_NONE,          /* No action is specified. */
    PRO_SIMPREP_REVERSE,       /* Reverse the default rule for this
                                component (for example, include it if the
                                default rule is exclude). */
    PRO_SIMPREP_INCLUDE,       /* Include this component in the simplified
                                representation. */
    PRO_SIMPREP_EXCLUDE,       /* Exclude this component from the
                                simplified representation. */
    PRO_SIMPREP_SUBSTITUTE,     /* Substitute the component in the
                                simplified representation. */
    PRO_SIMPREP_GEOM,          /* Use only the geometrical representation
                                of the component */
    PRO_SIMPREP_GRAPHICS,      /* Use only the graphics representation of
                                the component. */
    PRO_SIMPREP_SYMB,          /* Use only the symbolic representation of
                                the component. */
    PRO_SIMPREP_BOUNDBOX,      /* Use only the boundary box representation
                                of the component. */
    PRO_SIMPREP_DEFENV         /* Use only the envelope representation of
                                the component. */
} ProSimpredActionType;
```

Simplified Representations in Session

Functions Introduced:

- **ProSolidSimpredVisit()**
- **ProSimpredInit()**
- **ProSimpredSelect()**
- **ProSimpredActivate()**
- **ProSimpredActiveGet()**
- **ProSimpredTypeGet()**
- **ProSimpredIsDefault()**

This section describes the utility functions that relate to simplified representations.

ProSolidSimpredVisit() is like the other visit functions, and visits all the simplified representations of a parent *ProSolid*.

As all other visit functions, it takes four arguments—a pointer to the parent *ProSolid*, a filter function, the visit function itself, and a *ProAppData* field.

The function **ProSimprepInit()** initializes a *ProSimprep* structure. The function takes the following arguments:

- *ProName rep_name*—The name of the simplified representation in the solid. If you specify this argument, the function ignores the *rep_id*.
- *int rep_id*—The identifier of the simplified representation, if you did not specify *rep_name* (you specified NULL).
- *ProSolid p_solid*—The parent solid that contains the simplified representation.
- *ProSimprep p_simp_rep*—The handle to the newly initialized simplified representation.

The function **ProSimprepSelect()** creates a Pro/ENGINEER menu to enable interactive selection. The function takes the parent solid as input, and outputs the handle to the selected simplified representation. If you choose the **Quit** menu button, the function returns the value PRO_TK_USER_ABORT. If the user selects the master representation, the returned simplified representation structure has an identifier of -1.

ProSimprepActivate() enables you to set the currently active simplified representation. To set a simplified representation to be the currently displayed model, you must also call **ProSolidDisplay()**. This function enables you to bring inactive submodels into memory, and use their handles without displaying them.

ProSimprepActivate() does not support activation of part simplified representations, because part simplified representation handles cannot be passed to this function. To obtain a handle to a part simplified representation use **ProPartSimprepRetrieve()**. You can display the simplified representation in a window using **ProSolidDisplay()**.

ProSimprepActiveGet() enables you to find the currently active simplified representation. Given a model handle, **ProSimprepActiveGet()** returns the handle to the currently active simplified representation. If the current representation is the master representation, the identifier of the handle is set to -1.

The function **ProSimprepTypeGet()** outputs the type of a specified simplified representation. The type is expressed in terms of the enum *ProSimprepType* which has the following declaration:

```
typedef enum pro_simprep_types
{
    PRO_SIMPREP_MASTER_REP,      /* Master representation */

    PRO_SIMPREP_USER_DEFINED,    /* User Defined Simplified
                                representation */

    PRO_SIMPREP_GRAPH_REP,       /* Graphics only representation */

    PRO_SIMPREP_GEOM_REP         /* Geometry only representation */

    PRO_SIMPREP_SYMB_REP         /* Symbolic representation */

    PRO_SIMPREP_BOUNDBOX_REP     /* Boundary Box representation */

    PRO_SIMPREP_DEFENV_REP       /* Default Envelope representation */
} ProSimprepType;
```

The function **ProSimprepIsDefault()** determines if the specified simplified representation is the default representation for the owner model.

Retrieving Simplified Representations

Function Introduced:

- **ProAssemblySimprepRetrieve()**
- **ProGeomSimprepRetrieve()**
- **ProGraphicsSimprepRetrieve()**
- **ProSymbSimprepRetrieve()**
- **ProBoundaryBoxSimprepRetrieve()**
- **ProDefaultEnvelopeSimprepRetrieve()**
- **ProPartSimprepRetrieve()**

You can retrieve a named simplified representation from an assembly using the function **ProAssemblySimprepRetrieve()**. This function retrieves the handle of an existing simplified representation from an assembly without fetching the generic representation into memory.

The function takes four arguments—the names of the assembly and simplified representation, the representation data, and the handle to the assembly. Note that you must provide the name of the assembly. To retrieve an existing simplified representation, specify its name as the second argument to this function. The name of the simplified representation can be NULL if the representation data is provided. In this case, the instructions in the data are used to create a new simplified representation on the fly. The representation data can also be NULL if the name of the simplified representation is provided. Pro/ENGINEER retrieves the simplified representation and any active submodels, and returns the *ProAssembly* handle.

You can retrieve geometry, graphics, symbolic simplified, boundary box, and default envelope representations into session using the functions **ProGeomSimpRepRetrieve()**, **ProGraphicsSimpRepRetrieve()**, **ProSymbSimpRepRetrieve()**, **ProBoundaryBoxSimpRepRetrieve()**, and **ProDefaultEnvelopeSimpRepRetrieve()** respectively. Like **ProAssemblySimpRepRetrieve()**, these functions retrieve the simplified representation without bringing the master representation into memory. Supply the name of the assembly whose simplified representation is to be retrieved as the input argument for these functions. These functions output the handle to the assembly. None of the functions display the simplified representation.

You can retrieve part simplified representations into memory using the function **ProPartSimpRepRetrieve()**.

Creating and Deleting Simplified Representations

Functions Introduced:

- **ProSimpRepDataAlloc()**
- **ProSimpRepCreate()**
- **ProSimpRepDelete()**

Note: Pro/TOOLKIT does not support creation of part simplified representations.

To create a simplified representation, you must allocate and fill a *ProSimpdataprep* structure by calling the function **ProSimpdataprepAlloc()**. As input, the function requires the name of the new simplified representation, the *temp* value, and the default rule. The specific structure is initialized by the function in the Pro/ENGINEER database.

To generate the new simplified representation, call **ProSimpdataprepCreate()**. This function returns the *ProSimpdataprep* handle for the new representation.

The function **ProSimpdataprepDelete()** deletes a simplified representation from its model owner. The function requires only the *ProSimpdataprep* handle as input.

Example 1: Creating a Simplified Representation

The following example shows how to create a simplified representation with “Include” as the default rule. The user selects the components to be excluded.

```
/*=====*\
FUNCTION: UserSimpRepCreate
PURPOSE: Does the following:
    - Creates a simplified representation
    - Adds and delete some items
    - Modifies the simplified representation
\*=====*/
ProError UserSimpRepCreate()
{
    ProSimpdataprep      *rep;
    ProSimpdataprep      simp_rep;
    ProSimpdataprepAction sr_act;
    ProSimpdataprepitem  item;
    ProError             status;
    ProFileName          message_file;
    ProName              name;
    ProMdl               assembly;
    ProMdlType           type;
    ProSimpdataprepActionType action;
    ProAsmcomppath       comp_path;
    int                  i;
    int                  num;
    ProSelection          *sels;

    ProStringToWstring (message_file, "msg_ugfund.txt");
/*-----*\
    Get the current model and determine whether it is an assembly
\*-----*/
    status = ProMdlCurrentGet (&assembly);
```

```

if (status != PRO_TK_NO_ERROR)
{
    ProMessageDisplay (message_file, "USER %0s", "No valid model.");
    return (status);
}
status = ProMdlTypeGet (assembly, &type);

if (type != PRO_MDL_ASSEMBLY)
{
    ProMessageDisplay (message_file, "USER %0s", "Model type is not
        PRO_MDL_ASSEMBLY!");
    return (status);
}

/*-----*\
    Fill in the simplified representation data and create the
    simplified rep.
\*-----*/
ProStringToWstring (name, "NEW");
status = ProSimpdatapdataAlloc (name, PRO_B_FALSE,
    PRO_SIMPREP_INCLUDE, &rep);
status = ProSimpdatapdataCreate (assembly, rep, &simp_rep);
if (status != PRO_TK_NO_ERROR)
{
    ProMessageDisplay (message_file, "USER %0s",
        "Could not create Simplified Rep.");
    return (status);
}
ProMessageDisplay (message_file, "USER %0s",
    "SimpRep successfully created.");

/*-----*\
    Let the user select the parts to exclude.
\*-----*/
status = ProMessageDisplay (message_file, "USER %0s",
    "Please select parts to be excluded");
status = ProSelect ("part", -1, NULL, NULL, NULL, NULL, &sels, &num);

/*-----*\
    Add the items to the list of excluded items.
\*-----*/
status = ProSimpdatapdataActionInit (PRO_SIMPREP_EXCLUDE, NULL,
    &sr_act);
for (i = 0; i < num; i++)
{
    status = ProSelectionAsmcomppathGet (sels[i], &comp_path);
    status = ProSimpdatapdataitemInit (comp_path.comp_id_table,
        comp_path.table_num, PRO_VALUE_UNUSED, &sr_act, &item );
    status = ProSimpdatapdataitemAdd (rep , &item);
}
status = ProSimpdatapdataSet (&simp_rep, rep);

/*-----*\
    Set the new simplified rep to be the current representation,

```



```

    and redisplay the assembly.
/*-----*/
    status = ProSimpdataprepActivate (assembly, &simp_rep);
    status = ProSolidDisplay (assembly);
/*-----*/
    Free the data and return.
/*-----*/
    status = ProSimpdataprepdataFree (&rep);
    return (status);
}

```

Extracting Information About Simplified Representations

Functions Introduced:

- **ProSimpdataprepdataGet()**
- **ProSimpdataprepdataFree()**
- **ProSimpdataprepdataDefltGet()**
- **ProSimpdataprepdataNameGet()**
- **ProSimpdataprepdataTmpvalGet()**
- **ProSimpdataprepdataitemsVisit()**
- **ProSimpdataprepSubstitutionNameGet()**
- **ProAsmcompSubstitutionTypeGet()**
- **ProAsmcompSubstituteGet()**

Note: Pro/TOOLKIT supports simplified representation of Assemblies only, not Parts.

Given the handle to a simplified representation and the address of a pointer to a *ProSimpdataprepdata* structure, **ProSimpdataprepdataGet()** fills out the *ProSimpdataprepdata* structure. This function dynamically allocates storage for the data structure. When the memory is no longer needed, free it using the function **ProSimpdataprepdataFree()**.

The **ProSimpdataprepdataDefltGet()**, **ProSimpdataprepdataNameGet()**, and **ProSimpdataprepdataTmpvalGet()** functions return the associated values contained in the *ProSimpdataprepdata* structure. They all take two arguments—the data structure to be queried, and the appropriate data structure for the type to be retrieved.

ProSimpdataprepdataTmpvalGet() retrieves the value of the *temp* field from the specified *ProSimpdataprepdata* structure.

The function **ProSimpdataprepItemsVisit()** visits all the items that make up the simplified representation. The action and filter functions both have *ProSimpdataprepitem** as their first argument.

The function **ProSimpdataprepSubstitutionNameGet()** returns the name of the substituted representation at the given assembly path even when the substituted representation is deleted from the model at the given path.

The function **ProAsmcompSubstitutionTypeGet()** returns the substitution type performed on the simplified representation of an assembly component. It takes the path to the component representation that is being substituted, including the component ID, as one of its input arguments.

The function **ProAsmcompSubstituteGet()** returns the path to the substituted component representation in the form of *ProAsmcompPath* and the handle to the substituted component representation in the form of *ProAsmcomp*. It takes the path to the component representation that is being substituted, including the component ID, as one of its input arguments.

Example 2: Visiting the Items in a Simplified Representation

The following example shows how to use the Pro/TOOLKIT functions to visit the items in the specified simplified representation.

```
#include <ProToolkit.h>
#include <ProSimpdataprep.h>
#include <ProSimpdataprepdata.h>
#include <ProSelection.h>
#include <ProUtil.h>
#include <ProSolid.h>
#include <ProAsmcomp.h>
#include <ProAsmcompPath.h>
#include <ProModelitem.h>

#include <TestError.h>
#include <PTApplsUnicodeUtils.h>

/*=====*\
FUNCTION : UserSimpRepInfo
PURPOSE  : Prints out the simplified representation
           information.
\*=====*/
int UserSimpRepInfo ()
{
    ProMdl  owner;
    ProSimpdataprep      simp_rep;
```

```

ProSimpdatapdata      *data;
ProSimpprepActionType def_act;
wchar_t               w_sr_name[PRO_LINE_SIZE];
char                  temp[PRO_LINE_SIZE] = "";
ProBoolean            tmp_rep = PRO_B_FALSE;
ProError              status;
int                   num = 0;

ProError UserSimpRepVisAct();

status = ProMdlCurrentGet(&owner);
ERROR_CHECK("UserSimpRepInfo", "ProMdlCurrentGet", status);
if (status != PRO_TK_NO_ERROR) return (status);
status = ProSimpprepSelect ((ProSolid)owner, &simp_rep);
ERROR_CHECK("UserSimpRepInfo", "ProSimpprepSelect", status);
if (status == PRO_TK_NO_ERROR && simp_rep.id != -1)
{
    status = ProSimpdatapdataGet (&simp_rep, &data);
    ERROR_CHECK("UserSimpRepInfo", "ProSimpdatapdataGet", status);
    if (status == PRO_TK_NO_ERROR)
    {
        status = ProSimpdatapdataNameGet (data, w_sr_name);
        ERROR_CHECK("UserSimpRepInfo", "ProSimpdatapdataNameGet", status);
        if (status == PRO_TK_NO_ERROR)
        {
            pro_wstr_to_str (temp, w_sr_name);
            ProTKPrintf ("Simp rep name          : %s\n", temp);
        }
        status = ProSimpdatapdataDefltGet (data, &def_act);
        ERROR_CHECK("UserSimpRepInfo", "ProSimpdatapdataDefltGet", status);
        if (status == PRO_TK_NO_ERROR)
        {
            switch (def_act)
            {
                case PRO_SIMPREP_NONE:
                    strcpy (temp, "None");
                    break;

                case PRO_SIMPREP_REVERSE:
                    strcpy (temp, "Reverse");
                    break;

                case PRO_SIMPREP_SUBSTITUTE:
                    strcpy (temp, "Substitute");
                    break;

                case PRO_SIMPREP_INCLUDE:
                    strcpy (temp, "Include");
                    break;

                case PRO_SIMPREP_EXCLUDE:

```

```

        strcpy (temp, "Exclude");
        break;

        default:
            strcpy (temp, "Unknown");
    }
    ProTKPrintf ("Default action          : %s\n",
                temp);
}
status = ProSimpredataTmpvalGet (data, &tmp_rep);
ERROR_CHECK("UserSimpRepInfo", "ProSimpredataTmpvalGet", status);
if (status == PRO_TK_NO_ERROR)
{
    ProTKPrintf ("Temporary simp rep      : %s\n",
                tmp_rep == PRO_B_TRUE ? "Yes" : "No");
}

ProSimpredataitemsVisit (data, NULL,
UserSimpRepVisAct, (ProAppData) owner);
ERROR_CHECK("UserSimpRepInfo", "ProSimpredataitemsVisit", status);
status = ProSimpredataFree (&data);
}
}
return status;
}
/*=====*\
FUNCTION : UserSimpRepVisAct
PURPOSE  : Highlights the members of the simplified
            representation.
\*=====*/
ProError UserSimpRepVisAct (
    ProSimprepitem *item,
    ProError        err,
    ProSolid        owner)
{
    ProError        status;
    ProSelection     selection;
    ProAsmcomppath   comp_path;
    ProModelitem     mdl_item;
    ProFeature       feature;
    ProMdl           model;
    ProMdlType       mdl_type;
    int              id;

    if (item->item_path.path_size != -1)
    {
/*-----*\
Part
\*-----*/
        status = ProAsmcomppathInit (owner, item->item_path.comp_path,
                                    item->item_path.path_size, &comp_path);

```

```

ERROR_CHECK("UserSimpRepVisAct", "ProAsmcomppathInit", status);
if (status == PRO_TK_NO_ERROR)
{
    status = ProAsmcomppathMdlGet (&comp_path, &model);
    ERROR_CHECK("UserSimpRepVisAct", "ProAsmcompMdlGet", status);
}
if (status == PRO_TK_NO_ERROR)
{
    status = ProMdlIdGet(model, &id);
    ERROR_CHECK("UserSimpRepVisAct", "ProMdlIdGet", status);

    status = ProMdlTypeGet(model, &mdl_type);
    ERROR_CHECK("UserSimpRepVisAct", "ProMdlIdGet", status);

    if (status == PRO_TK_NO_ERROR)
    {
        status = ProModelitemInit (model, id, (ProType)mdl_type,
                                   &mdl_item);
        ERROR_CHECK("UserSimpRepVisAct", "ProModelitemInit", status);
    }

    if (status == PRO_TK_NO_ERROR)
    {
        status = ProSelectionAlloc(&comp_path, &mdl_item, &selection);
        ERROR_CHECK("UserSimpRepVisAct", "ProSelectionAlloc", status);
    }
}
else
{
    /*-----*\
    Feature
    \*-----*/
    status = ProFeatureInit (owner, item->item_path.feat_id, &feature);
    ERROR_CHECK("UserSimpRepVisAct", "ProFeatureInit", status);
    if (status == PRO_TK_NO_ERROR)
    {
        status = ProFeatureSelectionGet (&feature, &selection);
        ERROR_CHECK("UserSimpRepVisAct", "ProFeatureSelectionGet", status);
    }
}

if (status == PRO_TK_NO_ERROR)
{
    ProSelectionHighlight (selection, PRO_COLOR_WARNING);
    ERROR_CHECK("UserSimpRepVisAct", "ProSelectionHighlight", status);
}

return PRO_TK_NO_ERROR;
}

```

Modifying Simplified Representations

Functions Introduced:

- **ProSimprepActionInit()**
- **ProSimprepdataSet()**
- **ProSimprepdataDefltSet()**
- **ProSimprepdataNameSet()**
- **ProSimprepdataTmpvalSet()**

Note: Pro/TOOLKIT supports simplified representation of Assemblies only, not Parts.

Using Pro/TOOLKIT, you can modify the attributes of existing simplified representations. After you create or retrieve a simplified representation, you can make calls to the **ProSimprepdata*Set()** functions listed in this section to designate new values for the fields in the *ProSimprepdata* structure.

To modify an existing simplified representation, retrieve it, then get the handle to its *ProSimprepdata* structure by calling the function **ProSimprepdataGet()**. (If you created the representation programmatically within the same application, the *ProSimprepdata* handle is already available.) After modifying the data structure, reassign it to the corresponding simplified representation by calling the function **ProSimprepdataSet()**. Use function **ProSimprepdataTmpvalSet()** to set the value of the *temp* field in the specified *ProSimprepdata* structure.

Adding Items to and Deleting Items from a Simplified Representation

Functions Introduced:

- **ProSimprepdataitemAdd()**
- **ProSimprepdataitemDelete()**
- **ProSimprepdataitemInit()**

Note: Pro/TOOLKIT supports simplified representation of Assemblies only, not Parts.

You can add and delete items from the list of components in a simplified representation using Pro/TOOLKIT. If you created a simplified representation using the option **Exclude** as the default rule, you would generate a list containing the items you want to include. Similarly, if the default rule for a simplified representation is **Include**, you can add the items that you want to be excluded from the simplified representation to the list, setting the value of the *ProSimpredActionType* to PRO_SIMPREP_EXCLUDE.

To Add Items

1. Get the *ProSimpreddata* structure, as described in the previous section.
2. Specify the action to be applied to the item with a call to the function **ProSimpredActionInit()**.
3. Initialize a *ProSimpreditem* structure for the item by calling the function **ProSimpreddataitemInit()**.
4. Add the item to the *ProSimpreddata* structure using the function **ProSimpreddataitemAdd()**.
5. Reassign the *ProSimpreddata* structure to the corresponding *ProSimpred* object by calling **ProSimpreddataSet()**.

To Remove Items

1. Get the *ProSimpreddata* structure handle.
2. Pass the *ProSimpreddata* handle and the *ProSimpreditem* handle for the item to be deleted to the function **ProSimpreddataitemDelete()**.
3. Reassign the *ProSimpreddata* structure to the corresponding *ProSimpred* object by calling the function **ProSimpreddataSet()**.

Gathering Components by Rule

Function Introduced:

- **ProRuleEval()**

Pro/ENGINEER provides large assembly management tools. This section describes the access to some of this functionality through Pro/TOOLKIT.

You can specify different types of rules and use them to generate a list of components for which the rule applies. After initializing the rule, call the function **ProRuleEval()** to generate the list of components that follow this rule.

Note that the returned list of components is in the form of an expandable array (*ProArray*), which is allocated by this function. To release the allocated memory, call the function **ProArrayFree()**.

The components can be gathered using the following rules:

- By model name
- By parameters, using an expression
- By location with a zone
- By distance from a point
- By size
- By an existing simplified representation

See the *Assembly Modeling User's Guide* for more details on this functionality.

Gathering by Model Name

Function Introduced:

- **ProRuleInitName()**

The function **ProRuleInitName()** initializes the rule for gathering by model name. The *name_mask* variable can be a wildcard. For more information, see the *Introduction to ProENGINEER*.

Gathering by Parameters

Function Introduced:

- **ProRuleInitExpr()**

You can specify a expression in the relations format to search for components of a particular parameter value. For example, consider the following expression:

```
type == "electrical" | cost <= 10
```

When you supply this expression to the rule, it gathers the components that have a “cost” parameter of less than or equal to 10, or whose *type* parameter is set to “electrical.”

The *expr* variable is an array of *ProLine* structures. You allocate this array using the function **ProArrayAlloc()**. The **ProArray*** functions are used for all array manipulations.

Gathering by Zone

Function Introduced:

- **ProRuleInitZone()**

When you specify this rule, all the components that belong to the supplied zone feature are gathered.

See the *Assembly Modeling User's Guide* for detailed information about setting up and working with zones.

When you create a zone, the function creates a feature of type PRO_FEAT_ZONE in the top-level assembly.

Gathering by Distance from a Point

Function Introduced:

- **ProRuleInitDist()**

Using **ProRuleInitDist()** to set up a rule that specifies distance from a point, Pro/TOOLKIT gathers all the components within the specified spherical region.

By filling the *ProRuleDist* data structure, you can specify the center and the distance from the center. This information is in the coordinates of the top-level assembly.

Gathering by Size

Function Introduced:

- **ProRuleInitSize()**

By filling the *ProRuleSize* data structure, you can specify the size of components to be gathered.

If you want to gather the components greater than the specified size, set the field *greater* to PRO_B_TRUE. If you set the field to PRO_B_FALSE, the function gathers the components that are less than the specified size.

If you want the specified size to be in absolute terms, set the field *absolute* to PRO_B_TRUE. Note that in this case, the function uses the units of the top-level assembly.

If the information is relative, set the field *absolute* to `PRO_B_FALSE`. In this case, the only valid values that can be specified are in the range (0.0, 1.0). The function compares the component size to that of the top-level assembly, and uses this ratio to determine whether the component should be gathered.

Gathering by Simplified Representation

Function Introduced:

- **ProRuleInitRep()**

You can gather components that belong to an existing simplified representation by calling the function **ProRuleInitRep()**, which initializes the rule.

Assembly: Data Sharing Features

This chapter describes the Pro/TOOLKIT functions access for Data Sharing Features. These types of features are used to transfer information (geometry, annotations, and other details) from one model to another. Data Sharing Features also help you to consolidate your design information in a central location, control change propagation which aid towards accomplishing top down design objectives. Data Sharing Features are of the following types:

- Copy Geometry / Publish Geometry
- Shrinkwrap
- Merge / Cutout
- Inheritance

It also explains how to access the properties of variant features and lists read and write functions supporting the Inheritance Features.

Topic	Page
Copy Geometry, Publish Geometry, and Shrinkwrap Features	50 - 2
General Merge (Merge, Cutout and Inheritance Feature)	50 - 12
Inheritance Feature and Flexible Component Variant Items	50 - 15

Copy Geometry, Publish Geometry, and Shrinkwrap Features

Copy Geometry Features are used to pass any type of geometric reference information, user-defined parameters to and from parts, skeleton models, and assemblies. Copy Geometry features can only copy reference geometry such as surface, datum features and not solid geometry. They are used in a top-down design to reduce the amount of data in session, thus avoiding the retrieval of entire reference source models.

A Publish Geometry feature contains independent, local geometry references. Only local geometry can be referenced in a Publish Geometry feature and external references are not allowed. A Publish Geometry feature has no geometry and does not create local copies of the references selected for its definition. It simply consolidates multiple local references in a model so that they can be copied to other models.

A Shrinkwrap feature is a collection of surfaces and datum features of a model that represents the exterior of the model. You can use a part, skeleton, or top-level assembly as the source model for a Shrinkwrap feature. A Shrinkwrap feature is associative and automatically updates to reflect changes in the parent copied surfaces.

Feature Element Tree for the Copy Geometry, Publish Geometry, and Shrinkwrap Features

The element tree for Copy Geometry, Publish Geometry, and Shrinkwrap features is documented in the header file *ProDataShareFeat.h*.

The following figure demonstrates the feature element tree structure:

Figure 50-1: Feature Element Tree for Copy Geometry, Publish Geometry, and Shrinkwrap Features

```

PRO_E_FEATURE_TREE
|
|-PRO_E_FEATURE_TYPE
|-PRO_E_STD_FEATURE_NAME
|-PRO_E_CG_FEAT_SUB_TYPE
|-PRO_E_CG_REFS_TYPE
|-PRO_E_CG_LOCATION
|-PRO_E_CG_PG_OR_REFS
|   |-PRO_E_CG_PUBD_GEOM
|   |-PRO_E_CG_REFS_COLL
|       |-PRO_E_STD_SURF_COLLECTION_APPL
|       |-PRO_E_STD_CURVE_COLLECTION_APPL
|       |-PRO_E_CG_OBJS_COLL
|-PRO_E_SW_COLLECTION_TYPE
|-PRO_E_SW_OPTIONS
|   |-PRO_E_SW_QUALITY
|   |-PRO_E_SW_FILL_HOLES
|   |-PRO_E_SW_COLLECT_QUILTS
|   |-PRO_E_SW_SKIP_SURF_SIZE
|   |-PRO_E_SW_COLLECT_ORDER
|   |-PRO_E_SW_RES_GEOM_OPT
|   |-PRO_E_SW_FAILED_SLD_OPT
|   |-PRO_E_SW_FILL_CNTRS_ARR
|   |   |-PRO_E_SW_FILL_CNTRS
|   |   |   |-PRO_E_SW_FILL_CNTRS_SRF_SEL
|   |   |-PRO_E_SW_FILL_CNTRS_DISP_CRV
|-PRO_E_SW_COMP_SUBSET
|   |-PRO_E_SW_COMPONENT
|       |-PRO_E_SW_INCLUDE_COMP
|       |-PRO_E_SW_SEL_COMPONENT
|-PRO_E_SW_REFS_COLL
|   |-PRO_E_STD_SURF_COLLECTION_APPL
|   |-PRO_E_SW_EXCLUDE_SURF_COLL_APPL
|   |-PRO_E_STD_CURVE_COLLECTION_APPL
|   |-PRO_E_CG_OBJS_COLL
|-PRO_E_DSF_PROPAGATE_ANNOTS
|-PRO_E_CG_SRFS_COPY
|   |-PRO_E_SRF_COPY_TYPE
|   |-PRO_E_SRF_COPY_EXCL
|   |-PRO_E_SRF_COPY_FILL
|   |-PRO_E_STD_CURVE_COLLECTION_APPL
|--PRO_E_DSF_DTMS_FIT
|-PRO_E_DSF_DEPENDENCY

```

Figure 50-2: PRO_E_CG_LOCATION

```
PRO_E_CG_LOCATION
|-PRO_E_DSF_EXT_LOCAL_TYPE
|-PRO_E_DSF_SEL_REF_MDL
|-PRO_E_CG_PLACEMENT
  |-PRO_E_CG_PLACE_TYPE
  |-PRO_E_CG_CSYS_PLACE
    |-PRO_E_CG_PLC_LOCAL_CSYS
    |-PRO_E_CG_PLC_EXT_CSYS
```

Figure 50-3: PRO_E_DSF_PROPAGATE_ANNOTS

```
PRO_E_DSF_PROPAGATE_ANNOTS
|-PRO_E_DSF_ANNOT_COPY_ALL
|-PRO_E_DSF_ANNOT_DEPEND_ALL
|-PRO_E_DSF_ANNOT_AUTO_COPY_DTM
|-PRO_E_DSF_ANNOT_SELECTIONS
  |-PRO_E_DSF_ANNOT_SELECTION
    |-PRO_E_DSF_ANNOT_SEL_ANNOTS
    |-PRO_E_DSF_ANNOT_COPY_STATUS
    |-PRO_E_DSF_ANNOT_DEPENDENCY
```

Figure 50-4: PRO_E_DSF_DTM_FIT

```
PRO_E_DSF_DTM_FIT
|-PRO_E_DSF_DTM_SELECTION
|-PRO_E_DTMLN_FIT
  |-PRO_E_DTMLN_FIT_TYPE
  |-PRO_E_DTMLN_FIT_REF
  |-PRO_E_DTMLN_FIT_DTM_RAD
  |-PRO_E_DTMLN_FIT_OUTLINE
|-PRO_E_DTMAXIS_FIT
  |-PRO_E_DTMAXIS_FIT_TYPE
  |-PRO_E_DTMAXIS_FIT_REF
  |-PRO_E_DTMAXIS_FIT_LEN
```

The following list details special information about some of the elements in this tree:

- **PRO_E_FEATURE_TYPE**—Specifies the feature type and should be **PRO_FEAT_GEOM_COPY**, for Copy Geometry, Publish Geometry, and Shrinkwrap features.
- **PRO_E_CG_FEAT_SUB_TYPE**—Specifies the sub feature type and is a mandatory element. It is visible for all copy geometry features. The valid sub types are as follows:
 - **PRO_CG_COPY_GEOM**—Copy Geometry feature
 - **PRO_CG_SHRINKWRAP**—Shrinkwrap feature
 - **PRO_CG_PUB_GEOM**—Publish Geometry feature
- **PRO_E_CG_REFS_TYPE**—Specifies the type of references used in a copy geometry feature (a published geometry feature, or an selected array of surfaces, edges, curves and datums). It is visible for Copy Geometry features.
- **PRO_E_CG_LOCATION**—Specifies a compound element that indicates the method used for placement of the feature within the parent model. This is valid for Copy Geometry and Shrinkwrap features. See the section describing “Element Details of the Subtree **PRO_E_CG_LOCATION**” for more information.
- **PRO_E_CG_PG_OR_REFS**—Specifies either a published geometry feature to copy or a collection of local geometry references to copy. It is visible for Copy Geometry features and has the following elements:
 - **PRO_E_CG_PUBD_GEOM**—Specifies the selected publish geometry feature to be copied.
 - **PRO_E_CG_REFS_COLL**—Specifies the collection of references. It is used for Copy Geometry and Publish Geometry features and has the following elements:
 - **PRO_E_STD_SURF_COLLECTION_APPL**—Specifies a collection of selected surfaces to copy.
 - **PRO_E_STD_CURVE_COLLECTION_APPL**—Specifies a collection of selected curves and / or edges to copy.
 - **PRO_E_CG_OBJS_COLL**—Specifies a multivalued element containing miscellaneous references such as datums, quilts, points etc.

Note: The reference collection elements PRO_E_STD_SURF_COLLECTION_APPL, PRO_E_STD_CURVE_COLLECTION_APPL, PRO_E_CG_OBJS_COLL may return the item of type PRO_QUERY. If PRO_QUERY is encountered, the Pro/TOOLKIT application cannot access or modify the rules of the query. However, the application can redefine the other properties of the DSF without affecting or removing the query. For more information refer to the section “Saved Queries for Copy Geometry and Publish Geometry Features”.

- PRO_E_SW_COLLECTION_TYPE—Specifies the collection mode used while creating a Shrinkwrap feature. The values for this element, specified by the enumerated type *ProShrinkwrapCollectionType*, are as follows:
 - PRO_SW_OUTER_SHELL—Surfaces and datums that represent the exterior of the model are used. This is the default.
 - PRO_SW_ALL_SOLID_SURFS—All solid surfaces in the model are automatically collected.
 - PRO_SW_MANUAL—Surfaces, edges, curves, and datums that you select are used. You can copy geometry from more than one reference model.
- PRO_E_SW_OPTIONS—Specifies Shrinkwrap feature options. This element is visible for Shrinkwrap features and has the following elements:
 - PRO_E_SW_QUALITY—Specifies the shrinkwrap quality level used when identifying the contributing geometry to the Shrinkwrap feature. It can have valid values in the range of 1 to PRO_MAX_SHRINKWRAP_QUALITY_LVL.
 - PRO_E_SW_FILL_HOLES—Specifies whether or not to use the option to auto fill holes.
 - PRO_E_SW_COLLECT_QUILTS—Specifies whether or not to include external quilts in the Shrinkwrap feature.
 - PRO_E_SW_SKIP_SURF_SIZE—Specifies the shrinkwrap skip surfaces size. Pro/ENGINEER will not include surfaces smaller than the specified percentage of the model's size in the Shrinkwrap model. It can have a value ranging from 0(default) to 100%, to specify the relative size of the surface to ignore.

- PRO_E_SW_COLLECT_ORDER—Specifies how the system will handle the subcomponents in creating the feature. This value can be of the following types:
 - PRO_SW_SHRINKWRAP_AND_SELECT—Specifies to first shrinkwrap and then select (selected by default). The system analyzes the entire assembly to identify the external surfaces to be included and only the appropriate surfaces that belong to the selected components are included in the resulting shrinkwrap feature.
 - PRO_SW_SELECT_AND_SHRINKWRAP—Specifies select and shrinkwrap. The system builds a shrinkwrap based on selected components.
- PRO_E_SW_RES_GEOM_OPT—Specifies the options for the resultant Shrinkwrap geometry. This element is available only if the element PRO_E_SW_COLLECTION_TYPE is set to the value PRO_SW_ALL_SOLID_SURFS. The values for this element, specified by the enumerated type *ProShrinkwrapResGeomOpt*, are as follows:
 - PRO_SW_RES_GEOM_QUILT—The resulting Shrinkwrap feature is a quilt.
 - PRO_SW_RES_GEOM_SOLID—The resulting Shrinkwrap feature is a solid.
- PRO_E_SW__FAILED_SLD_OPT—Specifies the options to address the failed solidification subfeatures (external reference copy geometry features) of a Shrinkwrap feature. This element is available only if the element PRO_E_SW_RES_GEOM_OPT is set to the value PRO_SW_RES_GEOM_SOLID. The values for this element, specified by the enumerated type *ProShrinkwrapFailedSldOpt*, are as follows:
 - PRO_SW_FAILED_SLD_FAIL—Failed solidification subfeatures of the Shrinkwrap feature are not resolved. As a result, the Shrinkwrap feature also fails.
 - PRO_SW_FAILED_SLD_TO_QUILT—Failed solidification subfeatures of the Shrinkwrap feature are restored as quilts. As a result, the Shrinkwrap feature does not fail.

- PRO_E_SW_FILL_CNTRS_ARR—Specifies the contours of open spaces that need to be filled. This element consists of an array of compound elements of the type PRO_E_SW_FILL_CNTRS which consists of the following elements:
 - PRO_E_SW_FILL_CNTRS_SRF_SEL—Select the surface that defines the contour to be filled.
- PRO_E_SW_FILL_CNTRS_DISP_CRV—Displays a yellow curve instead of a filled contour.
- PRO_E_SW_COMP_SUBSET—Specifies components of the assembly to be considered when creating the Shrinkwrap. This array element consists of an array of Shrinkwrap component subset elements (PRO_E_SW_COMPONENT). That element includes the following elements:
 - PRO_E_SW_INCLUDE_COMP specifies whether or not to include the component in the shrinkwrap
 - PRO_E_SW_SEL_COMPONENT specifies the selected component.
- PRO_E_SW_REFS_COLL—Specifies the collection of surfaces and other references to be included or excluded from the Shrinkwrap feature. It has the following elements:
 - PRO_E_STD_SURF_COLLECTION_APPL—Specifies the collection of individual surfaces that must always be included in the Shrinkwrap feature.
 - PRO_E_SW_EXCLUDE_SURF_COLL_APPL—Specifies the collection of individual surfaces that must always be excluded from the Shrinkwrap feature.
 - PRO_E_STD_CURVE_COLLECTION_APPL—Specifies a collection of chains constructed by selection of edges or curves.
 - PRO_E_CG_OBJS_COLL—Specifies a multivalued element containing miscellaneous references such as datums, quilts, points etc.
- PRO_E_DSF_PROPAGATE_ANNOTS—Specifies rules about how to propagate annotations. See the section “Element Details of PRO_E_DSF_PROPAGATE_ANNOTS” below for the structure and contents of this element.
- PRO_E_CG_SRFS_COPY—Specifies a compound element that specifies copied surfaces. It is visible for Copy Geometry features and has the following elements:

- PRO_E_SRF_COPY_TYPE specifies the type of copied surface. It can have one of the following values:
 - PRO_SRFCOPY_AS_IS
 - PRO_SRFCOPY_EXCLD_FILL
 - PRO_SRFCOPY_INSIDE_BNDRY
- PRO_E_SRF_COPY_EXCL specifies excluded surfaces
- PRO_E_SRF_COPY_FILL specifies loops to fill
- PRO_E_STD_CURVE_COLLECTION_APPL specifies the boundary for the copied surfaces
- PRO_E_DSF_DTMS_FITS—Specifies a compound element that indicates the rules for fitting datums in the DSF feature. It is visible for all internal Copy Geometry features) and Shrinkwrap features. This compound element includes PRO_E_DTMLN_FIT and PRO_E_DTMAXIS_FIT subtrees for any or all of the copied datums. Refer to the chapter “Element Trees: Datum Features” for details on the datum fit subtrees.
- PRO_E_DSF_DEPENDENCY—Specifies the dependency flag of the Copy Geometry Feature and can be of the following values:
 - PRO_DSF_DEPENDENT specifies that the geometry can be updated when its parent changes.
 - PRO_DSF_INDEPENDENT suspends the relationship between the current feature and the original geometry. If you change the original part, the current feature does not change.

Element Details of the Subtree PRO_E_CG_LOCATION

The compound element PRO_E_CG_LOCATION has the following elements:

- PRO_E_DSF_EXT_LOCAL_TYPE—Specifies the DSF location type:
 - PRO_DSF_PLACE_LOCAL type is a local feature within the assembly. Therefore the references will be dependent upon the assembly structure and the feature does not need placement information.
 - PRO_DSF_PLACE_EXTERNAL type is set to externalize a Data Sharing Feature. An external data-sharing feature must be placed in its target model explicitly.

Note: The purpose of External DSF's is to copy geometry from one model to another model without the need to copy the geometry in the context of the assembly. External DSF's reduce the dependency on the assembly and all models along the path between the two components. Once a feature has been made "External", it cannot be converted to become internal.

- PRO_E_DSF_SEL_REF_MDL—Specifies the model to use for the external DSF.
- PRO_E_CG_PLACEMENT—Specifies the placement of the external reference model in the target model. It has the following elements:
 - PRO_E_CG_PLACE_TYPE—The external placement reference for the copied geometry can be of the following types:
 - PRO_CG_PLC_DEFAULT—Locates the copied geometry in the current model using the default location.
 - PRO_CG_PLC_CSYS_CSYS—Locates the copied geometry in the current model by aligning coordinate systems.
 - PRO_CG_PLC_CURRENT—Locates the copied geometry in the current using the current placement (applicable only during a conversion of a local DSF to an external DSF).
 - PRO_E_CG_CSYS_PLACE—The two reference elements below this element specifies Csys-Csys alignment.

Element Details of PRO_E_DSF_PROPAGATE_ANNOTS

The compound element describes the options available to propagate annotations in DSF's. It has the following subelements:

- PRO_E_DSF_ANNOT_CPY_ALL—Specifies a flag whether to copy all annotation elements.
- PRO_E_DSF_ANNOT_DEPEND_ALL—Specifies a flag whether to make the copied annotation elements as dependent on their originals.
- PRO_E_DSF_ANNOT_AUTO_CPY_DTM—Specifies a flag to propagate annotation planes and other datums referenced by annotation elements automatically. Annotation planes are propagated if the annotation reference comprising of the solid or surface geometry is copied.

- **PRO_E_DSF_ANNOT_SELECTIONS**—Instead of using the automatic flags for propagation, the DSF feature can specify a list of annotations to propagate. Each subelement representing an annotation has the following sub-elements:
 - **PRO_E_DSF_ANNOT_SEL_ANNOTS** specifies the manually selected annotation element
 - **PRO_E_DSF_ANNOT_CPY_STATUS** specifies the copy status for this selected element.
 - **PRO_E_DSF_ANNOT_DEPENDENCY** specifies whether to make the annotation dependant in the DSF. It can have one of the following values:
 - **PRO_DSF_DEPENDENT** is set by default.
 - **PRO_DSF_INDEPENDENT** makes the annotations independent of changes made to the parent annotation.

Saved Queries for Copy Geometry and Publish Geometry Features

Copy Geometry and Publish Geometry features have the ability to retain and reuse search tool queries defined from its collectors. The functions described in this section provide the ability to update these query-driven data sharing features using Pro/TOOLKIT.

Functions Introduced:

- **ProDatasharingfeatureIsQuerydriven()**
- **ProDatasharingfeatureQueryUpdate()**

The function **ProDatasharingfeatureIsQuerydriven()** returns true if the specified data sharing feature is query driven.

The function **ProDatasharingfeatureQueryUpdate()** updates the items collected by the query within the Copy Geometry or Publish Geometry feature. This will regenerate the feature and may cause geometry to be added or removed.

General Merge (Merge, Cutout and Inheritance Feature)

Feature Element Tree

The element tree for the general merge feature is documented in the header file *ProDataShareFeat.h*. The following figure demonstrates the feature element tree structure:

Figure 50-5: Feature Element tree for General Merge Feature

```
PRO_E_FEATURE_TREE
|
|-PRO_E_FEATURE_TYPE
|-PRO_E_STD_FEATURE_NAME
|-PRO_E_GMRG_FEAT_TYPE
|-PRO_E_DSF_REF_MDL
|   |--PRO_E_DSF_EXT_LOCAL_TYPE
|   |--PRO_E_DSF_SEL_REF_MDL
|   |--PRO_E_DSF_PLACEMENT
|-PRO_E_GMRG_MATERIAL_OPT
|-PRO_E_GMRG_VARIED_ITEMS
|-PRO_E_GMRG_COPY_DATUMS
|-PRO_E_DSF_PROPAGATE_ANNOTS
|--PRO_E_DSF_DTMS_FIT
|   |-PRO_E_DSF_DTM_FIT
|-PRO_E_DSF_DEPENDENCY
```

Figure 50-6: PRO_E_DSF_PLACEMENT

```

PRO_E_DSF_PLACEMENT
|--PRO_E_COMPONENT_CONSTRAINTS
  |--PRO_E_COMPONENT_CONSTRAINT
    |--PRO_E_COMPONENT_CONSTR_TYPE
    |--PRO_E_COMPONENT_COMP_CONSTR_REF
    |--PRO_E_COMPONENT_ASSEM_CONSTR_REF
    |--PRO_E_COMPONENT_CONSTR_REF_OFFSET
    |--PRO_E_COMPONENT_USER_DATA
    |--PRO_E_COMPONENT_CONSTR_ATTR
    |--PRO_E_COMPONENT_COMP_ORIENT
    |--PRO_E_COMPONENT_ASSM_ORIENT
    |--PRO_E_COMPONENT_CONSTR_SET_ID
    |--PRO_E_COMPONENT_SLOT_EXTRA_CRV_REF

```

The following list details special information about some of the elements in this tree:

- PRO_E_FEATURE_TYPE—Specifies the feature type and should be PRO_FEAT_GEN_MERGE.
- PRO_E_GMRG_FEAT_TYPE—Specifies the type of General Merge Feature:
 - PRO_GEN_MERGE_TYPE_MERGE (Merge or cutout feature)
 - PRO_GEN_MERGE_TYPE_INHERITANCE (Inheritance feature)
- PRO_E_DSF_REF_MDL—Specifies the reference model. It has the following elements:
 - PRO_E_DSF_EXT_LOCAL_TYPE—Specifies the DSF location type and is of the following type:
 - PRO_DSF_PLACE_LOCAL type is a local reference to the reference model
 - PRO_DSF_PLACE_EXTERNAL indicates an external merge, cutout, or inheritance feature.
 - PRO_E_DSF_SEL_REF_MDL—Specifies a selected reference model.
 - PRO_E_DSF_PLACEMENT—Specifies the placement of the Data Sharing feature. It contains the following elements:

- PRO_E_COMPONENT_CONSTRAINTS

For more information on component constraint elements, refer to chapter Assembly: Assembling Components.

- PRO_E_GMRG_MATERIAL_OPT—Specifies the general material options. These are listed as follows:
 - PRO_GEN_MERGE_RMV_MATERIAL (Remove material)
 - PRO_GEN_MERGE_ADD_MATERIAL (Add material)
- PRO_E_GMRG_VARIED_ITEMS—Specifies a pointer element that defines the inheritance feature varied items and their values. This handle cannot be directly read or modified by Pro/TOOLKIT. Instead, use the Visit functions available in *ProVariantFeat.h* to read varied items. To set varied items (after the feature has been created), use the appropriate modification function on an item whose owner is the variant feature model handle (**ProVariantfeatMdlGet()**). For more information, refer to the section “Inheritance Feature and Flexible Component Variant Items”.
- PRO_E_GMRG_COPY_DATUMS—True to copy datums with this merge or inheritance feature, false to leave them uncopied.
- PRO_E_DSF_PROPAGATE_ANNOTS—Specifies rules about how to propagate annotations. Refer to the section “Feature Element Tree for the Copy Geometry, Publish Geometry, and Shrinkwrap Features” for the details. A Shrinkwrap feature is a collection of surfaces and datum features of a model that represents the exterior of the model. You can use a part, skeleton, or top-level assembly as the source model for a Shrinkwrap feature. A Shrinkwrap feature is associative and automatically updates to reflect changes in the parent copied surfaces.
- PRO_E_DSF_DTMS_FIT—Specifies a compound element that indicates the rules for fitting datums in the DSF feature. It is visible for all internal Copy Geometry features) and Shrinkwrap features. This compound element includes PRO_E_DTMLN_FIT and PRO_E_DTMAXIS_FIT subtrees for any or all of the copied datums. Refer to the chapter “Element Trees: Datum Features” for details on the datum fit subtrees.
- PRO_E_DSF_DEPENDENCY—Specifies the dependency flag of the Copy Geometry Feature and can be of the following values:
 - PRO_DSF_DEPENDENT specifies that the geometry can be updated when its parent changes.

- PRO_DSF_INDEPENDENT suspends the relationship between the current feature and the original geometry. If you change the original part, the current feature does not change.

Inheritance Feature and Flexible Component Variant Items

An Inheritance feature allows one-way associative propagation of geometry and feature data from a reference part to target part within an assembly. The reference part is the original part and the target part contains the inheritance features. Inheritance features are always created by referencing existing parts. An inheritance feature begins with all of its geometry and data identical to the reference part from which it is derived.

Users can vary the visibility and values of items in inheritance features. Pro/TOOLKIT offers the ability to access these varied items and their properties using the regular Pro/TOOLKIT function appropriate for the property. Pro/TOOLKIT also provides the ability to locate and separate the varied properties from the non-varied ones.

A flexible component has similar capabilities for variance of dimensions and parameters of a model in the context of an assembly. Pro/TOOLKIT supports access to the variant properties of flexible components through the regular Pro/TOOLKIT functions as well.

This section refers collectively to inheritance features and flexible components as "variant features".

Variant Feature Model

A special *ProMdl* handle represents the "inherited" or "flexible" model handle in functions which access variant features. This handle is called the "Variant Feature Model". Only certain functions will support inputs including the Variant Feature Model handle.

Functions Introduced:

- **ProVariantfeatMdlGet()**
- **ProMdlIsVariantfeatMdl()**

Use the function **ProVariantfeatMdlGet()** to obtain the special model pointer from a variant feature. This pointer can be used for accessing properties that can be modified by the presence of an inheritance feature or flexible component. Other functions will return an error PRO_TK_INVALID_PTR if provided with this pointer.

Note: If you are using the variant feature model handle and you need to use an unsupported function on it, you can attempt to retrieve the actual model by extracting the model name and type from the model and using **ProMdlRetrieve()** to get the original model. This requires that the parent model be accessible in session. Remember that the parent model will not reflect variations applied by the variant feature.

Use the function **ProMdlIsVariantfeatMdl()** to identify if a model pointer is a variant feature model handle.

Accessing Properties of Variant Features

The following items are supported by functions that accept the variant feature model handle. Except where noted, both read and write access is supported

- Dimensions
 - Variant dimensions of the inheritance feature:
 - Value
 - Tolerance
 - Dimension bound
 - Nonvariant dimensions of the inheritance feature (Properties are read only)
- Parameters
 - Variant parameters of the inheritance feature:
 - Value
 - Nonvariant parameters of the inheritance feature (Properties are read only)
- Feature (Inheritance feature only)
 - Variant member features:
 - Suppressed or resumed or erased status
 - Nonvariant member features (Properties are read only)

- Replaced or alternate feature references
- Annotation (Inheritance feature only)
 - Variant geometric tolerances:
 - Copy status
 - Value
 - Nonvariant geometric tolerances (Properties are read only)
 - Variant notes
 - Copy status
 - Shown or hidden property
 - Nonvariant notes (Properties are read only)
 - Variant symbols
 - Copy status
 - Nonvariant symbols (Properties are read only)
 - Variant surface finishes
 - Copy status
 - Value
 - Nonvariant surface finishes (Properties are read only)

Read Functions Supporting Inheritance Features

Basic Model Properties	ProMdlNameGet() ProMdlTypeGet() ProMdlSubtypeGet() ProMdlPrincipalunitsystemGet() ProUnitsystemUnitGet() ProUnitsystemTypeGet() ProUnitsystemIsStandard() ProUnitIsStandard() ProUnitTypeGet() ProUnitConversionGet() ProSolidAccuracyGet() ProSolidOutlineGet() ProSolidOutlineCompute()

Basic Model Items	ProModelitemNameGet() ProModelitemDefaultnameGet() ProSelectionModelitemGet()
Dimensions	ProSolidDimensionVisit() ProFeatureDimensionVisit() ProDimensionNomvalueGet() ProDimensionBoundGet() ProDimensionSymtextGet() ProDimensionTollabelGet() ProDimensionSymbolGet() ProDimensionValueGet() ProDimensionToltypeGet() ProDimensionToleranceGet() ProDimensionTolerancedecimalsGet() ProDimensionTolerancedenominatorGet() ProDimensionTypeGet() ProDimensionIsFractional() ProDimensionDecimalsGet() ProDimensionDenominatorGet() ProDimensionIsRelDriven() ProDimensionIsRegenednegative() ProDimensionTextGet() ProDimensionTextstyleGet() ProDimensionIsToleranceDisplayed() ProDimensionIsBasic() ProDimensionIsInspection() ProDimensionIsBaseline() ProDimensionIsOrdinate() ProDimensionOrdinatestandardGet() ProDimensionLocationGet() ProDimensionPlaneGet() ProDimensionAttachmentsGet() ProDimensionOverridevalueGet() ProDimensionValuedisplayGet()

Parameters	ProParameterVisit() ProParameterValueGet() ProParameterUnitsGet() ProParameterIsEnumerated() ProParameterRangeGet() ProParameterScaledvalueGet()
Features	ProSolidFeatVisit() ProFeatureTypeGet() ProFeatureTypenameGet() ProFeatureSubtypeGet() ProFeatureStatusGet() ProSolidFeatstatusGet() ProFeatureChildrenGet() ProFeatureParentsGet() ProFeatureCopyinfoGet() ProFeatureGroupGet() ProFeatureGroupStatusGet() ProFeatureGrppatternStatusGet() ProFeatureHasGeomchks() ProFeatureIsIncomplete() ProFeatureIsNcseq() ProFeatureIsReadonly() ProFeatureNumSectionsGet() ProFeatureNumberGet() ProFeaturePatternGet() ProFeaturePatternStatusGet() ProFeatureSectionCopy() ProFeatureSolidGet() ProFeatureVerstampGet() ProFeatureVisibilityGet()
Annotations	ProMdlNoteVisit() ProSolidDispoutlineGet() ProNoteTextGet() ProNotePlacementGet() ProNoteURLGet() ProNoteOwnerGet() ProNoteTextstyleGet() ProTextStyle*Get() ProNoteDtlnoteGet()

Annotations	ProNoteLeaderstyleGet() ProNoteElbowlengthGet() ProMdlGtolVisit() ProGtoldataFree() ProGtoldataOwnerGet() ProGtoldataTypeGet() ProGtolDataGet() ProGtoldata*Get() ProGtolRightTextGet() ProGtolTopTextGet() ProGtolPrefixGet() ProGtolSuffixGet() ProGtolTextstyleGet() ProGtoltextTextstyleGet() ProSolidDtlsyminstVisit() ProDtlsyminstDataGet() ProSolidSurffinishVisit() ProSurffinishValueGet() ProSurffinishReferencesGet() ProSurffinishDataGet() ProAnnotationIsShown() ProAnnotationIsInactive() ProAnnotationElementGet() ProAnnotationelemFeatureGet() ProAnnotationplaneAngleGet() ProAnnotationplaneFrozenGet() ProAnnotationplanePlaneGet() ProAnnotationplaneReferenceGet() ProAnnotationplaneTypeGet() ProAnnotationplaneVectorGet() ProAnnotationplaneViewnameGet() ProGtoldatumrefFree() ProGtoldatumrefGet()

Write and Modification Functions Supporting Inheritance Features

Dimensions	ProDimensionValueSet() ProDimensionToleranceSet() ProDimensionBoundSet()
Parameters	ProParameterValueSet()
Features	ProFeatureSuppress() ProFeatureResume() ProFeatureDelete()
Annotations	ProNoteTextSet() ProNoteURLSet() ProNoteElbowlengthSet() ProNoteTextstyleSet() ProTextStyle*Set() ProGtolValueSet() ProGtolPrefixSet() ProGtolSuffixSet() ProGtolTopTextSet() ProGtolRightTextSet() ProGtolTextstyleSet() ProGtoltextTextstyleSet() ProAnnotationShow() ProAnnotationErase() ProSurffinishValueSet() ProGtoldatumrefAlloc() ProGtoldataGtoldatumrefSet()

Assembly: Data
Sharing Features

Variant Model Items

Function Introduced:

- **ProVariantfeatItemVisit()**
- **ProVariantfeatItemStandardize()**
- **ProVariantfeatItemCopyGet()**
- **ProVariantfeatItemCopySet()**
- **ProVariantfeatItemCopyUnset()**

Use the function **ProVariantfeatItemsVisit()** to visit the variant items (dimensions, features, and annotations) owned by an inheritance feature or flexible component. The item handles will contain the variant feature model pointer.

Use the function **ProVariantfeatItemStandardize()** to remove a varied item from the inheritance feature or flexible component.

Use the function **ProVariantfeatItemCopyGet()** to obtain the copy flag for a given item in the inheritance feature or flexible component. The input arguments of this function are:

- *feature*—Specifies the variant feature
- *item*—Specifies the item.

The output arguments of this function are:

- *copy*—Specifies whether or not to copy the item into the variant feature. If this argument returns true, the item is copied into the feature. If it is false, the item is not copied. This value overrides the value of the features "copy all" flag.

Use the function **ProVariantfeatItemCopySet()** to assign the copy flag for a given item in the inheritance feature or flexible component.

Use the function **ProVariantfeatItemCopyUnset()** to remove the copy flag for a given item in the inheritance feature or flexible component.

Variant Parameters

Function Introduced:

- **ProVariantfeatParamsVisit()**
- **ProVariantfeatParamStandardize()**

Use the function **ProVariantfeatParamsVisit()** to visit only the variant parameters owned by an inheritance feature or flexible component. The parameter handles contain the variant feature model pointer.

Use the function **ProVariantfeatParamStandardize()** to remove a varied parameter from the inheritance feature or flexible component.

Variant References

By using variant references, you can reroute or replace the references for features in the inheritance feature with new references located inside or outside the inheritance feature. In assemblies, references can be to models other than the target model, as long as the model is within the assembly.

In Pro/TOOLKIT, a variant reference is represented by a *ProVariantRef* handle.

Function Introduced:

- **ProVariantrefAlloc()**
- **ProVariantrefOriginalrefGet()**
- **ProVariantrefOriginalrefSet()**
- **ProVariantrefReplacementrefGet()**
- **ProVariantrefReplacementrefSet()**
- **ProVariantrefFeatidsGet()**
- **ProVariantrefFeatidsSet()**
- **ProVariantfeatVariantrefsGet()**
- **ProVariantfeatVariantrefsSet()**
- **ProVariantrefFree()**
- **ProVariantrefProarrayFree()**

Use the function **ProVariantrefAlloc()** to allocate a handle used to describe a variant reference assigned to a variant feature (like inheritance features). The input arguments of this function are:

- *original_ref*—Specifies the initial reference handle.
- *replacement_ref*—Specifies the replacement reference handle.
- *feat_ids*—Specifies a **ProArray** of feature ids (from the base model) which will be assigned to the reference replacement action.

Use the function **ProVariantrefOriginalrefGet()** to obtain the original reference that is replaced.

Use the function **ProVariantrefOriginalrefSet()** to assign the original reference that is to be replaced.

Use the function **ProVariantrefReplacementrefGet()** to obtain the replacement reference for replacing the original reference in this variant feature.

Use the function **ProVariantrefReplacementrefSet()** to assign the replacement reference for replacing the original reference in this variant feature.

Use the function **ProVariantrefFeatidsGet()** to obtain an array of the feature ids taken from the base model which are assigned the replacement reference.

Use the function **ProVariantrefFeatidsSet()** to assign an array of the feature ids taken from the base model which are assigned the reference replacement.

Note: An assignment of a replacement reference to a feature applies only if the features actually use the replacement reference.

Use the function **ProVariantfeatVariantrefsGet()** to obtain the variant reference assignments (a *ProArray* of the variant reference) stored by this variant feature.

Use the function **ProVariantfeatVariantrefsSet()** to assign the variant reference assignments stored by this variant feature.

Use the function **ProVariantrefFree()** to free a handle used to describe a variant reference assigned to a variant feature and the function **ProVariantrefProarrayFree()** to free an array of handles used to describe a variant reference assigned to a variant feature.

51

Drawings

This chapter describes the Pro/TOOLKIT functions that deal with drawings. Unless otherwise specified, functions that operate on drawings use screen coordinates. See the 'Core: 3D Geometry' chapter to find out more about screen coordinates and how to convert to drawing coordinates (or paper coordinates).

Topic	Page
Creating Drawings from Templates	51 - 2
Diagnosing Drawing Creation Errors	51 - 4
Drawing Setup	51 - 6
Context in Drawing Mode	51 - 6
Access Drawing Location in Grid	51 - 7
Drawing Tree	51 - 7
Merge Drawings	51 - 11
Drawing Sheets	51 - 11
Drawing Views and Models	51 - 16
Detail Items	51 - 49
Drawing Symbol Groups	51 - 87
Drawing Edges	51 - 90
Drawing Tables	51 - 91
Drawing Dimensions	51 - 104

Creating Drawings from Templates

Function Introduced:

- **ProDrawingFromTpltCreate()**

Use of drawing templates simplifies drawing creation. Such templates contain drawing views with various properties such as:

- Cross section view
- Simplified representation
- Dimensions On or Off
- Repeat regions (tables based on BOM balloons)

Use the function **ProDrawingFromTpltCreate()** to create a drawing from a template and to return a structure containing any errors encountered during drawing creation. This function requires:

- New drawing name
- Name of existing template to use
- Solid model to use when creating drawing
- Drawing output options that specify how you want to view drawings output. Chose any or all from the following list:
 - PRODWGCREATE_DISPLAY_DRAWING—display new drawing in a window
 - PRODWGCREATE_SHOW_ERROR_DIALOG—display the template error dialog to the user
 - PRODWGCREATE_WRITE_ERRORS_TO_FILE—write the errors to a disk file

The function returns an error structure if any errors occur. The error structure contains an array of errors. Each error message may have:

- Error type
- Name of view where error occurred
- Name of drawing sheet where error occurred
- Name of the invalid or missing object

Example 1: Drawing Creation from a Template

```
/*-----*\
File: DrawingFromTpltCreate.c
\*-----*/
#include <ProToolkit.h>
#include <ProDrawing.h>
#include <TestError.h>

#define TEMPLATE_NAME "custom_template_c"

/*=====*\
FUNCTION: UserDrawingCreateDemo()
PURPOSE: Creates a drawing for the current model.
\*=====*/
int UsrDrawingCreateDemo()
{
    ProError      err;
    ProMdl        solid_mdl;
    ProMdlType    mdl_type;
    ProMdlData    data;
    ProModel      model;
    ProFileName   msgfil;
    ProName       predefined_template;
    ProName       name;
    ProDrawing    created_drawing;
    ProDwgcreateOptions options = (ProDwgcreateOptions)0;
    ProDwgcreateErrs errors;

    created_drawing = NULL;

/*-----*\
Set up the name of the message file.
\*-----*/
    ProStringToWstring (msgfil, "msg_ugdrawing.txt");

/*-----*\
Use the current model to create the drawing.
\*-----*/
    err = ProMdlCurrentGet (&solid_mdl);
    if (err != PRO_TK_NO_ERROR)
        return (err);
    err = ProMdlTypeGet (solid_mdl, &mdl_type);
    if (err != PRO_TK_NO_ERROR ||
        (mdl_type != PRO_MDL_PART && mdl_type != PRO_MDL_ASSEMBLY))
        return PRO_TK_INVALID_TYPE;

    err = ProMdlDataGet (solid_mdl, &data);
    ProUtilWstrcpy (model.name, data.name);
    ProUtilWstrcpy (model.type, data.type);
```

```

/*-----*\
    Initialize the template name.
/*-----*\
    ProStringToWstring (predefined_template, TEMPLATE_NAME);

/*-----*\
    Read in the root of the new name of the drawings to create.
/*-----*\
    err = ProMessageDisplay (msgfil, "USER Enter new drawing name: ");
    err = ProMessageStringRead (PRO_NAME_SIZE, name);
    if (err != PRO_TK_NO_ERROR)
        return (err);

/*-----*\
    Create the required drawing.
/*-----*\
    err = ProDrawingFromTpltCreate (name,
                                   predefined_template,
                                   &model, options, &created_drawing,
                                   &errors);
    if (err != PRO_TK_NO_ERROR || err != PRO_TK_DWGCREATE_ERRORS)
        return err;
    if (err == PRO_TK_DWGCREATE_ERRORS)
    {
        ProMessageDisplay (msgfil,
                           "USER Errors found when creating drawing %w",
                           name);
    }

    return (PRO_TK_NO_ERROR);
}

```

Diagnosing Drawing Creation Errors

Functions Introduced:

- **ProDwgcreateErrsFree()**
- **ProDwgcreateErrsCountGet()**
- **ProDwgcreateErrTypeGet()**
- **ProDwgcreateErrViewNameGet()**
- **ProDwgcreateErrSheetGet()**
- **ProDwgcreateErrViewGet()**
- **ProDwgcreateErrObjNameGet()**

The function **ProDwgcreateErrsFree()** frees an existing errors table.

Use the function **ProDwgcreateErrsCountGet()** to return the number of drawing creation errors in the table.

The function **ProDwgcreateErrTypeGet()** returns the type of a drawing creation error.

Use the function **ProDwgcreateErrViewNameGet()** returns the name of the template view where the error occurred.

The function **ProDwgcreateErrSheetGet()** returns the drawing sheet number where the error occurred.

Use the function **ProDwgcreateErrViewGet()** returns the drawing view where the error occurred. This function is valid for the following error types:

- PRODWGCRTErr_EXPLODE_DOESNT_EXIST
- PRODWGCRTErr_MODEL_NOT_EXPLODABLE
- PRODWGCRTErr_SEC_NOT_PERP
- PRODWGCRTErr_NO_RPT_REGIONS
- PRODWGCRTErr_FIRST_REGION_USED
- PRODWGCRTErr_NOT_PROCESS_ASSEM
- PRODWGCRTErr_TEMPLATE_USED
- PRODWGCRTErr_SEC_NOT_PARALLEL
- PRODWGCRTErr_SIMP_REP_DOESNT_EXIST

ProDwgcreateErrObjNameGet() returns the name of the model invalid. This function is valid for the following error types:

- PRODWGCRTErr_SAVED_VIEW_DOESNT_EXIST
- PRODWGCRTErr_X_SEC_DOESNT_EXIST
- PRODWGCRTErr_EXPLODE_DOESNT_EXIST
- PRODWGCRTErr_SEC_NOT_PERP
- PRODWGCRTErr_SEC_NOT_PARALLEL
- PRODWGCRTErr_SIMP_REP_DOESNT_EXIST

Drawing Setup

Functions Introduced:

- **ProInputFileRead()**
- **ProOutputFileWrite()**
- **ProDrawingSetupOptionGet()**
- **ProDrawingSetupOptionSet()**

You can set all drawing setup file options (documented in the *Drawing User's Guide*) from a Pro/TOOLKIT application. To do this, import a text file in the format of the drawing setup file using the function **ProInputFileRead()**, with the file type set to PRO_DWG_SETUP_FILE. You can create such a file from Pro/TOOLKIT with function **ProOutputFileWrite()**. See the Interface: Data Exchange chapter for information on these functions.

Use the functions **ProDrawingSetupOptionGet()** and **ProDrawingSetupOptionSet()** to return and assign a specific drawing setup file option.

Context in Drawing Mode

Functions Introduced:

- **ProRibbonContextGet()**
- **ProRibbonContextSet()**

The functions **ProRibbonContextGet()** and **ProRibbonContextSet()** retrieve and set the context for the specified drawing window, respectively.

From Pro/ENGINEER Wildfire 5.0 onward, you can specify a context for a drawing. A context presents a set of commands relevant to the type of task that you are currently performing on a drawing. The contexts in the Drawing mode are Layout, Table, Annotate, Sketch, Autobuildz, Review, and Publish.

Note: To set the context to Autobuildz, you must first set the configuration option *autobuildz_enabled* to *yes*.

One of the arguments to the **ProDrawingContextGet()** and **ProDrawingContextSet()** functions is *context*, which is of the type *ProRibbonContext*. *ProRibbonContext* is an enumerated type, which takes the following values:

- PRO_RBNCONTEXT_DWG_NONE
- PRO_RBNCONTEXT_DWG_LAYOUT
- PRO_RBNCONTEXT_DWG_TABLE
- PRO_RBNCONTEXT_DWG_ANNOTATE
- PRO_RBNCONTEXT_DWG_SKETCH
- PRO_RBNCONTEXT_DWG_AUTOBUILDZ
- PRO_RBNCONTEXT_DWG_REVIEW
- PRO_RBNCONTEXT_DWG_PUBLISH

Note: If you set *context* to PRO_RBNCONTEXT_DWG_NONE, all actions for the selected object are available in all contexts, except the contexts in which selection is not allowed. For example, the Publish context.

Access Drawing Location in Grid

Function Introduced:

- **ProDrawingPosToLocgrid()**

Use the function **ProDrawingPosToLocgrid()** to find the grid coordinates of a location in the specified drawing. The function specifies the position of a point, expressed in screen coordinates. **ProDrawingPosToLocgrid()** returns strings representing the row and column containing the point.

Drawing Tree

Pro/ENGINEER Wildfire 5.0 onward, you can view a hierarchical representation of drawing items in an active drawing sheet. This representation is called a Drawing Tree. A drawing tree facilitates selection and availability of drawing operations for the represented drawing items. For more information, see the Pro/ENGINEER Help. Use the following functions to refresh, expand, and collapse the drawing tree:

Functions Introduced:

- **ProDrawingtreeRefresh()**
- **ProDrawingtreeExpand()**
- **ProDrawingtreeCollapse()**

Use the function **ProDrawingtreeRefresh()** to rebuild the drawing tree in the Pro/ENGINEER window that contains the drawing. You can use this function after adding a single drawing item or multiple drawing items to a drawing.

Use the function **ProDrawingtreeExpand()** to expand the drawing tree in the Pro/ENGINEER window and make all drawing sheets and drawing items in the active drawing sheet visible.

Use the function **ProDrawingtreeCollapse()** to collapse all nodes of the drawing tree in the Pro/ENGINEER window and make its child nodes invisible.

The input arguments to these functions are:

- *drawing*—Handle to the drawing that contains the drawing tree.
- *window_id*—ID of the Pro/ENGINEER window in which you want to refresh, expand, or collapse the drawing tree.

Note: Use PRO_VALUE_UNUSED to refresh, expand, or collapse the drawing tree in the active window.

Example 2: Performing Operations on a Drawing Tree

The following example shows how to expand, collapse, and refresh the drawing tree in the selected drawing.

```
/-----\  
FILE      : UgDrwTree.c  
PURPOSE   : Pro/TOOLKIT User Guide Example -Perform operations on current  
drawing  
\-----/  
  
/*----- Pro/Toolkit includes -----*/  
#include <ProToolkit.h>  
#include <ProMdl.h>  
#include <ProCsys.h>  
#include <PTApplsUnicodeUtils.h>  
#include <ProDrawing.h>  
#include <ProWindows.h>  
#include <ProSelection.h>  
  
/*----- Application includes -----*/
```

```
#include <TestError.h>

/*----- Function prototypes -----*/

extern ProError UserTreeExpand();
extern ProError UserTreeCollapse();
extern ProError UserTreeRefresh();

ProError status;
ProDrawing drawing;
ProMdlType mdl_type;
ProCharLine astr;
int win_id = -1;
ProFileName WMSGFIL = {'m','s','g','_','u','g','d','w','g','.','.',
                       't','x','t','\0'};

/*=====*\
FUNCTION   : UserTreeExpand
PURPOSE    : Expand the drawing tree in the current drawing.
\*=====*/
ProError UserTreeExpand ()
{

/*-----*\
  Get the current drawing
\*-----*/
  status = ProMdlCurrentGet((ProMdl*)&drawing);
  ERROR_CHECK ("UserTreeExpand","ProMdlCurrentGet",status);
  if (status != PRO_TK_NO_ERROR)
    return (status);

/*-----*\
  Check the type of current drawing
\*-----*/
  status = ProMdlTypeGet((ProMdl)drawing, &mdl_type);
  ERROR_CHECK ("UserTreeExpand","ProMdlTypeGet",status);

  if (status != PRO_TK_NO_ERROR || (mdl_type != PRO_MDL_DRAWING))
  {
    ProTKSprintf (astr, "Current model is not drawing", status);
    status = ProMessageDisplay (WMSGFIL,"USER %0s", astr);
    ERROR_CHECK("UserTreeExpand", "ProMessageDisplay",status);
    return(PRO_TK_GENERAL_ERROR);
  }

  status = ProDrawingtreeExpand(drawing, win_id);
  ERROR_CHECK ("UserTreeExpand","ProDrawingtreeExpand",status);

  return (status);
}
```

```

/*=====*\
FUNCTION   : UserTreeCollapse
PURPOSE    : Collapse the drawing tree in the current drawing.
\*=====*/
ProError UserTreeCollapse ()
{

/*-----*\
    Get the current drawing
\*-----*/
    status = ProMdlCurrentGet((ProMdl*)&drawing);
    ERROR_CHECK ("UserTreeExpand", "ProMdlCurrentGet", status);
    if (status != PRO_TK_NO_ERROR)
        return (status);

    status = ProMdlTypeGet((ProMdl)drawing, &mdl_type);
    ERROR_CHECK ("UserTreeExpand", "ProMdlTypeGet", status);

    if (status != PRO_TK_NO_ERROR || (mdl_type != PRO_MDL_DRAWING))
    {
        ProTKSprintf (astr, "Current model is not drawing", status);
        status = ProMessageDisplay (WMSGFIL, "USER %0s", astr);
        ERROR_CHECK("UserTreeExpand", "ProMessageDisplay", status);
        return(PRO_TK_GENERAL_ERROR);
    }

    status = ProDrawingtreeCollapse(drawing, win_id);
    ERROR_CHECK ("UserTreeExpand", "ProDrawingtreeExpand", status);

    return (status);
}

/*=====*\
FUNCTION   : UserTreeRefresh
PURPOSE    : Refresh the current drawing tree.
\*=====*/
ProError UserTreeRefresh ()
{

/*-----*\
    Get the current drawing
\*-----*/
    status = ProMdlCurrentGet((ProMdl*)&drawing);
    ERROR_CHECK ("UserTreeExpand", "ProMdlCurrentGet", status);
    if (status != PRO_TK_NO_ERROR)
        return (status);

    status = ProMdlTypeGet((ProMdl)drawing, &mdl_type);
    ERROR_CHECK ("UserTreeExpand", "ProMdlTypeGet", status);

    if (status != PRO_TK_NO_ERROR || (mdl_type != PRO_MDL_DRAWING))

```

```

    {
        ProTKSprintf (astr, "Current model is not drawing", status);
        status = ProMessageDisplay (WMSGFIL,"USER %0s", astr);
        ERROR_CHECK("UserTreeExpand", "ProMessageDisplay",status)
        return(PRO_TK_GENERAL_ERROR);
    }
    status = ProDrawingtreeRefresh(drawing, win_id);
    ERROR_CHECK ("UserTreeExpand", "ProDrawingtreeExpand",status);

    return (status);
}

```

Merge Drawings

Function Introduced:

- **ProDrawingMerge()**

Use the function **ProDrawingMerge()** to merge two drawings.

Drawing Sheets

Functions Introduced:

- **ProDrawingSheetNameGet()**
- **ProDrawingSheetsCount()**
- **ProDrawingSheetTrfGet()**
- **ProDrawingSheetSizeGet()**
- **ProDrawingSheetOrientationGet()**
- **ProDrawingSheetUnitsGet()**
- **ProDrawingCurrentSheetGet()**
- **ProDrawingCurrentSheetSet()**
- **ProDrawingSheetCreate()**
- **ProDrawingSheetDelete()**
- **ProDrawingSheetCopy()**
- **ProDrawingSheetsReorder()**
- **ProDrawingSheetFormatIsBlanked()**
- **ProDrawingFormatGet()**

- **ProDrawingFormatAdd()**
- **ProDwgSheetRegenerate()**

Drawing sheets are identified in Pro/TOOLKIT by the same sheet numbers the Pro/ENGINEER user sees.

The function **ProDrawingSheetNameGet()** retrieves the name of the specified drawing sheet.

The function **ProDrawingSheetsCount()** outputs the current number of sheets in the specified drawing.

The function **ProDrawingSheetTrfGet()** provides the matrix that transforms from screen to drawing units or vice versa. (See the Core: Coordinate Systems and Transformations chapter for more details on transformations.)

The function **ProDrawingSheetSizeGet()** returns the size of the sheet used for the drawing. If the sheet is a standard paper size, this will be returned, along with the width and height of the sheet measured in the sheet units.

The function **ProDrawingSheetOrientationGet()** returns the orientation of the drawing sheet. The orientation can be either portrait or landscape.

The function **ProDrawingSheetUnitsGet()** retrieves the name of the length units used for measuring the drawing sheet.

The function **ProDrawingCurrentSheetGet()** returns the sheet number of the currently selected sheet in the specified drawing. It returns a positive integer for the retrieved sheet number and a zero for an invalid input argument.

The function **ProDrawingCurrentSheetSet()** sets the sheet number for the currently selected sheet in the specified drawing. One of the input arguments to this function is an integer, *current_sheet*.

The function **ProDrawingSheetCreate()** adds a sheet to a drawing model of type Layout, Format, Drawing, Diagram, or Report. The output argument to this function is an integer, *new_sheet*. If the drawing sheet is successfully created, *new_sheet* takes the value of the sheet number for the newly created drawing sheet. Otherwise it takes the value, -1.

The function **ProDrawingSheetDelete()** deletes a sheet from a drawing model of type Layout, Format, Drawing, Diagram, or Report. You need to enter the type of the drawing model and the sheet number of the sheet to be deleted as input arguments.

The function **ProDrawingSheetCopy()** creates a copy of a specified drawing sheet. The input arguments to this function are:

- *drawing*—The drawing model handle. Set it to NULL to copy a sheet from the current drawing.
- *sheet_to_copy*—The sheet number of the sheet to be copied. Set it to a value less than 1 to create a copy of the currently selected sheet.

The output argument to this function is:

- *sheet_copy* (optional)—The sheet number of the newly created sheet. Set it to NULL, if you do not want the new sheet number.

The function **ProDrawingSheetsReorder()** assigns a new sheet number to a specified sheet, and renumbers the remaining sheets accordingly.

The function **ProDrawingSheetFormatIsBlanked()** identifies whether the format of a specified drawing sheet is blank. In case of the current sheet, set the input arguments *drawing* and *sheet* to NULL and to a value less than 1 respectively.

The function **ProDrawingFormatGet()** outputs the name of the drawing format that was used for the specified sheet.

ProDrawingFormatAdd() adds or replaces a specified format into a specified drawing sheet.

The function **ProDwgSheetRegenerate()** regenerates a specified drawing sheet.

Example 3: Listing Drawing Sheets

The following example shows how to list the sheets in the current drawing.

```
#include <ProToolkit.h>
#include <ProDrawing.h>
#include <pd_prototype.h>
#include <PTApplsUnicodeUtils.h>
/*=====*\
FUNCTION : UsrListSheets()
PURPOSE  : Command to list sheet info in an information window
\*=====*/
int UsrListSheets()
{
    ProDrawing drawing;
    int n_sheets, sheet;
    ProName wformat;
    ProCharName format;
    ProUnititem units;
```

```

double width;
double height;
char unit [PRO_NAME_SIZE];
FILE *fp;
ProName wfname;
char *fname = "sheets.txt";
/*-----*\
Open a text file to contain the information to be displayed
\*-----*/
    fp = PTApplsUnicodeFopen(fname, "w");
/*-----*\
Get the current drawing
\*-----*/
    ProMdlCurrentGet((ProMdl*)&drawing);
/*-----*\
Get the number of sheets
\*-----*/
    ProDrawingSheetsCount(drawing, &n_sheets);
    for(sheet=1;sheet<=n_sheets;sheet++)
    {
/*-----*\
Get the drawing sheet size etc.
\*-----*/
        ProDrawingSheetSizeGet (drawing, sheet, NULL, &width, &height);
        ProDrawingSheetUnitsGet (drawing, sheet, &units);
/*-----*\
Get the format used, if any
\*-----*/
        if (ProDrawingFormatGet(drawing, sheet, wformat) == PRO_TK_NO_ERROR)
            ProWstringToString(format, wformat);
        else
            strcpy(format, "(NONE)");
/*-----*\
Print the information to the text file
\*-----*/
        ProWstringToString (unit, units.name);
        ProTKFprintf(fp, "Sheet %d\n", sheet);
        ProTKFprintf(fp, "    Width  : %lf %s\n", width, unit);
        ProTKFprintf(fp, "    Height : %lf %s\n", height, unit);
        ProTKFprintf(fp, "    Format : %s\n", format);
    }
/*-----*\
Close the file, and display it
\*-----*/
    fclose(fp);
    ProStringToWstring(wfname, fname);
    ProInfoWindowDisplay(wfname, NULL, NULL);
    return 0;
}

```


Example 4: Creating a Copy of the Current Drawing Sheet

```
\*****/

/*----- Pro/Toolkit includes -----*/
#include <ProToolkit.h>
#include <ProMdl.h>
#include <ProCsys.h>

/*----- Application includes -----*/
#include <TestError.h>

/*=====*\
FUNCTION   : UserSheetCopy()
PURPOSE    : Adds a drawing sheet to the current drawing-type object
\*=====*/
ProError UserSheetCopy ()
{
    ProError      status;
    int           win_id, sheet_num;
    ProCharLine   astr;
    ProDrawing    drawing;
    ProMdl mdl;
    int sheet_to_copy, sheet_copy;
    ProMdlType type;
    ProFileName WMSGFIL =
        {'m','s','g','_','u','g','d','w','g','.','t','x','t','\0'};

    ProTKSprintf (astr, "Inside UserSheetCopy()");
    status = ProMessageDisplay (WMSGFIL,"USER %0s", astr);

    /*-----*\
    Get the current model
    \*-----*/
    status = ProMdlCurrentGet(&mdl);
    ERROR_CHECK("UserSheetCopy", "ProMdlCurrentGet", status);
    if (status != PRO_TK_NO_ERROR) return(status);

    status = ProMdlTypeGet(mdl, &type);
    ERROR_CHECK("UserSheetCopy", "ProMdlTypeGet", status);
    if (status != PRO_TK_NO_ERROR) return(status);

    if(type != PRO_DRAWING)
    {
        ProTKSprintf (astr, "Current model is not a drawing!", status);
        status = ProMessageDisplay (WMSGFIL,"USER %0s", astr);
        ERROR_CHECK("UserSheetCopy", "ProMessageDisplay", status);
        return(PRO_TK_GENERAL_ERROR);
    }
    /*-----*\
    Create a copy of the current drawing sheet.
```

```

/*-----*/

sheet_to_copy = -1;

status = ProDrawingSheetCopy((ProDrawing)mdl, sheet_to_copy,
                             &sheet_copy);
ERROR_CHECK("UserSheetCopy", "ProDrawingSheetCopy",
            (status != PRO_TK_NO_ERROR));

if (status != PRO_TK_NO_ERROR) return (status);

/*-----*\
   Print a message in the message window
\*-----*/
if (status != PRO_TK_NO_ERROR)
{
    ProTKSprintf (astr, "Error %d while adding sheet", status);
    status = ProMessageDisplay (WMSGFIL,"USER %0s", astr);
    ERROR_CHECK("UserSheetCopy", "ProMessageDisplay",status);
    return(PRO_TK_GENERAL_ERROR);
}

ProTKSprintf (astr, "Sheet No. %0d added to drawing.", sheet_copy);
status = ProMessageDisplay (WMSGFIL,"USER %0s", astr);
ERROR_CHECK("UserSheetCopy", "ProMessageDisplay",status);

/*-----*\
   Update the changes by repainting the window
\*-----*/
status = ProWindowCurrentGet(&win_id);
ERROR_CHECK("UserSheetCopy", "ProWindowCurrentGet",status);
if (status != PRO_TK_NO_ERROR) return(status);

status = ProWindowRepaint (win_id);
ERROR_CHECK("UserSheetCopy", "ProWindowRepaint",status);
return(status);
}

```

Drawing Views and Models

Drawing views are identified by the OHandle *ProView*. This is the same object handle used to reference 3D model views, accessed by the functions in *ProView.h*, but there are no cases where the same object can be accessed by both types of function. The general rule is that functions for 3D views are in *ProView.h*, and start with *ProView*; functions to manipulate drawing views are in *ProDrawing.h* and therefore always start with *ProDrawing*.

Each drawing view has a solid attached to it. The same solid can appear in more than one view. Some solids may be attached to the drawing, but not appear in any view, if the Pro/ENGINEER user has deleted all views of it without also removing the solid itself from the drawing.

Listing Drawing Views

Functions Introduced:

- **ProDrawingViewsCollect()**
- **ProDrawingViewVisit()**
- **ProDrawingViewSheetGet()**
- **ProDrawingViewOutlineGet()**
- **ProDrawingViewTypeGet()**
- **ProDrawingViewParentGet()**
- **ProDrawingViewChildrenGet()**
- **ProDrawingViewOriginGet()**
- **ProDrawingViewAlignmentGet()**
- **ProDrawingViewScaleIsUserdefined()**
- **ProDrawingViewScaleGet()**
- **ProDrawingScaleGet()**
- **ProDrawingViewPerspectiveScaleGet()**
- **ProDrawingViewFlagGet()**
- **ProDrawingViewDisplayGet()**
- **ProDrawingViewColorSourceGet()**
- **ProDrawingViewSolidGet()**
- **ProDrawingViewTransformGet()**
- **ProDrawingViewNameGet()**
- **ProDrawingViewZclippingGet()**
- **ProDrawingViewDatumdisplayGet()**
- **ProDrawingViewPipingdisplayGet()**
- **ProDrawingViewIsErased()**
- **ProDrawingViewNeedsRegen()**

ProDrawingViewsCollect() outputs an array of *ProView* handles to all the views in a drawing. **ProDrawingViewVisit()** is an alternative way to find the views, and conforms to the usual form of visit functions. **ProDrawingViewSheetGet()** outputs the number of the sheet in which a specified view appears.

ProDrawingViewOutlineGet() outputs the position of the view in the sheet.

The function **ProDrawingViewTypeGet()** returns the type of a specified drawing view. A drawing view can be of the following types:

- **PRO_VIEW_GENERAL**—Specifies a general drawing view.
- **PRO_VIEW_PROJECTION**—Specifies a projected drawing view.
- **PRO_VIEW_AUXILIARY**—Specifies an auxiliary drawing view.
- **PRO_VIEW_DETAIL**—Specifies a detailed drawing view.
- **PRO_VIEW_REVOLVE**—Specifies a revolved drawing view.
- **PRO_VIEW_COPY_AND_ALIGN**—Specifies a copy and align drawing view.
- **PRO_VIEW_OF_FLAT_TYPE**—Specifies a flat type drawing view.

The function **ProDrawingViewParentGet()** returns the parent view of a specified drawing view. The function **ProDrawingViewChildrenGet()** returns the child views of a drawing view and the total number of children.

The function **ProDrawingViewOriginGet()** returns the location of the origin in terms of the *ProPoint3d* object and the selection reference in terms of the *ProSelection* object for a specified drawing view.

The function **ProDrawingViewAlignmentGet()** returns the alignment of a drawing view with respect to another view. It returns the alignment style of the drawing view that can be either horizontal or vertical, the reference view to which the view is aligned, and the alignment references of both the aligned and reference views.

ProDrawingViewScaleIsUserdefined() returns a boolean value depending on whether the drawing has a user-assigned scale or not.

ProDrawingViewScaleGet() returns the scale factor applied, even if the view is not a scaled view. **ProDrawingScaleGet()** returns the overall drawing scale that is applied to all unscaled views.

The function **ProDrawingViewPerspectiveScaleGet()** returns the perspective scale applied to a drawing view. This scale option is available only for general views. The function returns the following output arguments:

- *eye_dist*—Specifies the eye-point distance from model space.
- *view_dia*—Specifies the view diameter in paper units such as mm.

The function **ProDrawingViewFlagGet()** identifies if the projection arrow flag has been set for a projected or detailed drawing view.

ProDrawingViewDisplayGet() outputs the *ProDrawingViewDisplay* structure that describes the display status of the drawing view. The fields in the structure, all either enums or booleans, are listed below:

- *style*—Whether wireframe, hidden line, or shaded.
- *quilt_hlr*—Whether hidden-line-removal is applied to quilts.
- *tangent_edge_display*—Style of line used for tangent edges.
- *cable_display*—Whether cables are shown by centerline, as thick, or using the current default.
- *concept_model*—Whether the skeleton is displayed.
- *weld_xsec*—Whether welds are included in the cross-section.

The function **ProDrawingViewColorSourceGet()** returns the color source of the drawing view representation. The color source can be of the following types:

- **PRO_VIEW_MODEL_COLOR**—Specifies that the drawing colors are determined by the model settings.
- **PRO_VIEW_DRAWING_COLOR**—Specifies that the drawing colors are determined by the drawings settings.

ProDrawingViewSolidGet() provides the handle to the solid that is being displayed in the drawing view.

ProDrawingViewTransformGet() returns the matrix that describes the transform between 3d solid coordinates and 2D screen coordinates for a specified view.

ProDrawingViewNameGet() returns the name of a specified view in the drawing.

ProDrawingViewZclippingGet() returns the reference of the Z-clipping on the drawing view. The reference can be an edge, datum, or point on the surface that is parallel to the view. Geometry contained in the Z-clipping plane and in front of the plane appears, but geometry behind the plane does not appear. The system clips geometry that intersects the plane.

The function **ProDrawingViewDatumdisplayGet()** determines if a solid model datum has been explicitly shown in a particular drawing view.

The function **ProDrawingViewPipingdisplayGet()** returns the piping display option for a drawing view.

The function **ProDrawingViewIsErased()** identifies if the drawing view is erased or not.

The function **ProDrawingViewNeedsRegen()** identifies whether the drawing or the specified drawing view needs to be regenerated.

Example 5: Listing the Views in a Drawing

The following example shows a command that creates an information window reporting information about all the views in a drawing.

```
#include <ProToolkit.h>
#include <ProDrawing.h>
#include <ProArray.h>
#include <UtilMatrix.h>

/*=====*\
FUNCTION : UsrListViews()
PURPOSE  : Command to list view info in an information window
\*=====*/
int UsrListViews()
{
    ProDrawing drawing;
    int n_views, view, sheet;
    ProSolid solid;
    ProMdldata mdata;
    ProVector outline[2];
    double scale;
    ProDrawingViewDisplay display;
    char *sstyle;
    ProView *views;
    FILE *fp;
    ProName wfname;
```

```

    ProCharName name, type;
    char *fname = "views.txt";
/*-----*/
    Open a text file to contain the information to be displayed
/*-----*/
    fp = PTAplUnicodeFopen(fname, "w");
/*-----*/
    Get the current drawing
/*-----*/
    ProMdlCurrentGet((ProMdl*)&drawing);
/*-----*/
    Collect the views into an array
/*-----*/
    ProDrawingViewsCollect(drawing, &views);
    ProArraySizeGet(views, &n_views);
    for(view=0;view<n_views;view++)
    {
/*-----*/
    Get the sheet number for this view
/*-----*/
        ProDrawingViewSheetGet(drawing, views[view], &sheet);
/*-----*/
    Get the name of the solid that the view contains
/*-----*/
        ProDrawingViewSolidGet(drawing, views[view], &solid);
        ProMdlDataGet((ProMdl)solid, &mdata);
        ProWstringToString(name, mdata.name);
        ProWstringToString(type, mdata.type);
/*-----*/
    Get the outline, scale, and display state
/*-----*/
        ProDrawingViewOutlineGet(drawing, views[view], outline);
        ProDrawingViewScaleGet(drawing, views[view], &scale);
        ProDrawingViewDisplayGet(drawing, views[view], &display);
/*-----*/
    Write the information to the text file
/*-----*/
        ProTKFprintf(fp, "View %d\n", view+1);
        ProTKFprintf(fp, "    Solid      : %s.%s.%d\n", name, type,
                        mdata.version);
        ProTKFprintf(fp, "    Sheet      : %d\n", sheet);
        ProTKFprintf(fp, "    Lower left : %0.3f, %0.3f\n", outline[0][0],
                        outline[0][1]);
        ProTKFprintf(fp, "    Upper right : %0.3f, %0.3f\n", outline[1][0],
                        outline[1][1]);
        ProTKFprintf(fp, "    Scale      : %0.3f\n", scale);
        switch(display.style)
        {
            case PRO_DISPSTYLE_DEFAULT      : sstyle = "default"; break;
            case PRO_DISPSTYLE_WIREFRAME    : sstyle = "wireframe"; break;
            case PRO_DISPSTYLE_HIDDEN_LINE : sstyle = "hidden line"; break;

```

```

        case PRO_DISPSTYLE_NO_HIDDEN    : sstyle = "no hidden";break;
        case PRO_DISPSTYLE_SHADED      : sstyle = "shaded"; break;
    }
    ProTKFprintf(fp,"    Disp style    : %s\n", sstyle);
}
/*-----*\
Close the file, and display it
\*-----*/
fclose(fp);
ProStringToWstring(wfname, fname);
ProInfoWindowDisplay(wfname, NULL, NULL);
return 0;
}

```

Modifying Views

Functions Introduced:

- **ProDrawingViewMove()**
- **ProDrawingViewDelete()**
- **ProDrawingViewErase()**
- **ProDrawingViewResume()**
- **ProDrawingViewOriginSet()**
- **ProDrawingViewAlignmentSet()**
- **ProDrawingViewScaleSet()**
- **ProDrawingScaleSet()**
- **ProDrawingViewProjectionSet()**
- **ProDrawingViewFlagSet()**
- **ProDrawingViewDisplaySet()**
- **ProDrawingViewNameSet()**
- **ProDrawingViewZclippingSet()**
- **ProDrawingViewPipingdisplaySet()**
- **ProDrawingViewRegenerate()**
- **ProDwgViewRegenerate()**

The function **ProDrawingViewMove()** moves the specified drawing view and its child views by a vector to a new position in screen coordinates. This function performs the same operation as the Pro/ENGINEER command **Edit > Move Special**.

The function **ProDrawingViewDelete()** deletes the drawing view and any of its child views, or fails if the view has any children.

The function **ProDrawingViewErase()** erases a specified drawing view.

The function **ProDrawingViewResume()** resumes the specified drawing view.

The function **ProDrawingViewOriginSet()** assigns the location of the origin in terms of the *ProPoint3d* object and the selection reference in terms of the *ProSelection* object for a specified drawing view.

The function **ProDrawingViewAlignmentSet()** assigns the alignment of a drawing view with respect to another view. This function assigns the following input arguments related to alignment:

- *view_reference*—Specifies the reference view to which the drawing view is aligned.
- *align_style*—Specifies the alignment style in terms of the enumerated type *ProDrawingViewAlignStyle*. The alignment style can be either horizontal or vertical. In case of horizontal alignment, the drawing view and the view it is aligned to lie on the same horizontal line. In case of vertical alignment, the drawing view and the view it is aligned to lie on the same vertical line.
- *align_ref_1*—Specifies the alignment reference of the referenced view. If this is set to NULL, the reference view is aligned according to its view origin.
- *align_ref_2*—Specifies the alignment reference of the aligned view. If this is set to NULL, the aligned view is aligned according to its view origin.

The function **ProDrawingViewScaleSet()** modifies the scale of a scaled view; **ProDrawingScaleSet()** modifies the overall drawing scale that is applied to all unscaled views.

The function **ProDrawingViewProjectionSet()** assigns the specified drawing view as a projection.

The function **ProDrawingViewFlagSet()** sets the projection arrow flag to TRUE for a projected or detailed drawing view.

The function **ProDrawingViewDisplaySet()** assigns the fields in the *ProDrawingViewDisplay* structure that describe the display status of the drawing view. This function does not repaint the drawing view, use the function **ProWindowRepaint()** to repaint the view.

Note: In order to modify the *concept_model* field in the *ProDrawingViewDisplay* structure, you require a Pro/ASSEMBLY license.

The function **ProDrawingViewNameSet()** assigns a name to a specified drawing view.

The function **ProDrawingViewZclippingSet()** sets the Z-clipping on the drawing view to reference a given edge, datum, or point on the surface that is parallel to the view. Geometry contained in the Z-clipping plane and in front of the plane appears, but geometry behind the plane does not appear. The system clips geometry that intersects the plane.

The function **ProDrawingViewPipingdisplaySet()** assigns the piping display option for a drawing view. The piping display options are as follows:

- **PRO_PIPINGDISP_DEFAULT**—Displays the default appearance of pipes for the piping assembly.
- **PRO_PIPINGDISP_CENTERLINE**—Displays pipes as centerlines without insulation.
- **PRO_PIPINGDISP_THICK_PIPES**—Displays thick pipes without insulation.
- **PRO_PIPINGDISP_THICK_PIPES_AND_INSULATION**—Displays thick pipes and insulation.

The function **ProDrawingViewRegenerate()** regenerates the drawing view specified by the *ProView* view handle.

The function **ProDwgViewRegenerate()** erases the displayed view of the current object, regenerates the view from the current drawing, then redisplay the view.

Creating Views

Function Introduced:

- **ProDrawingGeneralviewCreate()**
- **ProDrawingProjectedviewCreate()**

A general view is usually the first view placed on a drawing sheet. It is the most versatile view, that is, it can be scaled or rotated to any setting.

A projected view is an orthographic projection of another view's geometry along a horizontal or vertical direction.

The above functions create general and projected drawing views.

Example 6: Creating General and Projected Drawing Views

The following example code adds a new sheet to a drawing, and creates one general and two projected views of a selected model.

```
#include <ProToolkit.h>
#include <ProDrawing.h>
#include <ProArray.h>
#include <UtilMatrix.h>
/*=====*\
FUNCTION : UsrSolidSelect()
PURPOSE : Utility to select a solid using the file browser and retrieve
          it if it is not already in session.
\*=====*/
int UsrSolidSelect(
    ProSolid *solid)
{
    ProName title;
    ProLine filter;
    ProPath wfile, wdir, wcurrent_dir;
    char file[PRO_PATH_SIZE], dir[PRO_PATH_SIZE],
        current_dir[PRO_PATH_SIZE];
    ProBoolean different_dir;
    char *slash, *root, *ext;
    ProFamilyName wroot;
    ProMdlType mtype;
    ProError status;
/*-----*\
    Ask the user to select a prt or asm file
\*-----*/
    ProStringToWstring(title, "Solid");
    ProStringToWstring(filter, "*.prt,*.asm");
    if(ProFileOpen(title, filter, NULL, NULL, NULL, NULL, wfile) !=
        PRO_TK_NO_ERROR)
        return(0);
    ProWstringToString(file, wfile);
/*-----*\
    Parse out the directory, and see if it is different from the current
\*-----*/
    different_dir = PRO_B_FALSE;
    if(slash = strrchr(file, '/'))
    {
```

```

        *slash = '\\0';
        strcpy(dir, file);
        strcpy(file, ++slash);
        ProStringToWstring(wdir, dir);
        ProDirectoryCurrentGet(wcurrent_dir);
        ProWstringToString(current_dir, wcurrent_dir);
        if(strcmp(current_dir, dir))
            different_dir = PRO_B_TRUE;
    }
/*-----*\
Parse out the file root name and model type
\*-----*/
    root = strtok(file, ".");
    ProStringToWstring(wroot, root);
    ext = strtok(NULL, ".");
    if(!strcmp(ext, "prt"))
        mtype = PRO_MDL_PART;
    else if(!strcmp(ext, "asm"))
        mtype = PRO_MDL_ASSEMBLY;
    else
        return(0);
/*-----*\
If the solid is already in session, return with it
\*-----*/
    if(ProMdlInit(wroot, mtype, (ProMdl*)solid) == PRO_TK_NO_ERROR)
        return(1);
/*-----*\
Move the to correct directory, and try to retrieve the solid
\*-----*/
    if(different_dir)
        ProDirectoryChange(wdir);
    status = ProMdlRetrieve(wroot, mtype, (ProMdl*)solid);
    if(different_dir)
        ProDirectoryChange(wcurrent_dir);
    if(status != PRO_TK_NO_ERROR)
        return(0);
    return(1);
}
/*=====*\
FUNCTION : UsrCreateSheet()
PURPOSE  : Create a new drawing sheet with a general, and two
           projected, views of a selected solid
\*=====*/
int UsrCreateSheet()
{
    ProDrawing drawing;
    int sheet;
    ProSolid solid;
    ProVector pos;
    ProView view, pview;
    ProVector outline[2];

```

```

    ProMatrix matrix;
/*-----*\
Create a new sheet and make it current
/*-----*\
    ProMdlCurrentGet((ProMdl)&drawing);
    ProDrawingSheetCreate(drawing, &sheet);
    ProDrawingCurrentSheetSet(drawing, sheet);
/*-----*\
Ask the user to select a solid and add it to the drawing
/*-----*\
    if(!UsrSolidSelect(&solid))
        return(0);
    ProDrawingSolidAdd(drawing, solid);
/*-----*\
Create a general view from the Z axis direction
/*-----*\
    ProUtilMatrixCopy(NULL, matrix);
    pos[0] = 200.0;
    pos[1] = 600.0;
    pos[2] = 0.0;
    ProDrawingGeneralviewCreate(drawing, solid, sheet, PRO_B_FALSE,
                                pos, 0.5, matrix, &view);
/*-----*\
Get the position and size of the new view
/*-----*\
    ProDrawingViewOutlineGet(drawing, view, outline);
/*-----*\
Create a projected view to the right of the general view
/*-----*\
    pos[0] = outline[1][0] + (outline[1][0] - outline[0][0]);
    pos[1] = (outline[0][1] + outline[1][1]) / 2.0;
    pos[2] = 0.0;
    ProDrawingProjectedviewCreate(drawing, view, PRO_B_FALSE, pos,
                                &pview);
/*-----*\
Create a projected view below the general view
/*-----*\
    pos[0] = (outline[0][0] + outline[1][0]) / 2.0;
    pos[1] = outline[0][1] - (outline[1][1] - outline[0][1]);
    ProDrawingProjectedviewCreate(drawing, view, PRO_B_FALSE, pos,
                                &pview);

    return(1);
}

```

Background Views

Functions Introduced:

- **ProDrawingViewsBackground()**
- **ProDrawingBackgroundViewGet()**

Views can be assigned not only to the solid views that the user creates, but also to each sheet. The view assigned to a drawing sheet is called a background view. Function

ProDrawingViewIsBackground() determines whether the specified view is a background view.

If you use **ProSelect()** for an item in a drawing that is not inside a solid view, such as a detail item, the ProView view handle output by **ProSelectionViewGet()** is the background view.

Note: These functions supersede the functions **ProDrawingViewIsOverlay()** and **ProDrawingOverlayviewGet()**, which previously served the same purpose.

Detailed Views

Functions Introduced:

- **ProDrawingViewDetailCreate()**
- **ProDrawingViewDetailReferenceGet()**
- **ProDrawingViewDetailReferenceSet()**
- **ProDrawingViewDetailCurvedataGet()**
- **ProDrawingViewDetailCurvedataSet()**
- **ProDrawingViewDetailBoundaryGet()**
- **ProDrawingViewDetailBoundarySet()**

A detailed view is a small portion of a model shown enlarged in another view.

The function **ProDrawingViewDetailCreate()** creates a detailed view given the reference point on the parent view, the spline curve data, and location of the new view. A note with the detailed view name and the spline curve border are included in the parent view for the created detailed view.

The functions **ProDrawingViewDetailReferenceGet()** and **ProDrawingViewDetailReferenceSet()** return and assign, respectively, the reference point on the parent view for a specified detailed view.

The functions **ProDrawingViewDetailCurvedataGet()** and **ProDrawingViewDetailCurvedataSet()** retrieve and assign, respectively, the spline curve data in terms of the *ProCurvedata* handle for a specified detailed view. Use the function **ProCurvedataAlloc()** to allocate memory for the curve data structure.

The function **ProDrawingViewDetailBoundaryGet()** retrieves the type of spline curve used to define the boundary of the detailed view and also identifies whether the boundary is displayed on the parent view.

The function **ProDrawingViewDetailBoundarySet()** assigns the boundary type for a detailed view in terms of the enumerated type *ProViewDetailBoundaryType*. The types of boundaries are:

- **PRO_DETAIL_BOUNDARY_CIRCLE**—Draws a circle in the parent view.
- **PRO_DETAIL_BOUNDARY_ELLIPSE**—Draws an ellipse in the parent view.
- **PRO_DETAIL_BOUNDARY_HORZ_VER_ELLIPSE**—Draws an ellipse with a horizontal or vertical major axis.
- **PRO_DETAIL_BOUNDARY_SPLINE**—Displays the spline boundary drawn by the user in the parent view.
- **PRO_DETAIL_BOUNDARY_ASME_CIRCLE**—Displays an ASME standard-compliant circle as an arc with arrows and the detailed view name.

This function also sets the *ProBoolean* argument *show* to display the boundary of the detailed view in the parent view.

Auxiliary Views

Functions Introduced:

- **ProDrawingViewAuxiliaryCreate()**
- **ProDrawingViewAuxiliarySet()**
- **ProDrawingViewAuxiliaryInfoGet()**

An auxiliary view is a type of projected view that projects at right angles to a selected surface or axis. The selected surface or axis in the parent view must be perpendicular to the plane of the screen.

The function **ProDrawingViewAuxiliaryCreate()** creates an auxiliary view given the selection reference and the point location.

The function **ProDrawingViewAuxiliarySet()** sets a specified drawing view as the auxiliary view.

The function **ProDrawingViewAuxiliaryInfoGet()** retrieves information such as the selection reference in terms of the *ProSelection* object and the point location in terms of the *ProPoint3d* object for a specified auxiliary view.

Revolved Views

Functions Introduced:

- **ProDrawingViewRevolveCreate()**
- **ProDrawingViewRevolveInfoGet()**

A revolved view is a cross section of a drawing view revolved 90 degrees around a cutting plane projection.

The function **ProDrawingViewRevolveCreate()** creates a revolved view given a cross section, the selection reference, and the point location.

The function **ProDrawingViewRevolveInfoGet()** retrieves information such as the cross section in terms of the *ProXsec* object, the selection reference in terms of the *ProSelection* object, and the point location in terms of the *ProPoint3d* object for a specified revolved view.

View Orientation

Functions Introduced:

- **ProDrawingViewOrientationFromNameSet()**
- **ProDrawingViewOrientationFromReferenceSet()**
- **ProDrawingViewOrientationFromAngleSet()**

The orientation type for a drawing view is given by the enumerated type *ProDrawingViewOrientationType*. The view orientation can be of the following types:

- **PRO_VIEW_ORIENT_NAME**—The drawing view is oriented using saved views from the model.
- **PRO_VIEW_ORIENT_GEOM_REF**—The drawing view is oriented using geometric references from the model.
- **PRO_VIEW_ORIENT_ANGLE**—The drawing view is oriented using angles of selected references or custom angles.

Note: The drawing view must be displayed before applying any orientation to it.

The function **ProDrawingViewOrientationFromNameSet()** assigns the orientation of a specified drawing view according to the following input arguments:

- *mdl_view_name*—Specifies the name of the saved view in the model.
- *orientation_name*—Specifies the name of the user-defined orientation for the saved view.
- *x_angle*—Specifies the X angle in degrees for the user-defined orientation.
- *y_angle*—Specifies the Y angle in degrees for the user-defined orientation.

The function

ProDrawingViewOrientationFromReferenceSet() assigns the orientation of a specified drawing view according to the following input arguments:

- *ref_name_1*—Specifies the name of the first geometric reference.
- *ref_sel_1*—Specifies the first reference selection on the model in terms of the *ProSelection* object.
- *ref_name_2*—Specifies the name of the second geometric reference.
- *ref_sel_2*—Specifies the second reference selection on the model in terms of the *ProSelection* object.

The function **ProDrawingViewOrientationFromAngleSet()** assigns the orientation of a specified drawing view according to the following input arguments:

- *sel*—Specifies the reference selection in terms of the *ProSelection* object. It can be an axis or NULL for other type.
- *angle*—Specifies the angle in degrees with the selected reference.
- *rot_ref_name*—Specifies the name of the rotational angle.
- *index*—Specifies the index of the angle setting.

Visible Areas of Views

Functions Introduced:

- **ProDrawingViewVisibleareaTypeGet()**
- **ProDrawingViewFullVisibleAreaSet()**
- **ProDrawingViewHalfVisibleAreaGet()**
- **ProDrawingViewHalfVisibleAreaSet()**
- **ProDrawingViewPartialVisibleAreaSet()**
- **ProDrawingViewPartialVisibleAreaGet()**
- **ProDrawingViewBrokenVisibleAreaGet()**
- **ProDrawingViewBrokenVisibleAreaSet()**
- **ProDrawingViewBrokenNumberGet()**

As you detail your model, certain portions of the model may be more relevant than others or may be clearer if displayed from a different view point.

The function **ProDrawingViewVisibleareaTypeGet()** retrieves the type of visible area for a specified drawing view in terms of the enumerated type *ProDrawingViewVisibleareaType*. The visible area can be of the following types:

- **PRO_VIEW_FULL_AREA**—The complete drawing view is retained as the visible area.
- **PRO_VIEW_HALF_AREA**—A portion of the model from the view on one side of a cutting plane is removed.
- **PRO_VIEW_PARTIAL_AREA**—A portion of the model in a view within a closed boundary is displayed.
- **PRO_VIEW_BROKEN_AREA**—A portion of the model view from between two or more selected points is removed, and the gap between the remaining two portions is closed within a specified distance.

The function **ProDrawingViewFullVisibleAreaSet()** retains the specified drawing view as the full visible area.

The function **ProDrawingViewHalfVisibleAreaGet()** retrieves the half visible area for a specified drawing view.

The function **ProDrawingViewHalfVisibleAreaSet()** assigns the following arguments to define the half visible area for a specified drawing view:

- *plane_ref*—Specifies the selection reference in terms of the *ProSelection* object that divides the drawing view. The cutting plane can be a planar surface or a datum, but it must be perpendicular to the screen in the new view.
- *keep_side*—Specifies the half side of the model that is to be retained.
- *line_standard*—Specifies the line style used to define the half view in terms of the enumerated type *ProDrawingLineStandardType*. The line style can be of the following types:
 - PRO_HVL_NONE—Specifies no line
 - PRO_HVL_SOLID—Specifies a solid line.
 - PRO_HVL_SYMMETRY—Specifies a symmetry line.
 - PRO_HVL_SYMMETRY_ISO—Specifies an ISO-standard symmetry line.
 - PRO_HVL_SYMMETRY_ASME—Specifies an ASME-standard symmetry line.

The function **ProDrawingViewPartialVisibleAreaGet()** retrieves the partial visible area for a specified drawing view.

The function **ProDrawingViewPartialVisibleAreaSet()** assigns the following arguments to define the partial visible area for a specified drawing view:

- *ref_point*—Specifies the reference selection point on the view in terms of the *ProSelection* object.
- *curve_data*—Specifies the spline curve data in terms of the *ProCurvedata* object.
- *show_boundary*—Specifies a *ProBoolean* value that identifies whether the spline boundary for the partial visible area is displayed. Set this argument to PRO_B_TRUE to display the boundary.

The function **ProDrawingViewBrokenVisibleAreaGet()** retrieves the broken visible area for a specified drawing view.

The function **ProDrawingViewBrokenVisibleAreaSet()** assigns the following arguments to define the broken visible area for a specified drawing view:

- *dir*—Specifies the direction of the broken lines that define the broken area to be removed. The direction is given by the enumerated type *ProViewBrokenDir* and takes the following values:

- PRO_VIEW_BROKEN_DIR_HORIZONTAL—Specifies the horizontal direction.
- PRO_VIEW_BROKEN_DIR_VERTICAL—Specifies the vertical direction.
- *first_sel*—Specifies the selection point in terms of the *ProSelection* object for the first break line.
- *second_sel*—Specifies the selection point in terms of the *ProSelection* object for the second break line.
- *line_style*—Specifies the line style for the broken lines in terms of the enumerated type *ProViewBrokenLineStyle*. It can be one of the following types:
 - PRO_VIEW_BROKEN_LINE_STRAIGHT—Specifies a straight broken line.
 - PRO_VIEW_BROKEN_LINE_SKETCH—Specifies a random sketch drawn by the user that defines the broken line.
 - PRO_VIEW_BROKEN_LINE_S_CURVE_OUTLINE—Specifies an S-curve on the view outline.
 - PRO_VIEW_BROKEN_LINE_S_CURVE_GEOMETRY—Specifies an S-curve on geometry.
 - PRO_VIEW_BROKEN_LINE_HEART_BEAT_OUTLINE—Specifies a heartbeat type of curve on the view outline.
 - PRO_VIEW_BROKEN_LINE_HEART_BEAT_GEOMETRY—Specifies a heartbeat type of curve on the geometry.
- *curve_data*—Specifies the spline curve data in terms of the *ProCurvedata* object when the *line_style* is of the type PRO_VIEW_BROKEN_LINE_SKETCH.

The function **ProDrawingViewBrokenNumberGet()** returns the number of breaks defined for the broken visible area. Two broken lines define one break.

Note: A broken visible area can be created only for general and projected view types.

Sections of a View

Functions Introduced:

- **ProDrawingViewSectionTypeGet()**
- **ProDrawingView2DSectionGet()**
- **ProDrawingView2DSectionSet()**
- **ProDrawingView2DSectionNumberGet()**
- **ProDrawingView3DSectionGet()**
- **ProDrawingView3DSectionSet()**
- **ProDrawingViewSinglepartSectionGet()**
- **ProDrawingViewSinglepartSectionSet()**

The function **ProDrawingViewSectionTypeGet()** retrieves the section type for a specified drawing view in terms of the enumerated type *ProDrawingViewSectionType*. A section can be of the following types:

- **PRO_VIEW_NO_SECTION**—Specifies no section.
- **PRO_VIEW_TOTAL_SECTION**—Specifies the complete drawing view.
- **PRO_VIEW_AREA_SECTION**—Specifies a 2D cross section.
- **PRO_VIEW_3D_SECTION**—Specifies a 3D cross section.
- **PRO_VIEW_PART_SURF_SECTION**—Specifies a section created out of a solid surface or a datum quilt in the model.

The function **ProDrawingView2DSectionGet()** retrieves the 2D cross section for a specified drawing view.

The function **ProDrawingView2DSectionSet()** assigns the following arguments to define the 2D cross section for a drawing view:

- *sec_name*—Specifies the name of the 2D cross section.
- *sec_area_type*—Specifies the type of section area. The section area is given by the enumerated type *ProDrawingViewSectionAreaType* and can be of the following types:
 - **PRO_VIEW_SECTION_AREA_FULL**—Sectioning is applied to full drawing view.

- PRO_VIEW_SECTION_AREA_HALF—Sectioning is applied to half drawing view depending upon the inputs for half side.
- PRO_VIEW_SECTION_AREA_LOCAL—Specifies local sectioning.
- PRO_VIEW_SECTION_AREA_UNFOLD—Unfold the drawing view and section it.
- PRO_VIEW_SECTION_AREA_ALIGNED—Sectioning is as per the aligned views.
- *ref_sel*—Specifies the selection reference in terms of the *ProSelection* object.
- *curve_data*—Specifies the spline curve data in terms of the *ProCurvedata* handle.
- *arrow_display_view*—Specifies the drawing view, that is, either the parent or child view, where the section arrow is to be displayed.

Note: For a section area of type PRO_VIEW_SECTION_AREA_FULL in the above function, you can pass the input arguments *ref_sel*, *curve_data*, and *arrow_display_view* as NULL.

The function **ProDrawingView2DSectionNumberGet()** retrieves the number of 2D cross sections defined for a drawing view.

The function **ProDrawingView3DSectionGet()** retrieves the 3D cross section for a specified drawing view.

The function **ProDrawingView3DSectionSet()** assigns the following arguments to define a 3D cross section for a view:

- *sec_name*—Specifies the name of the 3D cross section.
- *show_x_hatch*—Specifies a *ProBoolean* value that determines whether X-hatching is displayed in the 3D cross-sectional view. Set this argument to PRO_B_TRUE to display X-hatching.

The function **ProDrawingViewSinglepartSectionGet()** retrieves the section created out of a solid surface or a datum quilt in the model for a specified drawing view.

The function **ProDrawingViewSinglepartSectionSet()** assigns the reference selection in terms of the *ProSelection* object for the solid surface or datum quilt that is used to create the section in the view.

View States

Functions Introduced:

- **ProDrawingViewSimplifiedGet()**
- **ProDrawingViewSimplifiedSet()**
- **ProDrawingViewExplodedGet()**
- **ProDrawingViewExplodedSet()**

The functions **ProDrawingViewSimplifiedGet()** and **ProDrawingViewSimplifiedSet()** retrieve and set the simplified representation for a specified drawing view.

The functions **ProDrawingViewExplodedGet()** and **ProDrawingViewExplodedSet()** retrieve and set the exploded state for a specified drawing view.

Example 7: Creating Drawing Views and Accessing their Properties

The following example code creates both general and projection types of drawing views and accesses the view scale, orientation, exploded state, and visible areas of the created drawing views.

```
/*----- Pro/Develop includes -----*/
#include <prodevelop.h>
#include <select3d.h>

/*----- Pro/Toolkit includes -----*/
#include <ProToolkit.h>
#include <ProMdl.h>
#include <ProCsys.h>
#include <ProMenu.h>
#include <ProDrawingView.h>
#include <ProDwgtable.h>
/*----- Application includes -----*/
#include <TestError.h>

/*----- Function Prototypes -----*/
ProError UserSelectViewType (char * selected);
ProError UserSelectExplodeState (int * exploded);
ProError UserPickPosition ();
ProError UserVisibleAreaHalf();
ProError UserVisibleAreaBroken();

/*=====*\
FUNCTION   : UserViewCreate
PURPOSE   : Creates both general and projection type drawing views.
\*=====*/
int UserViewCreate ()
```

```

{
    /* Current Drawing Variables */
    ProMdlp_draw;
    int num_sheets, sheet_num;

    /* Created view variables */
    Pro_draw_view      view_data;
    charview_type[PRO_NAME_SIZE];
    charexpl_state[PRO_NAME_SIZE];
    ProPoint3dposition;
    int view_id;

    /* View Selection Variables  for Projection View Creation*/
    Select3d*view_sel;
    Object_elementsub, *ref;
    intsel_num;

    /* Model Variables for General View Creation */
    ProMdlp_model;
    ProMdlTypetype;
    ProNamenname;

    /* Csys variables for General View Creation */
    ProCsys      p_csys;
    ProGeomitemdata *csys_data;
    ProNamecsys_name;
    double      matrix[4][4];

    /* General use variables */
    int      status;
    int anint;
    int      i_range[2];
    doubled_range[2];
    char      astr[PRO_LINE_SIZE];
    int win_id;

    ProSelection*edge_sel;
    intnum_edge;
    ProPoint3d      location;
    ProView      output_view;
    ProDrawing      drawing;
    ProMouseButton      btn;

    ProFileName WMSGFIL =
        {'m','s','g','_','u','g','d','w','g','.','t','x','t','\0'};
    /*-----*\
        Get the current model
    \*-----*/
    status = ProMdlCurrentGet(&p_draw);
    ERROR_CHECK("UserViewCreate", "ProMdlCurrentGet", status);
    if (status != PRO_TK_NO_ERROR) return(status);

```



```

/*-----*\
    Prompt for the sheet to insert to
/*-----*/
num_sheets = prodrw_number_sheets ( (Prohandle) p_draw);
ERROR_CHECK("UserViewCreate", "prodrw_number_sheets", (num_sheets<=0));
if (num_sheets<=0) return (PRO_TK_GENERAL_ERROR);

i_range[0] = 1;
i_range[1] = num_sheets;
ProTKSprintf (astr, "Enter sheet number [1-%d] [CUR] : ", num_sheets);
status = ProMessageDisplay(WMSGFIL, "USER %0s", astr);
ERROR_CHECK("UserViewCreate", "ProMessageDisplay", status);

status = ProMessageIntegerRead(i_range, &sheet_num);
ERROR_CHECK("UserViewCreate", "ProMessageIntRead", status);
if (status != PRO_TK_NO_ERROR)
{
    sheet_num = prodrw_get_current_sheet ( (Prohandle) p_draw);

ERROR_CHECK("UserViewCreate", "prodrw_get_current_sheet", (sheet_num<=0));
    if (sheet_num<=0) return (PRO_TK_GENERAL_ERROR);
}

/*-----*\
    get the type of view to create (Projection or General)
    (warning : should test if there is a view on this sheet first)
/*-----*/
status = ProMessageDisplay(WMSGFIL, "USER %0s",
"Select the type of view to create ");
ERROR_CHECK("UserViewCreate", "ProMessageDisplay", status);

status = UserSelectViewType(view_type);
ERROR_CHECK("UserViewCreate", "UserSelectViewType", status);
if (status != PRO_TK_NO_ERROR) return(status);

/*-----*\
    For a Projection, do the following
/*-----*/
if (!strcmp(view_type, "-Projection"))
{
    view_data.type = PRODRWVIEW_PROJECTION;

/*-----*\
    Get the view to project
/*-----*/
    status = ProMessageDisplay (WMSGFIL, "USER %0s",
"Select a parent view.");
    ERROR_CHECK ("UserViewCreate", "ProMessageDisplay", status);

```

```

sel_num = pro_select("dwg_view", 1, &view_sel, NULL,NULL);
ERROR_CHECK("UserViewCreate","pro_select", (sel_num<=0));
if (sel_num <0) return (PRO_TK_GENERAL_ERROR);
else if (sel_num =0) return (PRO_TK_USER_ABORT);

sub.type = SEL_DWG_VIEW;
sub.ptr = view_sel[0].view_ptr;
sub.id = -1;
status = pro_element_info ((Prohandle) p_draw, &sub, PRO_IDENTIFY,
-1, &ref);
    ERROR_CHECK("UserViewCreate","pro_element_info", (status <=0));
    if (status <=0) return (PRO_TK_GENERAL_ERROR);

view_data.form.projection.view_id = ref[0].id;

/*-----*\
    Obtain the type of the model in that view
\*-----*/
p_model = (ProMdl) prodrw_get_view_model( (Prohandle) p_draw,
    ref[0].id);
ERROR_CHECK ("UserViewCreate","prodrw_get_view_model",
(p_model == 0));

status = ProMdlTypeGet(p_model, &type);
    ERROR_CHECK ("UserViewCreate","ProMdlTypeGet",status);
    if (status != PRO_TK_NO_ERROR) return(status);

/*-----*\
    If the model is an assembly, explode ?
\*-----*/
if ( type == PRO_MDL_ASSEMBLY)
{
    status
=UserSelectExplodeState(&(view_data.form.projection.exploded));
    if (status != PRO_TK_NO_ERROR) return (status);
}

    else view_data.form.projection.exploded = 0;

/*-----*\
    Get the position where to put the view
\*-----*/
status = UserPickPosition(WMSGFIL, "Select position for view creation",
    view_data.form.projection.pos);
if (status != PRO_TK_NO_ERROR) return (status);
}

/*-----*\
    For a General view, do the following
\*-----*/
    else if (!strcmp(view_type, "-General"))
    {
        view_data.type = PRODRWVIEW_GENERAL;

```

```

        view_data.form.general.sheet_num = sheet_num;

/*-----*\
    Prompt the user for the model to add to the drawing
\*-----*/
    status = UserUtilNameTypeGet(WMSGFIL, name, &type );
    ERROR_CHECK("UserViewCreate", "UserUtilNameTypeGet", status);
    if (status != PRO_TK_NO_ERROR) return(PRO_TK_GENERAL_ERROR);

    status = ProMdlRetrieve(name, type, (&p_model));
    ERROR_CHECK("UserViewCreate", "ProMdlRetrieve", status);
    if (status != PRO_TK_NO_ERROR) return(PRO_TK_E_NOT_FOUND);

/*-----*\
    Add it to the drawing and to the drawing_view_data
\*-----*/
    status = prodrw_add_drawing_model ((Prohandle) p_draw,
                                       (Prohandle) p_model);
    ERROR_CHECK("UserViewCreate", "prodrw_add_drawing_model",
    (status != PRODRW_ADD_MODEL_SUCCESS));
    if ((status != PRODRW_ADD_MODEL_SUCCESS) &&
        (status != PRODRW_ADD_MODEL_EXIST))
        return (PRO_TK_GENERAL_ERROR);

    view_data.form.general.model = (Prohandle) p_model;

/*-----*\
    If the model is an assembly, explode ?
\*-----*/
    if ( type == PRO_MDL_ASSEMBLY)
    {
        status =
UserSelectExplodeState(&(view_data.form.general.exploded));
        ERROR_CHECK("UserViewCreate", "UserSelectExplodeState", status);
        if (status != PRO_TK_NO_ERROR) return (status);
    }
    else view_data.form.general.exploded = 0;

/*-----*\
    Get the position where to put the view
\*-----*/
status = UserPickPosition(WMSGFIL, "Select position for view creation",
    view_data.form.general.pos);
    ERROR_CHECK("UserViewCreate", "UserSelectExplodeState", status);
    if (status != PRO_TK_NO_ERROR) return (status);

/*-----*\
    View Scale
\*-----*/
    status = ProMessageDisplay (WMSGFIL, "USER %0s",
    "Enter view scale factor [CR=default]:");

```

```

ERROR_CHECK("UserViewCreate", "ProMessageDisplay", status);

status = ProMessageDoubleRead(d_range, &view_data.form.general.scale);
ERROR_CHECK("UserViewCreate", "ProMessageDoubleRead", status);
if (status != PRO_TK_NO_ERROR) view_data.form.general.scale = 0.0;

/*-----*\
    Orientation
\*-----*/
ProUtilMatrixCopy (NULL, view_data.form.general.orient);
status = ProMessageDisplay(WMSGFIL, "USER %0s",
    "Enter model csys name to define view orientation [CR=default]:");
ERROR_CHECK("UserViewCreate", "ProMessageDisplay", status);

status = ProMessageStringRead (PRODEV_NAME_SIZE, csys_name);
ERROR_CHECK("UserViewCreate", "ProMessageStringRead", status);

if (status == PRO_TK_NO_ERROR)
{
    status = ProUtilCsysFind((Prohandle) p_model, csys_name, &p_csys);
    ERROR_CHECK("UserViewCreate", "ProUtilCsysFind", status);

    if (status == PRO_TK_NO_ERROR)
    {
        status = ProCsysDataGet (p_csys, &csys_data);
        ERROR_CHECK("UserViewCreate", "ProCsysDataGet", status);

        ProUtilTransfToVectors( matrix,
            csys_data->data.p_csys_data->x_vector,
            csys_data->data.p_csys_data->y_vector,
            csys_data->data.p_csys_data->z_vector,
            csys_data->data.p_csys_data->origin);

        status = ProGeomitemdataFree(&csys_data);
        ERROR_CHECK("UserViewCreate", "ProGeomitemdataFree", status);

        ProUtilMatrixInvert(matrix, view_data.form.general.orient);
    }
}

else
    return (PRO_TK_GENERAL_ERROR);

status = prodrw_create_view (p_draw, &view_data, &view_id);
ERROR_CHECK("UserViewCreate", "prodrw_create_view",
    status != PRODEV_NO_ERROR);

if (status != PRODEV_NO_ERROR)
{
    ProTKSprintf(astr, "Error %d in UserViewCreate", status);

```

```

    status = ProMessageDisplay(WMSGFIL, "USER %0s", astr);
    ERROR_CHECK ("UserViewCreate", "ProMessageDisplay", status);
    return (PRO_TK_GENERAL_ERROR);
}

    strcpy(astr, "View Created Successfully");
    status = ProMessageDisplay(WMSGFIL, "USER %0s", astr);
    ERROR_CHECK ("UserViewCreate", "ProMessageDisplay", status);

/*-----*\
    Update the changes by repainting the window
\*-----*/
    status = ProWindowCurrentGet(&win_id);
    ERROR_CHECK("UserViewCreate", "ProWindowCurrentGet", status);
    if (status != PRO_TK_NO_ERROR) return (status);

    status = ProWindowRepaint (win_id);
    ERROR_CHECK("UserViewCreate", "ProWindowRepaint", status);
    return (status);
}

/*=====*\
HELPER FUNCTIONS
\*=====*/
/*=====*\
FUNCTION : UserSelectViewType
PURPOSE : Select Projection or General
\*=====*/
ProError UserSelectViewType (char * selected)
{
    int i, status;
    ProNamewtitle;
    wchar_t ** buttons;
    wchar_t ** selection;
    int num;

    ProStringToWstring(wtitle, "UG View Type");

    buttons = (wchar_t **) calloc(3, sizeof(wchar_t *));
    for (i=0 ; i<3 ; i++)
    buttons[i] = (wchar_t *) calloc(1, sizeof(ProName));
    ProStringToWstring(buttons[0], "-Projection");
    ProStringToWstring(buttons[1], "-General");
    ProStringToWstring(buttons[2], "");

    status = ProMenuStringsSelect (wtitle, buttons, 1, NULL, &selection,
    &num);
    if (status != PRO_TK_NO_ERROR) return (status);

    ProWstringToString(selected, selection[0]);

```

```

    return (PRO_TK_NO_ERROR);
}

/*=====*\
FUNCTION   : UserSelectExplodeState
PURPOSE    : Select NotExploded or Exploded
\*=====*/
ProError UserSelectExplodeState (int * explode)
{
    int          i, status;
    ProName      wtitle;
    wchar_t      ** buttons;
    wchar_t      ** selection;
    int          num;
    ProNameWMSGFIL;
    char          selected[PRO_NAME_SIZE];

    ProMessageDisplay(WMSGFIL, "USER %0s",
"Select explode state of the view to create ");

    ProStringToWstring(wtitle, "UG Explode");

    buttons = (wchar_t **) calloc(3, sizeof(wchar_t *));
    for (i=0 ; i<3 ; i++)
        buttons[i] = (wchar_t *) calloc(1, sizeof(ProName));
    ProStringToWstring(buttons[0], "-Not Exploded");
    ProStringToWstring(buttons[1], "-Exploded");
    ProStringToWstring(buttons[2], "");

    status = ProMenuStringsSelect (wtitle, buttons, 1, NULL, &selection,
&num);
    if (status != PRO_TK_NO_ERROR) return (status);

    ProWstringToString(selected, selection[0]);

    if (!strcmp(selected, "-Exploded"))
        (*explode) = 1;
    else
        (*explode) = 0;

    return (PRO_TK_NO_ERROR);
}

/*=====*\
FUNCTION   : UserVisibleAreaHalf
PURPOSE    : To set half visible area for view
\*=====*/
ProError UserVisibleAreaHalf ()
{
    ProDrawing drawing;
    ProSelection *view_sel, *plane_sel;

```

```

int num_sel,num_plane;
ProView view;

ProFileName WMSGFIL =
{'m','s','g','_','u','g','d','w','g','.', 't','x','t','\0'};

/*-----*\
    Get the current model
\*-----*/
status = ProMdlCurrentGet ((ProMdl)&drawing);
ERROR_CHECK("UserVisibleAreaHalf","ProMdlCurrentGet",status);
if (status != PRO_TK_NO_ERROR) return(status);

/*-----*\
    Get view to set half visible area
\*-----*/
status = ProMessageDisplay(WMSGFIL, "USER %0s",
"Select view to set visible area");
ERROR_CHECK("UserVisibleAreaHalf","ProMessageDisplay",status);

status = ProSelect ("dwg_view", 1, NULL, NULL, NULL, NULL, &view_sel,
&num_sel);
ERROR_CHECK("UserVisibleAreaHalf","ProSelect", status);
if (status != PRO_TK_NO_ERROR) return(status);

status = ProSelectionViewGet (view_sel[0], &view);
ERROR_CHECK("UserVisibleAreaHalf","ProSelectionViewGet",status);

/*-----*\
    Get plane to set half visible area
\*-----*/
status = ProMessageDisplay (WMSGFIL,"USER %0s",
    "Select plane");
ERROR_CHECK ("UserVisibleAreaHalf","ProMessageDisplay",status);

status = ProSelect ("datum", 1, NULL, NULL, NULL, NULL, &plane_sel,
&num_plane );
ERROR_CHECK("UserVisibleAreaHalf","ProSelect", status);
if (status != PRO_TK_NO_ERROR) return(status);

status = ProDrawingViewHalfVisibleAreaSet (drawing, view, plane_sel[0],
    PRO_B_TRUE, PRO_HVL_SOLID);
ERROR_CHECK("UserVisibleAreaHalf","ProDrawingViewHalfVisibleAreaSet",
status);

status = ProDrawingViewRegenerate (drawing, view);
ERROR_CHECK("UserVisibleAreaHalf","ProDrawingViewRegenerate",status);

return (status);
}

```

```

/*=====*\
FUNCTION   : UserVisibleAreaBroken
PURPOSE    : To set broken visible area for view
\*=====*/
ProError UserVisibleAreaBroken()
{
    ProDrawing drawing;
    ProView view;
    ProSelection *view_sel,*first_sel,*sec_sel;
    int num_first,num_sec,num_sel;

    ProFileName WMSGFIL =
{'m','s','g','_','u','g','d','w','g','.','t','x','t','\0'};

    /*-----*\
        Get the current model
    \*-----*/
    status = ProMdlCurrentGet ((ProMdl)&drawing);
    ERROR_CHECK("UserVisibleAreaBroken","ProMdlCurrentGet",status);
    if (status != PRO_TK_NO_ERROR) return(status);

    /*-----*\
        Get view to set broken visible area
    \*-----*/
    status = ProMessageDisplay(WMSGFIL, "USER %0s",
"Select view to set visible area");
    ERROR_CHECK("UserVisibleAreaBroken","ProMessageDisplay",status);

    status = ProSelect ("dwg_view", 1, NULL, NULL, NULL, NULL, &view_sel,
&num_sel);
    ERROR_CHECK("UserVisibleAreaBroken","ProSelect", status);
    if (status != PRO_TK_NO_ERROR) return(status);

    status = ProSelectionViewGet (view_sel[0], &view);
    ERROR_CHECK("UserVisibleAreaBroken","ProSelectionViewGet",status);

    /*-----*\
        Get break lines
    \*-----*/
    status = ProMessageDisplay (WMSGFIL,"USER %0s",
        "Select first break line");
    ERROR_CHECK ("UserVisibleAreaBroken","ProMessageDisplay",status);

    status = ProSelect ("edge", 1, NULL, NULL, NULL, NULL, &first_sel,
&num_first );
    ERROR_CHECK("UserVisibleAreaBroken","ProSelect", status);
    if (status != PRO_TK_NO_ERROR) return(status);

    status = ProMessageDisplay (WMSGFIL,"USER %0s",
        "Select second break line");
    ERROR_CHECK ("UserVisibleAreaBroken","ProMessageDisplay",status);

```



```

    status = ProSelect ("edge", 1, NULL, NULL, NULL, NULL, &sec_sel,
&num_sec );
    ERROR_CHECK("UserVisibleAreaBroken","ProSelect", status);
    if (status != PRO_TK_NO_ERROR) return(status);

    status = ProDrawingViewBrokenVisibleAreaSet
(drawing,view,PRO_VIEW_BROKEN_DIR_HORIZONTAL,
    first_sel[0], sec_sel[0], PRO_VIEW_BROKEN_LINE_STRAIGHT, NULL);

ERROR_CHECK("UserVisibleAreaBroken","ProDrawingViewBrokenVisibleAreaSet",
status);

    status = ProDrawingViewRegenerate (drawing, view);

ERROR_CHECK("ProDrawingViewRegenerate","ProDrawingViewRegenerate",status)
;

    return (status);
}

```

Drawing Models

Functions Introduced:

- **ProDrawingSolidsCollect()**
- **ProDrawingSolidsVisit()**
- **ProDrawingSolidAdd()**
- **ProDrawingSolidDelete()**
- **ProDrawingSimprepsCollect()**
- **ProDrawingAsmsimprepAdd()**
- **ProDrawingAsmsimprepDelete()**
- **ProDrawingSolidReplace()**
- **ProDrawingCurrentsolidGet()**
- **ProDrawingCurrentsolidSet()**

The function **ProDrawingSolidsCollect()** outputs an array of the solids attached to the drawing, including those not currently displayed in a view. Function **ProDrawingSolidsVisit()** is a visit function of the usual form which visits the same solids.

The function **ProDrawingSolidAdd()** adds a new solid to a drawing, but does not display it. (To create a drawing view, refer to the Creating Views section.)

ProDrawingSolidDelete() deletes a solid from a drawing, provided that solid is not currently displayed in a view.

Functions **ProDrawingSimpregsCollect()**, **ProDrawingAsmsimpregAdd()**, and **ProDrawingAsmsimpregDelete()** are the equivalents to the above functions but take a handle to a simplified rep.

The function **ProDrawingSolidReplace()** replaces a drawing model solid with another solid. The old and new solids must be members of the same family table. The following example code describes this function.

The functions **ProDrawingCurrentsolidGet()** and **ProDrawingCurrentsolidSet()** provide access to the current solid model for a given drawing.

Example 8: Replace Drawing Model Solid with a Solid

```
#include <ProToolkit.h>
#include <ProDrawing.h>
#include <ProFaminstance.h>
#include <PTApplsUnicodeUtils.h>
/*=====*\
FUNCTION: UserDrawingSolidReplace()
PURPOSE:  Replaces all instance solid models in a drawing with their
          generic.  Similar to the Pro/ENGINEER behavior,
          the function will not replace models if the target generic
          model is already present in the drawing.
\*=====*/
int UsrDrawingSolidReplace()
{
    ProError      err;
    ProDrawing    drawing;
    ProMdlType    mdl_type;
    ProSolid*     solid_array;
    int           array_size, i;
    ProSolid      generic;
/*-----*\
    Use the current model to create the drawing.
\*-----*/
    err = ProMdlCurrentGet ((ProMdl*)&drawing);
    if (err != PRO_TK_NO_ERROR)
        return (err);
    err = ProMdlTypeGet ((ProMdl)drawing, &mdl_type);
    if (err != PRO_TK_NO_ERROR ||
        (mdl_type != PRO_MDL_DRAWING))
        return PRO_TK_INVALID_TYPE;
/*-----*\
    Visit the drawing models.
\*-----*/
```

```

err = ProDrawingSolidsCollect (drawing, &solid_array);
if (err != PRO_TK_NO_ERROR)
    return (err);
ProArraySizeGet (solid_array, &array_size);
/*-----*\
Loop on all of the drawing models.
\*-----*/
for (i = 0; i < array_size; i++)
{
/*-----*\
If the generic is not an instance, continue.
\*-----*/
err = ProFamInstanceGenericGet ((ProMdl)solid_array[i], PRO_B_TRUE,
                                (ProMdl*)&generic);

    if (err != PRO_TK_NO_ERROR)
        continue;
/*-----*\
Replace all instances with their (top-level) generic.
\*-----*/
err = ProDrawingSolidReplace (drawing, solid_array[i], generic,
                              PRO_B_TRUE);

    if (err == PRO_TK_E_FOUND)
    {
        ProTKPrintf ("Model was not replaced because");
        ProTKPrintf (" the target model is already in the drawing.\n");
    }
}
return (PRO_TK_NO_ERROR);
}

```

Detail Items

The functions described in this section operate on detail items. Detail items are those drawing items that you create in Pro/ENGINEER using the options under **Drawing>Detail**.

In Pro/TOOLKIT, you have the ability to create, delete, and modify detail items, control their display, and examine what detail items are present in the drawing.

There types of detail items available in Pro/TOOLKIT are:

- *Draft entities*—Contain the graphical items created in Pro/ENGINEER using the options under **Detail>Sketch**. The items are as follows:
 - Arc
 - Ellipse

- Line
- Point
- Polygon
- Spline
- *Notes*—Textual annotations created in Pro/ENGINEER using the command **Detail>Create>Note**. They can also contain special symbols.
- *Symbol definitions*—Named groups of other detail items that the Pro/ENGINEER user can save to disk. You create them in Pro/ENGINEER using the options under **Detail>Create>Symbol Definition**.
- *Symbol instances*—Instances of a symbol.
- *Draft groups*—Groups of detail items such as draft entities, notes, symbol instances, and drawing dimensions. You create them in Pro/ENGINEER using the command **Detail>Tools>Group**.
- OLE objects—Object Linking and Embedding (OLE) objects embedded in the Pro/ENGINEER drawing file from the **Insert Object** dialog box that opens when you click **Insert > Object**.

All detail items are identified by DHandles which are equivalent to ProModelitem, and inherit from ProModelitem. This implies that functions such as **ProSelectionModelitemGet()**, **ProSelectionAlloc()**, and **ProModelitemInit()**, can be used for detail items. The values of the type field for the types of detail item are:

- PRO_DRAFT_ENTITY—This type is used for draft entities and OLE objects. Special functions exist to distinguish OLE objects from other detail entities.
- PRO_NOTE
- PRO_SYMBOL_DEFINITION
- PRO_SYMBOL_INSTANCE
- PRO_DRAFT_GROUP

There is generic detail object called ProDtlitem, whose type field can take any of these values, and is used for arguments to functions that can represent any detail item. The following object handles are used in the more specific cases:

- ProDtlentity
- ProDtlnote

- ProDtlsymdef
- ProDtlsyminst
- ProDtlgroup

Listing Detail Items

Functions Introduced:

- **ProDrawingDtlentitiesCollect()**
- **ProDrawingDtntityVisit()**
- **ProDrawingDtlnotesCollect()**
- **ProDrawingDtlnoteVisit()**
- **ProDrawingDtlsymdefsCollect()**
- **ProDrawingDtlsymdefVisit()**
- **ProDrawingDtlsyminstsCollect()**
- **ProDrawingDtlsyminstVisit()**
- **ProDrawingDtlgroupsCollect()**
- **ProDrawingDtlgroupVisit()**
- **ProDrawingOLEobjectsVisit()**

The function **ProDrawingDtlentitiesCollect()** collects all the detail entities in the specified drawing and sheet. Set the input argument *symbol* to null if you are collecting a detail item in the drawing. If you are collecting a draft entity in a symbol definition, set *symbol* to specify the owning symbol definition.

The function **ProDrawingDtntityVisit()** visits all the draft entities stored in the specified drawing and sheet.

The function **ProDrawingDtlnotesCollect()** collects all the notes in the specified drawing. Set the input argument *symbol* to null if you are collecting a note from the drawing. If you are collecting a note from a symbol definition, set *symbol* to specify the owning symbol definition.

The function **ProDrawingDtlnoteVisit()** visits the notes in the specified drawing.

The function **ProDrawingDtlsymdefsCollect()** collects the symbol definitions in the specified drawing.

The function **ProDrawingDtlsymdefVisit()** visits the symbol definitions in the drawing.

The function **ProDrawingDtlsyminstsCollect()** collects symbol instances in the specified drawing.

The function **ProDrawingDtlsyminstVisit()** visits symbol instances in the specified drawing.

The function **ProDrawingDtlgroupsCollect()** collects groups in the specified drawing.

The function **ProDrawingDtlgroupVisit()** visits groups in the specified drawing.

The function **ProDrawingOLEobjectsVisit()** visits the OLE objects embedded in the model. Specify the visit action and visit filter functions.

Displaying Detail Items

Functions Introduced:

- **ProDtlententityDraw()**
- **ProDtlententityErase()**
- **ProDtlnoteDraw()**
- **ProDtlnoteErase()**
- **ProDtlnoteShow()**
- **ProDtlnoteRemove()**
- **ProDtlsyminstDraw()**
- **ProDtlsyminstErase()**
- **ProDtlsyminstShow()**
- **ProDtlsyminstRemove()**
- **ProDtlgroupDraw()**
- **ProDtlgroupErase()**

Each of the displayable item types has four display functions.

The Show function displays the detail item, such that it is repainted on the next draft regeneration.

The Remove function undraws the detail item permanently, so that it is not redrawn on the next draft regeneration.

The Draw function draws the detail item temporarily, so that it is removed on the next draft regeneration.

The Erase function undraws the detail item temporarily, so that it is redrawn on the next draft regeneration, if it was previously “shown”.

Use the Show function after creating an item, and the Remove function before deleting it. Use the Erase function before modifying an item, and the Draw function afterwards.

Note: These functions require that the drawing must be the currently displayed model. To create or modify detail items in a model that is not currently displayed, use the attributes in the data structures related to Display. For example use **ProDtlnotedataDisplayedSet()** to set the item to be saved with the displayed status turned on, so that the next retrieval of the model will display the item.

Creating, Modifying and Reading Detail Items

Functions Introduced:

- **ProDtlentityCreate()**
- **ProDtlentityDataGet()**
- **ProDtlentityDelete()**
- **ProDtlentityModify()**
- **ProDtlentityErase()**
- **ProDtlentityIsOLEObject()**
- **ProDtlnoteCreate()**
- **ProDtlnoteDataGet()**
- **ProDtlnoteDelete()**
- **ProDtlnoteLineEnvelopeGet()**
- **ProDtlnoteModify()**
- **ProDtlsymdefCreate()**
- **ProDtlsymdefDataGet()**
- **ProDtlsymdefDelete()**
- **ProDtlsymdefModify()**
- **ProDtlsymdefToModelCopy()**
- **ProDtlsyminstCreate()**
- **ProDtlsyminstDataGet()**

- **ProDtlsyminstDelete()**
- **ProDtlsyminstModify()**
- **ProDtlgroupCreate()**
- **ProDtlgroupDataGet()**
- **ProDtlgroupDelete()**
- **ProDtlgroupModify()**

For each of the five detail item types there is an opaque data structure which describes the contents of the detail item. You build the appropriate data structure first, using functions provided for that purpose, and then pass it as input to the appropriate **Create()** function. The ***DataGet()** functions output a filled structure describing an existing detail item. The data structures are built and unpacked by their own functions for that purpose described in the following sections.

The functions **ProDtlnoteDataGet()** and **ProDtlsyminstDataGet()** have an argument for the display mode. Both notes and symbols may contain parameterized text, and the display mode specifies whether the data structure output by the ***DataGet()** function must contain the text before substitution of the parameters (SYMBOLIC mode), or after the displayed text (NUMERIC mode). If using **ProDtlnoteDataGet()** as a first step in note modification, always set “mode” to SYMBOLIC or the modification removes the parameterization. Refer to section Detail Note Line Data for more information.

Some data structures contain arrays of, or pointers to, deeper structures which have their own manipulation functions, also described in later sections. Lower level data structures should be built before the upper level ones when creating detail items. The data structures and their member structures are listed below.

- **ProDtlentitydata**—A draft entity
- **ProCurvedata**—The 2d geometry of the entity (described in the ‘Core: 3D Geometry’ chapter)
- **ProDtlnotedata**—A detail note
- **ProDtlnoteline**—A line of text in a note
- **ProDtlnotetext**—A segment of text in a line that may have it's own cosmetic properties, such as font, height, and so on
- **ProDtlattach**—One structure for the attachment of the note itself, and one per leader on the note
- **ProDtlsymdefdata**—A symbol definition

- **ProDtlSymDefAttach**—The types of attachment support for an instance of this symbol
- **ProDtlSymInstData**—A symbol instance
- **ProDtlVarText**—A variable text substitution

The sequence of calls to create a draft entity containing, for example, a line would be:

1. **ProDtlEntityDataAlloc()**— Allocate the entity data (see the section on Draft Entity Data).
2. **ProCurveDataAlloc()**—Allocate memory for a curve structure
3. **ProLineDataInit()**—Set the curve structure to describe the required line by initializing a line data structure.
4. **ProDtlEntityDataCurveSet()**—Add the curve data to the entity data (see the section on Draft Entity Data).
5. **ProDtlEntityCreate()**—Create the entity (see the section on Creating, Modifying and Reading Detail Items).

Note: You must set the drawing view before attempting to create a detail entity, unless you are creating entities in a symbol definition. The view can be a traditional drawing view obtained through **ProDrawingViewsCollect()**, or the drawing sheet background view obtained from **ProDrawingBackgroundViewGet()**.

The other detail items follow the same principles, although for symbol definitions there are added complexities; these are explained in section Creating a Symbol Definition.

The function **ProDtlEntityIsOLEObject()** identifies if the specified detail entity is actually an OLE object. For more information on OLE objects refer to Accessing OLE Objects.

Each entity item has its own **Delete()** function which removes it permanently from the Pro/ENGINEER drawing.

Each entity item also has its own **Modify()** function which passes a new Data structure.

Use the function **ProDtlNoteErase()** to temporarily undraw the note (see the section on Displaying Detail Items).

The function **ProDtlNoteDataGet()** returns the data for the note.

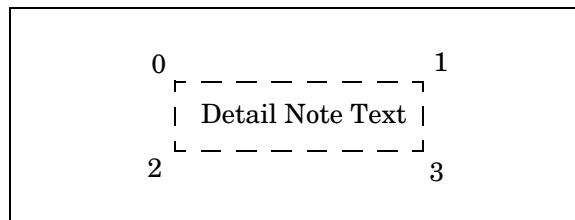
The function **ProDtlNoteDataColorSet()** modifies the color in the data (see the section on Detail Note Data).

The function **ProDtlnoteModify()** uses the modified data to modify the note itself.

The function **ProDtlnoteDraw()** redraws the note (see the section on Displaying Detail Items).

The function **ProDtlnoteLineEnvelopeGet()** determines the screen coordinates of the envelope around a detail note. This envelope is defined by four points. See figure Detail Note Envelope Point Order for how point order is determined.

Figure 51-1: Detail Note Envelope Point Order



The function **ProDtlsymdefToModelCopy()** copies a specified symbol definition from one model to another.

Draft Entity Data

Functions Introduced:

- **ProDtlentitydataAlloc()**
- **ProDtlentitydataFree()**
- **ProDtlentitydataIdGet()**
- **ProDtlentitydataCurveGet()**
- **ProDtlentitydataCurveSet()**
- **ProDtlentitydataColorGet()**
- **ProDtlentitydataColorSet()**
- **ProDtlentitydataFontGet()**
- **ProDtlentitydataFontSet()**
- **ProDtlentitydataWidthGet()**
- **ProDtlentitydataWidthSet()**
- **ProDtlentitydataViewGet()**
- **ProDtlentitydataViewSet()**

- **ProDtentitydataIsConstruction()**
- **ProDtentitydataConstructionSet()**

The opaque data structure which describes the contents of a draft entity is called *ProDtentitydata*.

The only lower-level opaque data structure contained by *ProEntitydata* is *ProCurvedata* which is also used for other 2d and 3d geometry, especially Import Features, and is described elsewhere.

The functions **ProDtentitydataAlloc()** and **ProDtentitydataFree()** allocate and free an opaque entity data structure.

Functions **ProDtentitydataCurveGet()** and **ProDtentitydataCurveSet()** get and set the geometry of the entity in the form of a *ProCurvdata* object.

ProDtentitydataColorGet() and **ProDtentitydataColorSet()** get and set the color of the draft entity. The visible data structure *ProColor*, declared in *ProDtlitem.h*, can specify the color in three ways:

- By color index, that is, by choosing one of colors predefined in *Pro/E*, represented by the values of *ProColortype* in *ProToolkit.h*
- By choosing the default color for this type of detail item. (For entities, the default is *PRO_COLOR_DRAWING* and for notes the default is *PRO_COLOR_LETTER*.)
- By specifying the three RGB color values.

If do you do not call **ProDtentitydataColorSet()** when creating a new entity, the color will be set to be the default color for draft entities.

ProDtentitydataFontGet() and **ProDtentitydataFontSet()** get and set the line style font which determines the line style used to display the entity. The values are those which appear in the Line Font selector in the Modify Line Style dialog in *Pro/ENGINEER*. If you do not call **ProDtentitydataFontSet()** when creating an entity, the font will be "SOLIDFONT".

ProDtentitydataWidthGet() and

ProDtentitydataWidthSet() get and set the line width of the draft entity. The value is interpreted in the same way as the value entered into the Width text box in Pro/ENGINEER's Modify Line Style dialog. The value -1.0 indicates that the entity should have the default width for entities currently set for the drawing. If you do not call **ProDtentitydataWidthSet()** when creating a new entity, the width is -1.0.

ProDtentitydataViewGet() and **ProDtentitydataViewSet()** get and set the drawing view to which the entity will be attached. If an entity is attached to a view, it moves whenever the Pro/ENGINEER user moves that view. Entities not attached to a model view must be assigned to the drawing sheet background view instead.

ProDtentitydataIsConstruction() and **ProDtentitydataConstructionSet()** get and set the flag that controls whether the entity is created normal or as a construction entity.

Example 9: Create a Draft Line with Predefined Color

The following example shows a utility that creates a draft line in one of the colors predefined in Pro/ENGINEER.

```
#include <ProToolkit.h>
#include <ProDtentity.h>

*=====*\
FUNCTION : UsrLineentityCreate()
PURPOSE  : Utility to create a line entity between specified points
\*=====*/
int UsrLineentityCreate(
    ProDrawing drawing,
    ProVector start,
    ProVector end,
    ProColor type color)
{
    ProDtentitydata edata;
    ProCurvedata *curve;
    ProDtentity entity;
    ProColor entity_color;
    int cur_sheet;
    ProView view;

    /*-----*\
        Allocate data for the draft entity
    \*-----*/
    ProDtentitydataAlloc(drawing, &edata);
```

```

/*-----*\
    Allocate and initialize a curve description
\*-----*/
    ProCurvedataAlloc(&curve);
    ProLinedataInit(start, end, curve);
/*-----*\
    Use the curve description to set the entity data curve
\*-----*/
    ProDtlentitydataCurveSet(edata, curve);
/*-----*\
    Set the view to the current background view for this sheet
\*-----*/
    ProDrawingCurrentSheetGet (drawing, &cur_sheet);

    ProDrawingBackgroundViewGet (drawing, cur_sheet, &view);

    ProDtlentitydataViewSet (edata, view);

/*-----*\
    Set the color to the specified Pro/ENGINEER predefined color
\*-----*/
    entity_color.method = PRO_COLOR_METHOD_TYPE;
    entity_color.value.type = color;
    ProDtlentitydataColorSet(edata, &entity_color);

/*-----*\
    Create the entity
\*-----*/
    ProDtlentityCreate(drawing, NULL, edata, &entity);

/*-----*\
    Display the entity
\*-----*/
    ProWindowRepaint (PRO_VALUE_UNUSED);
/*-----*\
    Release the entity data memory
\*-----*/
    ProDtlentitydataFree(edata);
}

int UsrLineentityCreateWrapper ()
{
    ProDrawing drawing;
    ProMouseButton button;
    ProVector start, end;
    ProFileName msgfil;

    ProStringToWstring (msgfil, "msg_ugdrawing.txt");

    ProMdlCurrentGet ((ProMdl*)&drawing);

```

```

ProMessageDisplay(msgfil,"USER Select ends of line");

if(ProMousePickGet(PRO_ANY_BUTTON, &button, start) != PRO_TK_NO_ERROR)
    return(0);
if(ProMousePickGet(PRO_ANY_BUTTON, &button, end) != PRO_TK_NO_ERROR)
    return(0);

UsrLineentityCreate(drawing, start, end, PRO_COLOR_WARNING);

return PRO_TK_NO_ERROR;
}

```

Accessing OLE Objects

An object linking and embedding (OLE) object is an external file, such as a document, graphics file, or video file that is created using an external application and which can be inserted into another application, such as Pro/ENGINEER. You can create and insert supported OLE objects into a two-dimensional Pro/ENGINEER file, such as a drawing, report, format file, layout, or diagram. The functions described in this section enable you to identify and access OLE objects embedded in drawings.

Function Description

- **ProDrawingOLEObjectSheetGet()**
- **ProDrawingOLEObjectOutlineGet()**
- **ProDrawingOLEObjectApplicationtypeGet()**
- **ProDrawingOLEObjectPathGet()**

The function **ProDrawingOLEObjectSheetGet()** returns the sheet number for the OLE object.

The function **ProDrawingOLEObjectOutlineGet()** returns the extent of the OLE object embedded in the drawing.

The function **ProDrawingOLEObjectApplicationtypeGet()** returns the type of the OLE object as a string, for example, "Microsoft Word Document".

The function **ProDrawingOLEObjectPathGet()** returns the path to the external file for each OLE object, if it is linked to an external file.

Detail Note Text Data

Functions Introduced:

- **ProDtlnotetextAlloc()**
- **ProDtlnotetextFree()**
- **ProDtlnotetextHeightGet()**
- **ProDtlnotetextHeightSet()**
- **ProDtlnotetextWidthGet()**
- **ProDtlnotetextWidthSet()**
- **ProDtlnotetextSlantGet()**
- **ProDtlnotetextSlantSet()**
- **ProDtlnotetextThicknessGet()**
- **ProDtlnotetextThicknessSet()**
- **ProDtlnotetextFontGet()**
- **ProDtlnotetextFontSet()**
- **ProDtlnotetextUlineGet()**
- **ProDtlnotetextUlineSet()**
- **ProDtlnotetextStringGet()**
- **ProDtlnotetextStringSet()**

Each line of a drawing note may contain text in several different fonts, heights, and so on. So each line is described in terms of an array of text items, whose contents are described by the data structure **ProDtlnotetext**.

ProDtlnotetextAlloc() and **ProDtlnotetextFree()** allocate and free a **ProDtlnotetext** data structure.

ProDtlnotetextHeightGet() and **ProDtlnotetextHeightSet()** get and set the height of the text. The value -1.0 means that the text has the default height for text currently specified for the drawing.

ProDtlnotetextWidthGet() and **ProDtlnotetextWidthSet()** get and set the width factor of the text. The width factor is the ratio of the width of each character to the height. The value -1.0 means that the width factor has the default value for text currently specified for the drawing.

ProDtlnotetextSlantGet() and **ProDtlnotetextSlantSet()** get and set the slant angle of the text.

ProDtlnotetextThicknessGet() and **ProDtlnotetextThicknessSet()** get and set the line thickness of the text. The value -1.0 means that the text has the default thickness for text currently specified for the drawing.

ProDtlnotetextFontGet() and **ProDtlnotetextFontSet()** get and set the font used to display the text. The fonts are those available in the “font” selector in the Text Style dialog in Pro/ENGINEER.

ProDtlnotetextUlineGet() and **ProDtlnotetextUlineSet()** get and set whether the text item is underlined. The default is no underline.

ProDtlnotetextStringGet() and **ProDtlnotetextStringSet()** get and set the string of characters contained in the text item.

Detail Note Line Data

Functions Introduced:

- **ProDtlnotelineAlloc()**
- **ProDtlnotelineFree()**
- **ProDtlnotelineTextAdd()**
- **ProDtlnotelineTextsSet()**
- **ProDtlnotelineTextsCollect()**

The **ProDtlnoteline** data structure describes the contents of a single line of text in a detail note.

ProDtlnotelineAlloc() and **ProDtlnotelineFree()** allocate and free a **ProDtlnoteline** structure.

ProDtlnotelineTextAdd() adds a text item, described by a **ProDtlnotetext** data structure, to a note line. If the line already contained text items, the new one is added at the end of the array.

ProDtlnotelineTextsSet() sets the contents of a whole text line, by providing a new array of **ProDtlnotetext** items. If the note line already contained text items, they are replaced by the new ones.

ProDtlnotelineTextsCollect() outputs an array of the text items contained in a specified text line.

Points to note about Text Lines and parameterization:

- If the string in a Text Line you put in a note contains one or more parameters, Pro/ENGINEER will divide the Text Line into several Text Items to ensure that each parameter has its own Text Item.
- When you look at the text in an existing note by using the function **ProDtlnoteDataGet()** with the “mode” option set to SYMBOLIC (that is, to see the text before substitution of the parameters), you will see the text bracketing and text item identifiers that you also see when you edit a text line in Pro/ENGINEER.

For example, if you make a text line containing a single text item with the text

```
"model = &model_name"
```

Pro/ENGINEER will put the &model_name into a separate text item when the note is created. If you then use **ProDtlnoteDataGet()** on the created note with the “mode” option set to SYMBOLIC, you will see the following two text items in the relevant text line

```
"model = " "&model_name"
```

If you set “mode” to NUMERIC, you see these text items:

```
"model = " "MODEL"
```

where MODEL is the name of the model.

Note: Pro/ENGINEER does not resolve and replace symbolic callouts for notes which are not displayed. Therefore, if the note is not displayed or is hidden in a layer, the text retrieved may contain symbolic callouts, even when the “mode” is set to NUMERIC.

Note that **ProDtlnotetextStringGet()** does not return the brackets and numbers for each individual text entity. In addition, the function does not return the special escape characters (such as "\\}") to represent characters previously provided.

Refer to the section Creating a Symbol Definition for a description of how to find which Pro/ENGINEER model owns the parameter referred to by parameterized text.

Detail Attachments and Leaders

Functions Introduced:

- **ProDtlattachAlloc()**
- **ProDtlattachGet()**
- **ProDtlattachFree()**
- **ProDtlattachArrowtypeGet()**

The opaque data structure *ProDtlattach* is used for two tasks:

- The way in which a drawing note or a symbol instance is attached to the drawing.
- The way in which a leader on a drawing note or symbol instance is attached.

Each note and symbol instance must contain one *ProDtlattach* to describe its attachment in the drawing, and may contain any number of *ProDtlattach* objects describing the leaders.

ProDtlattachAlloc() allocates and initializes the memory for a detail attachment. The inputs are:

- *type*—The type of attachment to the drawing view. The detail attachment types are as follows:
 - **FREE**—The attachment is to a 2D location in the drawing view.
 - **PARAMETRIC**—The attachment is to a point on a surface or an edge of a solid.
 - **OFFSET**—The attachment is offset to another drawing view, to a model item, or to a 3D model annotation.
- *view*—The drawing view. If the type is **FREE**, the attachment is relative to the drawing view, that is the attachment moves when the drawing view is moved. This is **NULL**, if the detail attachment is not related to the drawing view, but is placed at a specified location in the drawing sheet, or if the attachment is offset to a model item or to a 3D model annotation.

- *location*—If the type is FREE or OFFSET, this argument provides the location of the attachment. This location is in screen coordinates for drawing items, symbol instances and surface finishes on flat-to-screen annotation planes, and in model coordinates for symbols and surface finishes on 3D model annotation planes. The distance from this location to the location of the item to which the detail item is attached (given by the argument *attach_point*) is saved as the offset distance for an OFFSET attachment.
- *attach_point*—If the type is PARAMETRIC or OFFSET, this *ProSelection* structure provides the location of the item to which the detail item is attached. This includes the drawing view in which the attachment is made. If you are building this structure using **ProSelectionAlloc()**, set the location using **ProSelectionUvParamSet()**, and the drawing view using **ProSelectionViewSet()**.

ProDtlattachGet() unpacks the above information for an existing attachment. The output arguments are:

- *type*—The type of attachment to the drawing view. The detail attachment types are as follows:
 - FREE—The attachment is to a 2D location in the drawing view.
 - PARAMETRIC—The attachment is to a point on a surface or an edge of a solid in a drawing view.
 - OFFSET—The attachment is offset to another drawing view, to a model item, or to a 3D model annotation.
 - UNIMPLEMENTED—The attachment is to an item that is not currently supported in Pro/TOOLKIT. However, you can still retrieve the location and the view to which the attachment is connected.
- *view*—If the type is FREE or UNIMPLEMENTED, this argument specifies the drawing view. This is NULL, if the detail attachment is not related to the drawing view, but is placed at a specified location in the drawing sheet, or if the attachment is offset to a model item or to a 3D model annotation.

- *location*—If the type is FREE, OFFSET, or UNIMPLEMENTED, this argument specifies the location of the attachment. This location is in screen coordinates for drawing items, symbol instances and surface finishes on flat-to-screen annotation planes, and in model coordinates for symbols and surface finishes on 3D model annotation planes. The distance from this location to the location of the item to which the detail item is attached (given by the argument *attach_point*) is saved as the offset distance for an OFFSET attachment.
- *attach_point* —If the type is PARAMETRIC or OFFSET, this argument provides the location of the item to which the detail item is attached. This includes the drawing view in which the attachment is made.

ProDtlattachFree() frees an attachment that was allocated with **ProDtlattachAlloc()**.

The function **ProDtlattachArrowtypeGet()** extracts the type of arrowhead used for the leaders attached to a drawing note or symbol instance. This function only works for *ProDtlattach* leader attachment objects obtained using the functions **ProDtlsyminstdataLeadersCollect()** and **ProDtlnotedataLeadersCollect()**.

Detail Note Data

Functions Introduced:

- **ProDtlnotedataAlloc()**
- **ProDtlnotedataFree()**
- **ProDtlnotedataIdGet()**
- **ProDtlnotedataLineAdd()**
- **ProDtlnotedataLinesSet()**
- **ProDtlnotedataLinesCollect()**
- **ProDtlnotedataMirrorSet()**
- **ProDtlnotedataIsMirrored()**
- **ProDtlnotedataColorGet()**
- **ProDtlnotedataColorSet()**
- **ProDtlnotedataAttachmentGet()**
- **ProDtlnotedataAttachmentSet()**
- **ProDtlnotedataLeadersCollect()**

- **ProDtlnotedataLeadersSet()**
- **ProDtlnotedataLeaderAdd()**
- **ProDtlnotedataElbowlengthGet()**
- **ProDtlnotedataElbowlengthSet()**
- **ProDtlnotedataAngleGet()**
- **ProDtlnotedataAngleSet()**
- **ProDtlnotedataJustifGet()**
- **ProDtlnotedataJustifSet()**
- **ProDtlnotedataIsDisplayed()**
- **ProDtlnotedataDisplayedSet()**
- **ProDtlnoteDtIsyminstsCollect()**

The object **ProDtlnotedata** is an opaque pointer to a data structure that describes the contents of a drawing note.

ProDtlnotedataAlloc() and **ProDtlnotedataFree()** allocate and free memory for the data.

ProDtlnotedataIdGet() gives you the integer id of the note in Pro/ENGINEER that the data describes. This will be set if the **ProDtlnotedata** has been acquired using **ProDtlnoteDataGet()**. It is not necessary to set this when creating a note; the function **ProDtlnoteCreate()** will assign an id to the new note.

ProDtlnotedataLineAdd() adds a **ProDtlnoteline** object to a **ProDtlnotedata** description. If the note already contains lines of text, the new line will be added at the end.

ProDtlnotedataLinesSet() sets an array of **ProDtlnoteline** objects as the lines in a **ProDtlnotedata** description. If the note already contains text lines, they will be replaced by the new lines.

ProDtlnotedataLinesCollect() outputs an array of **ProDtlnoteline** objects describing the lines in a given **ProDtlnotedata** description.

ProDtlnotedataMirrorSet() sets the **ProDtlnotedata** to specify the note as mirrored, or un-mirrored.

ProDtlnotedataIsMirrored() outputs the current choice.

ProDtlnotedataColorGet() and **ProDtlnotedataColorSet()** get and set the color for the note described by a ProDtlnotedata description. If you do not call **ProDtlnotedataColorSet()** when creating a note, the note will have the default color (defined by PRO_COLOR_LETTER). Refer to the Draft Entity Data section for a fuller description of the ProColor object.

ProDtlnotedataAttachmentGet() and **ProDtlnotedataAttachmentSet()** get and set the ProDtlattach object which describes the attachment of the note, that is, where and how it is positioned on the drawing.

ProDtlnotedataLeadersCollect() outputs an array of ProDtlattach objects which described the attachment points of the leaders on the note. **ProDtlnotedataLeadersSet()** adds an array of leaders to a note, replacing existing leaders.

ProDtlnotedataLeaderAdd() adds a new leader to the end of the array of current leaders on a note.

ProDtlnotedataElbowlengthGet() and **ProDtlnotedataElbowlengthSet()** get and set the length of the elbow that connects each leader to the note. If you do not call **ProDtlnotedataElbowlengthSet()** when creating a note, there will be no elbow.

ProDtlnotedataAngleGet() and **ProDtlnotedataAngleSet()** set the angle of rotation of the note. If you do not call **ProDtlnotedataAngleSet()** when creating the note, the rotation defaults to 0.0.

ProDtlnotedataJustifGet() and **ProDtlnotedataJustifSet()** retrieve and set the note vertical and horizontal justification. Vertical justification applies only to notes in drawing tables and **On Item** notes.

ProDtlnotedataIsDisplayed() and **ProDtlnotedataDisplayedSet()** retrieve and set the flag that controls whether the note is visible or not. If the note is created with this flag set to true, regenerate the drawing using **ProDwgDraftRegenerate()** to see the displayed note.

ProDtlnoteDtlSymInstsCollect() returns a list of all the symbol instances that are declared in a detail note via the “sym()” callout format. The symbol instances are returned in the order they are encountered in the note text.

Example 10: Create Drawing Note at Specified Location with Leader to Surface and Surface Name

The following example shows a function which creates a drawing note at a specified location, with a leader attached to a solid surface, and which shows the name of the surface.

```
#include <ProToolkit.h>
#include <ProDtlnote.h>
#include <PTApplsUnicodeUtils.h>
/*=====*\
FUNCTION : UstrSurfNoteCreate()
PURPOSE  : Utility to create a note that documents the surface name or id.
\*=====*/
int UstrSurfNoteCreate(
    ProDrawing drawing,
    ProSelection surf_sel,      /* The surface - where the note leader
                                should be attached. */
    ProVector pos)             /* The location of the note itself */
{
    ProDtlnotetext text;
    ProName font;
    ProDtlnoteline line;
    ProModelitem modelitem;
    ProName wname;
    ProCharName name;
    ProLine wstr;
    ProCharLine str;
    ProDtlnotedata ndata;
    ProView view;
    ProDtllattach attach, leader;
    ProDtlnote note;
/*-----*\
Allocate a text item, and set its properties
\*-----*/
    ProDtlnotetextAlloc(&text);
    ProDtlnotetextHeightSet(text, -1.0);
    ProDtlnotetextWidthSet(text, -1.0);
    ProDtlnotetextSlantSet(text, 0.0);
    ProDtlnotetextThicknessSet(text, 0.0);
    ProStringToWstring(font, "font");
    ProDtlnotetextFontSet(text, font);
/*-----*\
Get the name of the selected surface, form a string which describes it,
and add the string to the text item
\*-----*/
    ProSelectionModelitemGet(surf_sel, &modelitem);
    if(ProModelitemNameGet(&modelitem, wname) == PRO_TK_NO_ERROR)
        ProWstringToString(name, wname);
    else
        ProTKSprintf(name, "id %d", modelitem.id);
```

```

    ProTKSprintf(str, "Surface %s", name);
    ProStringToWstring(wstr, str);
    ProDtlnotetextStringSet(text, wstr);
/*-----*\
Allocate a new text line, and add the text item to it
\*-----*/
    ProDtlnotelineAlloc(&line);
    ProDtlnotelineTextAdd(line, text);
/*-----*\
Allocate a note description, and add the line to it
\*-----*/
    ProDtlnotedataAlloc(drawing, &ndata);
    ProDtlnotedataLineAdd(ndata, line);
/*-----*\
Set the attachment of the note itself
\*-----*/
    ProSelectionViewGet(surf_sel, &view);
    ProDtlattachAlloc(PRO_DTLATTACHTYPE_FREE, view, pos, NULL, &attach);
    ProDtlnotedataAttachmentSet(ndata, attach);
/*-----*\
Add the note leader
\*-----*/
    ProDtlattachAlloc(PRO_DTLATTACHTYPE_PARAMETRIC, NULL, NULL, surf_sel,
                      &leader);
    ProDtlnotedataLeaderAdd(ndata, leader);
/*-----*\
Create the note
\*-----*/
    ProDtlnoteCreate(drawing, NULL, ndata, &note);
/*-----*\
Display the note
\*-----*/
    ProDtlnoteShow(&note);
/*-----*\
Free the memory for the note description
\*-----*/
    ProDtlnotedataFree(ndata);
}

int UsrSurfNoteCreateWrapper ()
{
    ProDrawing drawing;
    ProMouseButton button;
    ProSelection *sel;
    int n_sel;
    ProVector pos;
    ProFileName msgfil;
    ProStringToWstring (msgfil, "msg_ugdrawing.txt");
    ProMdlCurrentGet((ProMdl*)&drawing);
    ProMessageDisplay(msgfil, "USER Select surface");
    if(ProSelect("surface", 1, NULL, NULL, NULL, NULL, &sel, &n_sel) !=

```



```

        PRO_TK_NO_ERROR || n_sel < 1)
return(0);
ProMessageDisplay(msgfil,"USER Select note position");
if(ProMousePickGet(PRO_ANY_BUTTON, &button, pos) != PRO_TK_NO_ERROR)
return(0);
UsrSurfNoteCreate(drawing, sel[0], pos);
return (1);
}

```

Read-Only Notes

Functions Introduced:

- **ProDtlnotedataReadonlySet()**
- **ProDtlnotedataReadonlyGet()**
- **ProDrawingReadonlyselectionAllow()**

You can make an existing drawing note unselectable by Pro/ENGINEER users if you wish to protect it from modification. The functions **ProDtlnotedataReadonlySet()** and **ProDtlnotedataReadonlyGet()** set and get this property on ProDtlnotedata objects. Use function **ProDtlnotedataReadonlySet()** in conjunction with **ProDtlnoteDataGet()** and **ProDtlnoteModify()** to change the setting.

The function **ProDrawingReadonlyselectionAllow()** will temporarily allow the selection of read-only notes.

Parameterized Note Text

Function introduced:

- **ProDtlnoteModelrefGet()**

A note in a drawing, or in a symbol definition, can be parameterized. This means that it contains the name of a parameter from a Pro/ENGINEER model, preceded by an '&'. The '&' and the parameter name are replaced by the value of the parameter when the note is displayed, or when the symbol is instantiated.

The parameterizations in different notes and symbols in a single drawing may refer to parameters on different Pro/ENGINEER models, depending upon the history of the drawing. The function **ProDtlnoteModelrefGet()** allows you to find out which model is referred to by a specific parameter.

Cross-referencing Gtols and Drawing Annotations

The functions described in this section provide the drawing object that represents a shown gtol (if the gtol is shown in the drawing), or vice-versa.

Functions Introduced:

- **ProGtolDtlnoteGet()**
- **ProDtlnoteGtolGet()**

The function **ProGtolDtlnoteGet()** returns the detail note that represents a shown geometric tolerance.

Note: This function returns the first detail note that calls out the geometric tolerance. Pro/ENGINEER does not restrict users to showing only a single version of a geometric tolerance callout.

The function **ProDtlnoteGtolGet()** returns the geometric tolerance shown in a detail note, if applicable.

Cross-referencing 3D Notes and Drawing Annotations

The functions described in this section provide access to the drawing object that represents a shown 3D note, (if the 3D note is shown in the drawing), or vice-versa.

Functions Introduced:

- **ProNoteDtlnoteGet()**
- **ProDtlnoteNoteGet()**

The function **ProNoteDtlnoteGet()** returns a detail note that represents a shown model tree.

Note: This function returns the first detail note that calls out the solid model note. Pro/ENGINEER does not restrict users to showing only a single version of a solid model note callout.

The function **ProDtlnoteNoteGet()** returns the solid model note that is displayed as a detail note, if applicable.

Symbol Definition Attachments

Functions Introduced:

- **ProDtlsymdefattachAlloc()**
- **ProDtlsymdefattachGet()**
- **ProDtlsymdefattachFree()**
- **ProDtlsymdefdataAttachAdd()**
- **ProDtlsymdefdataAttachSet()**
- **ProDtlsymdefdataAttachGet()**

A symbol definition has several different ways in which instances of that symbol can be attached to the drawing. In Pro/ENGINEER users set these attachments from the General tab on the Symbol Definition Attributes dialog. Each attachment type is described in Pro/TOOLKIT by an opaque data structure called **ProDtlsymdefattach**. This is allocated and filled by the function **ProDtlsymdefattachAlloc()**. The types of attachment are:

- **FREE**—The symbol will have no leaders, and will be attached by a specified location.
- **ON_ITEM**—The symbol will be attached to an entity in the drawing.
- **NORM_ITEM**—The symbol will be attached to an entity, and be rotated to be normal to that entity.
- **LEFT_LEADER**—The attachment is by a leader to a point on an entity at the left of the symbol.
- **RIGHT_LEADER**—The attachment is by a leader to a point on an entity at the right of the symbol.
- **RADIAL_LEADER**—The attachment is by a leader attached to a circular entity in the symbol.

The input arguments to the function are these

- **type**—The type of attachment
- **entity_id**—The id of the entity in the symbol definition which has the attachment point, if the attachment type is *_LEADER. **entity_parameter** The “t” value of the location on the entity which forms the attachment point, if the attachment type is *_LEADER.
- **position**—The location in the symbol coordinate system which forms the attachment point, if the attachment type is FREE, ON_ITEM, or NORM_ITEM.

Symbol Definition Data

Functions Introduced:

- **ProDtlsymdefdataAlloc()**
- **ProDtlsymdefdataFree()**
- **ProDtlsymdefdataIdGet()**
- **ProDtlsymdefdataHeighttypeGet()**
- **ProDtlsymdefdataHeighttypeSet()**
- **ProDtlsymdefdataTextrefSet()**
- **ProDtlsymdefdataTextrefGet()**
- **ProDtlsymdefdataElbowGet()**
- **ProDtlsymdefdataElbowSet()**
- **ProDtlsymdefdataTextangfixedGet()**
- **ProDtlsymdefdataTextangfixedSet()**
- **ProDtlsymdefdataScaledheightGet()**
- **ProDtlsymdefdataPathSet()**
- **ProDtlsymdefdataPathGet()**
- **ProDtlsymdefdataNameGet()**

The opaque object *ProDtlsymdefdata* describes the contents of a symbol definition. The functions **ProDtlsymdefdataAlloc()** and **ProDtlsymdefdataFree()** allocate and free this data.

ProDtlsymdefdataIdGet() gives you the integer id of the symbol definition in Pro/ENGINEER that the data describes. This will be set if the *ProDtlsymdefdata* has been acquired using **ProDtlsymdefDataGet()**. It is not necessary to set this when creating a symbol definition; the function **ProDtlsymdefCreate()** will assign an id to the new note.

ProDtlsymdefdataHeighttypeGet() and **ProDtlsymdefdataHeighttypeSet()** get and set the way in which the size of an instance of this symbol definition is set. The three types are:

- **FIXED**—The symbol instance height is fixed.
- **VARIABLE**—The symbol instance height may be modified by the Pro/ENGINEER user.
- **TEXTRELATED**—The symbol instance height is related to the height of a text item in the definition.

If the height type is TEXTRELATED the functions

ProDtlsymdefdataTextrefSet() and

ProDtlsymdefdataTextrefGet() set and get the text item in the symbol definition which determines the symbol instance height.

The reference is by note id, line index, and text item index.

ProDtlsymdefdataElbowGet() and

ProDtlsymdefdataElbowSet() get and set the bit flag representing the elbow of the symbol definition.

ProDtlsymdefdataTextangfixedGet() and

ProDtlsymdefdataTextangfixedSet() get and set whether the angle of text in the symbol is fixed.

ProDtlsymdefdataScaledheightGet() returns the height of the symbol definition in inches.

ProDtlsymdefdataPathSet() and

ProDtlsymdefdataPathGet() set and get the path and file name of the file in which the symbol definition may be saved. This is used to give the symbol its name.

ProDtlsymdefdataNameGet() gets the name of the symbol definition.

Creating a Symbol Definition

The notes and draft entities that are contained by a symbol definition are created using **ProDtlentityCreate()** and **ProDtlnoteCreate()**, using the ProDtlsymdef handle as the “symbol” argument. So you need to create the empty symbol definition first, and then add the notes and entities.

If you want to add parametric leader attachments, using **ProDtlsymdefdataAttachAdd()** and so on, these identify the entities to which the leaders should attach using the object handles output by the calls to **ProDtlnoteCreate()** and **ProDtlentityCreate()** that created them. So these attachment types should also be added after the symbol is created.

So the steps in creating a symbol definition are:

- Allocate a description—**ProDtlsymdefdataAlloc()**
- Add a FREE attachment—**ProDtlsymdefattachAlloc()**,
ProDtlsymdefdataAttachAdd()
- Create the symbol—**ProDtlsymdefCreate()**
- Add the notes and entities (as for creating notes and entities in the drawing)
- Add any leader attachments—**ProDtlsymdefattachAlloc()**,
ProDtlsymdefdataAttachAdd()

Example 11: Create Symbol Definition

The following example shows a function which creates a symbol definition which contains four line entities forming a box, a note at the middle of the box, and a free attachment.

```
/*=====*\
FUNCTION : UsrBoxSymdefCreate()
PURPOSE  : To create a symbol definition with the specified name,
           containing a box and a note with the specified text
\*=====*/
int UsrBoxSymdefCreate(
    ProDrawing drawing,
    ProCharName name,
    ProCharLine text)
{
    ProDtlsymdefdata sdata;
    ProPath path;
    ProVector origin, e1, e2;
    ProDtlsymdefattach attach;
    ProDtlsymdef symdef;
    double defheight;
    ProName w_size;
    ProMatrix matrix;

/*-----*\
    Allocate symbol definition description data
\*-----*/
ProDtlsymdefdataAlloc(drawing, &sdata);
/*-----*\
    Set the name of the symbol
\*-----*/
ProStringToWstring(path, name);
    ProDtlsymdefdataPathSet(sdata, path);
/*-----*\
    Set the height type
\*-----*/
ProDtlsymdefdataHeighttypeSet(sdata, PRODTLSYMDEFHGHTTYPE_FIXED);
/*-----*\
    Set a FREE attachment at the origin of the symbol
\*-----*/
memset(origin, '\0', sizeof(ProVector));
    ProDtlsymdefattachAlloc(PROSYMDEFATTACHTYPE_FREE, 0, 0.0,
        origin, &attach);
    ProDtlsymdefdataAttachAdd(sdata, attach);
    ProDtlsymdefattachFree(attach);
/*-----*\
    Create the empty symbol, and free the description
\*-----*/
if(ProDtlsymdefCreate(drawing, sdata, &symdef) != PRO_TK_NO_ERROR)
    return(0);
    ProDtlsymdefdataFree(sdata);
```

```

/*-----*\
Calculate the default text height for the drawing
\*-----*/
pro_get_drawing_text_height((Prohandle)drawing, &defheight, NULL);
prodb_drawing_sheet_info((Prohandle)drawing,w_size,matrix);
defheight /= matrix[0][0];
/*-----*\
Create four lines to form a box, twice the default text height,
around the origin
\*-----*/
e1[0] = -defheight;
e1[1] = -defheight;
e1[2] = 0.0;
e2[0] = defheight;
e2[1] = -defheight;
e2[2] = 0.0;
UsrSymdefLineAdd(&symdef, e1, e2, PRO_COLOR_ERROR);
memcpy(e1, e2, sizeof(ProVector));
e2[1]=defheight;
UsrSymdefLineAdd(&symdef, e1, e2, PRO_COLOR_ERROR);
memcpy(e1, e2, sizeof(ProVector));
e2[0]=-defheight;
UsrSymdefLineAdd(&symdef, e1, e2, PRO_COLOR_ERROR);
memcpy(e1, e2, sizeof(ProVector));
e2[1]=-defheight;
UsrSymdefLineAdd(&symdef, e1, e2, PRO_COLOR_ERROR);
/*-----*\
Add a note with the specified text at the origin
\*-----*/
origin[1] -= defheight / 2.0;
UsrSymdefNoteAdd(&symdef, text, origin);
return(1);
}
/*=====*\
FUNCTION : UsrSymdefNoteAdd()
PURPOSE : To add a note with the specified text and location to a
symbol definition.
\*=====*/
int UsrSymdefNoteAdd(
ProDtlsymdef *symdef,
ProCharLine ntext,
ProVector pos)
{
ProDtlnotetext text;
ProName font;
ProDtlnoteline line;
ProModelitem modelitem;
ProName wname;
ProCharName name;
ProLine wstr;
ProCharLine str;

```

```

        ProDtlnotedata ndata;
        ProView view;
        ProDtlattach attach, leader;
        ProDtlnote note;
/*-----*\
    Allocate a text item, and set its properties
\*-----*/
    ProDtlnotetextAlloc(&text);
    ProDtlnotetextHeightSet(text, -1.0);
    ProDtlnotetextWidthSet(text, -1.0);
    ProDtlnotetextSlantSet(text, 0.0);
    ProDtlnotetextThicknessSet(text, 0.0);
    ProStringToWstring(font, "font");
    ProDtlnotetextFontSet(text, font);
    ProStringToWstring(wstr, ntext);
    ProDtlnotetextStringSet(text, wstr);
/*-----*\
    Allocate a new text line, and add the text item to it
\*-----*/
    ProDtlnotelineAlloc(&line);
    ProDtlnotelineTextAdd(line, text);
/*-----*\
    Allocate a note description, and add the line to it
\*-----*/
    ProDtlnotedataAlloc(symdef->owner, &ndata);
    ProDtlnotedataLineAdd(ndata, line);
    ProDtlnotedataJustifSet(ndata, PRO_HORZJUST_CENTER,
                            PRO_VERTJUST_MIDDLE);
/*-----*\
    Set the attachment of the note itself
\*-----*/
ProDtlattachAlloc(PRO_DTLATTACHTYPE_FREE, NULL, pos, NULL, &attach);
ProDtlnotedataAttachmentSet(ndata, attach);
/*-----*\
    Create the note
\*-----*/
ProDtlnoteCreate(symdef->owner, symdef, ndata, &note);
/*-----*\
    Free the memory for the note description
\*-----*/
ProDtlnotedataFree(ndata);
    return(1);
}
/*=====*\
FUNCTION : UstrSymdefLineAdd() PURPOSE :
Utility to add a line entity to a symbol definition
\*=====*/
int UstrSymdefLineAdd(
    ProDtlsymdef *symdef,
    ProVector start,
    ProVector end,

```



```

        ProColortype color)
    {
        ProDtlenity entity;
        ProDtlenitydata edata;
        ProCurvedata *curve;
        ProColor entity_color;
/*-----*\
    Allocate data for the draft entity
\*-----*/
ProDtlenitydataAlloc(symdef->owner, &edata);
/*-----*\
    Allocate and initialize a curve description
\*-----*/
ProCurvedataAlloc(&curve);
ProLinedataInit(start, end, curve);
/*-----*\
    Use the curve description to set the entity data curve
\*-----*/
ProDtlenitydataCurveSet(edata, curve);
/*-----*\
    Set the color to the specified Pro/ENGINEER predefined color
\*-----*/
entity_color.method = PRO_COLOR_METHOD_TYPE;
entity_color.value.type = color;
ProDtlenitydataColorSet(edata, &entity_color);
/*-----*\
    Create the entity
\*-----*/
ProDtlenityCreate(symdef->owner, symdef, edata, &entity);
/*-----*\
    Release the entity data memory
\*-----*/
ProDtlenitydataFree(edata);
        return(1);
    }

```

Retrieving a Symbol Definition from Disk

Function introduced:

- **ProDrawingDtIsymdefRetrieve()**
- **ProDrawingSystemDtIsymdefRetrieve()**

Pro/ENGINEER symbols exist in two different areas: the user-defined area and the system symbols area.

The function **ProDrawingDtlsymdefRetrieve()** enables you to retrieve a symbol definition from the user-defined location designated by the configuration option "pro_symbol_dir". The symbol definition should have been previously saved to a file using Pro/ENGINEER.

The function **ProDrawingSystemDtlsymdefRetrieve()** retrieves a symbol definition from the system directory. The system area contains symbols provided by Pro/ENGINEER with the Pro/DETAIL module (such as the Welding Symbols Library).

Symbol Instance Variable Text

Functions Introduced:

- **ProDtlvartextAlloc()**
- **ProDtlvartextFree()**
- **ProDtlvartextDataGet()**

A symbol instance may replace any text inside a note in the symbol definition that is enclosed in back slash characters. The opaque data structure ProDtlvartext describes such a substitution. It describes the “prompt” string, that is, the string in the symbol definition which it is replacing, and the “value”, that is, the new text string.

The function **ProDtlvartextAlloc()** allocates and initializes a ProDtlvartext object. **ProDtlvartextFree()** frees the memory, and **ProDtlvartextDataGet()** unpacks the information in a ProDtlvartext.

Symbol Instance Data

Functions Introduced:

- **ProDtlsyminstdataAlloc()**
- **ProDtlsyminstdataFree()**
- **ProDtlsyminstdataColorSet()**
- **ProDtlsyminstdataColorGet()**
- **ProDtlsyminstdataDefSet()**
- **ProDtlsyminstdataDefGet()**
- **ProDtlsyminstdataAttachtypeSet()**
- **ProDtlsyminstdataAttachtypeGet()**
- **ProDtlsyminstdataDefattachSet()**

- **ProDtlsyminstdataDefattachGet()**
- **ProDtlsyminstdataAttachmentGet()**
- **ProDtlsyminstdataAttachmentSet()**
- **ProDtlsyminstdataLeadersCollect()**
- **ProDtlsyminstdataLeadersSet()**
- **ProDtlsyminstdataLeaderAdd()**
- **ProDtlsyminstdataElbowlengthGet()**
- **ProDtlsyminstdataElbowlengthSet()**
- **ProDtlsyminstdataAngleSet()**
- **ProDtlsyminstdataAngleGet()**
- **ProDtlsyminstdataScaledheightSet()**
- **ProDtlsyminstdataScaledheightGet()**
- **ProDtlsymInstnoteDataGet()**
- **ProDtlsymInstentityDataGet()**
- **ProDtlsyminstdataDisplayedSet()**
- **ProDtlsyminstdataIsDisplayed()**
- **ProDtlsyminstdataVartextAdd()**
- **ProDtlsyminstdataVartextsSet()**
- **ProDtlsyminstdataVartextsCollect()**
- **ProDtlsyminstdataTransformGet()**
- **ProDtlsyminstdataGroupoptionsSet()**

ProDtlsyminstdataAlloc() and **ProDtlsyminstdataFree()** allocate and free a ProDtlsyminstdata description.

ProDtlnotedataColorGet() and **ProDtlnotedataColorSet()** get and set the color for the symbol instance described by a ProDtlsyminst description. If you do not call **ProDtlnotedataColorGet()** when creating a symbol instance, the color will have the default color which is PRO_COLOR_UNDEFINED, and the symbol will show the colors defined for the entities in the definition. Refer to the section on Draft Entity Data for a fuller description of the *ProColor* object.

ProDtlsyminstdataDefSet() and **ProDtlsyminstdataDefGet()** set and get the reference to the symbol definition that this instance instantiates.

ProDtlsyminstdataAttachtypeSet() and **ProDtlsyminstdataAttachtypeGet()** set and get the type of attachment being chosen for the symbol instance. The corresponding attachment types much exist in the symbol definition.

If you want to make an attachment to a symbol instance of a type that was not specified in the symbol definition, you can add you own symbol definition attachment to the symbol instances.

ProDtlsyminstdataDefattachSet() and **ProDtlsyminstdataDefattachGet()** set and get a **ProDtlsymdefattach** object on a symbol instance with this purpose.

ProDtlsyminstdataAttachmentGet() and **ProDtlsyminstdataAttachmentSet()** get and set the actual attachment for the symbol instance, that is, where it is positioned on the drawing, in the form of a *ProDtlattach* object. Refer to the section on Detail Attachments and Leaders for more information about this object.

ProDtlsyminstdataLeaderAdd() adds a leader to a symbol instance description. **ProDtlsyminstdataLeadersSet()** sets an array of leaders in a symbol instance, replacing any existing leaders. **ProDtlsyminstdataLeadersCollect()** outputs an array of the leaders on a *ProDtlsyminstdata* description.

ProDtlsyminstdataElbowlengthGet() and **ProDtlsyminstdataElbowlengthSet()** get and set the length of the elbow that connects each leader to the symbol instance. If you do not call **ProDtlnotedataElbowlengthSet()** when creating a symbol instance, there will be no elbow.

ProDtlsyminstdataAngleSet() and **ProDtlsyminstdataAngleGet()** get and set the rotation angle of the symbol, if the symbol definition allows rotation. (See also the function **ProDtlsymdefdataTextangfixedSet()** in the section on Symbol Definition Data.)

ProDtlsyminstdataScaledheightSet() and **ProDtlsyminstdataScaledheightGet()** assign and return the height of a symbol instance in the units of the owner drawing or model, respectively. This value is consistent with the height value shown for a symbol instance in the Properties dialog box in the Pro/ENGINEER User Interface.

Note: The scaled height obtained using the above functions is partially based on the properties of the symbol definition assigned using the function **ProDtlsyminstdataDefSet()**. Changing the symbol definition may change the calculated value for the scaled height.

ProDtlsymInstnoteDataGet() and **ProDtlsymInstentityDataGet()** retrieve the data of a note and an entity, respectively, in the symbol instance.

ProDtlsyminstdataDisplayedSet() and **ProDtlsyminstdataIsDisplayed()** set and get the flag that controls whether or not the instance is visible.

ProDtlsyminstdataVartextAdd(), **ProDtlsyminstdataVartextsSet()**, **ProDtlsyminstdataVartextsCollect()** manipulate *ProDtlsvartext* objects in the symbol instance, which provide for substitution of text in the symbol definition. See section Symbol Instance Variable Text for more information about the *ProDtlsvartext* object.

The function **ProDtlsyminstdataTransformGet()** provides a matrix that describes the transformation between symbol definition coordinates and screen coordinates for this instances, that is, it describes the location and orientation of the symbol. It is read-only.

The function **ProDtlsyminstdataGroupoptionsSet()** sets the option for displaying groups in the symbol instance. The possible options are:

- Interactive—prompt the user to select the groups to activate
- All—activate all groups
- None—do not activate any group
- Custom—activate only those groups included in the array of *ProDtlsymgroup* handles passed to this function.

See the section Drawing Symbol Groups to learn more about accessing groups during symbol placement.

Cross-referencing Weld Symbols and Drawing Annotations

The functions described in this section provide a drawing object that represents a shown weld symbol (if the weld symbol is shown in the drawing), or the weld feature that owns a shown weld symbol.

Functions Introduced:

- **ProFeatureDtlsyminstGet()**
- **ProDtlsyminstFeatureGet()**

The function **ProFeatureDtlsyminstGet()** returns the detail symbol instance that represents a shown model symbol.

The function **ProDtlsyminstFeatureGet()** returns the weld feature that owns the shown weld symbol.

Example 12: Create Free Instance of Symbol Definition

The following example shows a function which creates a free instance of a symbol definition.

```
/*=====*\
FUNCTION : UstrSymInstCreate()
PURPOSE  : Create a symbol with no leaders at a specified location
\*=====*/
int UstrSymInstCreate(
    ProDrawing drawing,
    ProDtlsymdef *definition,
    ProVector pos
)
{
    ProDtlsyminstdata sdata;
    ProDtlattach attach;
    ProDtlsyminst syminst;
/*-----*\
    Allocate the symbol instance description
\*-----*/
    ProDtlsyminstdataAlloc(drawing, &sdata);
/*-----*\
    Set the definition this is an instance of
\*-----*/
    ProDtlsyminstdataDefSet(sdata, definition);
/*-----*\
    Set the attachment type to FREE
\*-----*/
    ProDtlsyminstdataAttachtypeSet(sdata, PROSYMDEFATTACHTYPE_FREE);
/*-----*\
    Allocate a FREE attachment and set the position
\*-----*/
    ProDtlattachAlloc(PRO_DTLATTACHTYPE_FREE, NULL, pos, NULL, &attach);
    ProDtlsyminstdataAttachmentSet(sdata, attach);
    ProDtlattachFree(attach);
/*-----*\
    Create the symbol and free the description
\*-----*/
    ProDtlsyminstCreate(drawing, sdata, &syminst);
```

```

    ProDtlSymInstDataFree(sdata);
/*-----*\
Display the symbol
\-----*/
    ProDtlSymInstShow(&syminst);
}

```

Detail Group Data

Functions Introduced:

- **ProDtlGroupDataAlloc()**
- **ProDtlGroupDataFree()**
- **ProDtlGroupDataIdGet()**
- **ProDtlGroupDataNameGet()**
- **ProDtlGroupDataIsDisplayed()**
- **ProDtlGroupDataDisplayedSet()**
- **ProDtlGroupDataItemAdd()**
- **ProDtlGroupDataItemsSet()**
- **ProDtlGroupDataItemsCollect()**

ProDtlGroupDataAlloc() and **ProDtlGroupDataFree()** allocate and free a detail group structure in the form of a **ProDtlGroup** object. **ProDtlGroupDataAlloc()** also sets the name of the group.

ProDtlGroupDataIdGet() returns the internal ID of an existing group. **ProDtlGroupDataNameGet()** gets the name of the group.

ProDtlGroupDataDisplayedSet() and **ProDtlGroupDataIsDisplayed()** set and get the flag that controls whether or not the group is visible.

ProDtlGroupDataItemAdd() adds an item to the group contents. Items supported in the groups include entities, notes, symbol instances, and draft drawing dimensions.

ProDtlGroupDataItemsSet() sets the array of items into a group structure, replacing any existing items that may have been assigned.

ProDtlGroupDataItemsCollect() returns an array of the items in the group structure.

Example 13: Create New Group of Items

The following example shows a command which creates a group from a set of selected detail items.

```
#include <ProToolkit.h>
#include <ProDtlgroup.h>
/*=====*\
FUNCTION : UsgroupCreate()
PURPOSE  : Command to create a new group with selected items
\*=====*/
int UsgroupCreate()
{
    ProError status;
    ProSelection *sel;
    int n_sel, s;
    ProModelitem *items;
    ProDrawing drawing;
    ProName name;
    ProDtlgroupdata gdata;
    ProDtlgroup group;
    ProFileName msgfil;
/*-----*\
    Select notes, draft entities, symbol instances
\*-----*/
    ProStringToWstring (msgfil, "msg_ugdrawing.txt");
    ProMessageDisplay(msgfil,"USER Select the items to belong to the
                        group");
    status = ProSelect("any_note,draft_ent,dtl_symbol",-1,
                      NULL,NULL,NULL,NULL,&sel,&n_sel);
    if(status != PRO_TK_NO_ERROR || n_sel < 1)
        return(0);
/*-----*\
    Allocate and fill a ProArray with the detail item handles
\*-----*/
    ProArrayAlloc(n_sel, sizeof(ProModelitem), 1, (ProArray*)&items);
    for(s=0;s<n_sel;s++)
    {
        status = ProSelectionModelitemGet(sel[s], &items[s]);
        ProTKPrintf ("  **** SEL MDLITEM GET: %d *****\n", status);
        ProTKPrintf ("  *** MDLITEM %d TYPE: %d %%%%\n", items[s].id,
                      items[s].type);
    }
/*-----*\
    Get the current drawing
\*-----*/
    ProMdlCurrentGet((ProMdl*)&drawing);
/*-----*\
    Prompt for the group name
\*-----*/
    ProMessageDisplay(msgfil,"USER Enter group name");
```



```

    if(ProMessageStringRead(PRO_NAME_SIZE, name) != PRO_TK_NO_ERROR)
        return(0);
    ProTKPrintf ("About to create\n");
/*-----*\
Allocate group data
\*-----*/
    status = ProDtlgroupdataAlloc(drawing, name, &gdata);
    ProTKPrintf (" **** DTLGROUP DATA ALLOC: %d *****\n", status);
/*-----*\
Set the group items
\*-----*/
    status = ProDtlgroupdataItemsSet(gdata, items);
    ProTKPrintf (" **** DTLGROUP DATA SET: %d *****\n", status);
/*-----*\
Create the group
\*-----*/
    status = ProDtlgroupCreate(drawing, gdata, &group);
    ProTKPrintf (" **** DTLGROUP CREATE: %d *****\n", status);
/*-----*\
Free the data
\*-----*/
    ProDtlgroupdataFree(gdata);
    ProArrayFree((ProArray*)&items);
}

```

Drawing Symbol Groups

This section describes Pro/TOOLKIT functions that give access to user-defined groups contained in drawing symbols.

User-defined groups in symbol definitions are represented by the following handle in Pro/TOOLKIT:

```

typedef struct pro_dtlvargroup
{
    ProDtlsymdef symbol_def;
    int var_group_id;
} ProDtlsymgroup;

```

The group handle contains the definition handle and an identifier that is unique in the group definition.

Pro/ENGINEER allows a hierarchal relationship between the groups in a symbol definition. Thus, some groups contain groups, or are parents of other groups. To transmit the “level” in which a group resides to Pro/TOOLKIT functions, pass the ProDtlsymgroup handle of the parent group. To look at the groups at the top level, pass a ProDtlsymgroup handle with an identifier of -1.

Identifying Symbol Groups in an Instance

Function introduced:

- **ProDtIsymInstSymgroupsCollect()**

The function **ProDtIsymInstSymgroupsCollect()** indicates which groups are included in the symbol instance. You can collect the groups based on their status:

- All—retrieve all groups in the definition of the symbol instance
- Active—retrieve only those groups which are actively shown in this symbol instance
- Inactive—retrieve only those groups which are not shown in this symbol instance

Identifying Symbol Groups in a Definition

Functions Introduced:

- **ProDtIsymgroupSubgroupsCollect()**
- **ProDtIsymgroupDataGet()**
- **ProDtIsymgroupdataNameGet()**
- **ProDtIsymgroupdataItemsCollect()**
- **ProDtIsymgroupParentGet()**
- **ProDtIsymgroupLevellsExclusive()**

The function **ProDtIsymgroupSubgroupsCollect()** returns the names of all subgroups stored in the symbol definition at the indicated level.

Use the function **ProDtIsymgroupDataGet()** to get the data for the group stored in the symbol definition.

The function **ProDtIsymgroupdataNameGet()** returns the name of the group using the symbol group data as input, while the function **ProDtIsymgroupdataItemsCollect()** returns the names of all the group members using symbol data as input. Note that all these group members are entities or notes contained only within the symbol definition.

The function **ProDtIsymgroupParentGet()** returns the parent group to which the current group belongs.

The function **ProDtlsymgroupLevelIsExclusive()** indicates if the subgroups stored in the symbol definition at the current level are exclusive or independent. If groups are exclusive, only one of the groups at this level may be active in the model at any time. If groups are independent, any number of groups may be active.

Manipulating Symbol Groups

Functions Introduced:

- **ProDtlsymgroupdataAlloc()**
- **ProDtlsymgroupdataNameSet()**
- **ProDtlsymgroupdataItemsSet()**
- **ProDtlsymgroupdataItemAdd()**
- **ProDtlsymgroupdataFree()**
- **ProDtlsymgroupSubgroupCreate()**
- **ProDtlsymgroupModify()**
- **ProDtlsymgroupDelete()**
- **ProDtlsymgroupLevelExclusiveSet()**

The opaque handle **ProDtlsymgroupdata** contains the information needed to define or redefine a group.

The function **ProDtlsymgroupdataAlloc()** allocates the data structure.

The function **ProDtlsymgroupdataNameSet()** sets the name of the symbol group while the function **ProDtlsymgroupdataItemsSet()** sets the specified items to be contained in the symbol group, provided such items belong to the symbol definition. The items to be included can be detail entities and notes.

The function **ProDtlsymgroupdataItemAdd()** adds a single item to the symbol group, provided such an item belongs to the symbol definition. The item to be added can be a detail entity or a note.

The function **ProDtlsymgroupdataFree()** frees the data structure.

The function **ProDtlsymgroupSubgroupCreate()** creates a new group in the symbol definition at the specified level below the parent group.

The function **ProDtlsymgroupModify()** modifies the symbol group definition.

The function **ProDtlsymgroupDelete()** deletes a group from the symbol definition. This function does not delete the entities contained in the group.

The function **ProDtlsymgroupLevelExclusiveSet()** makes the groups at the indicated level, exclusive or independent, in the symbol definition.

Drawing Edges

The functions described in this section provide access to the display properties such as color, line font, and thickness of model edges in drawing views. Model edges can be regular edges, silhouette edges, or non-analytical silhouette edges.

The opaque handle *ProDrawingEdgeDisplay* provides access to the display properties of model edges.

Note: You can select model edges from detailed views for modification, but no change will be applied. To modify the display of a model edge in a detailed view, you must select the edge in the parent view.

Functions Introduced:

- **ProDrawingEdgeDisplayGet()**
- **ProDrawingEdgeDisplaySet()**
- **ProDrawingedgedisplayFree()**
- **ProDrawingedgedisplayColorGet()**
- **ProDrawingedgedisplayColorSet()**
- **ProDrawingedgedisplayFontGet()**
- **ProDrawingedgedisplayFontSet()**
- **ProDrawingedgedisplayWidthGet()**
- **ProDrawingedgedisplayWidthSet()**
- **ProDrawingedgedisplayIsGlobal()**
- **ProDrawingedgedisplayGlobalSet()**

The function **ProDrawingEdgeDisplayGet()** obtains the display properties of a specified model edge in a drawing view. It allocates the *ProDrawingEdgeDisplay* object for storing the display properties.

The function **ProDrawingEdgeDisplaySet()** assigns the display properties of a specified model edge in a drawing view. After assigning the properties, you must repaint the drawing view to update the display.

The function **ProDrawingedgedisplayFree()** frees the memory allocated for the *ProDrawingEdgeDisplay* object.

The functions **ProDrawingedgedisplayColorGet()** and **ProDrawingedgedisplayColorSet()** obtain and assign the color to be used for the display of a specified model edge.

The functions **ProDrawingedgedisplayFontGet()** and **ProDrawingedgedisplayFontSet()** obtain and assign the line font to be used for the display of a specified model edge.

The functions **ProDrawingedgedisplayWidthGet()** and **ProDrawingedgedisplayWidthSet()** obtain and assign the width to be used for the display of a specified model edge. You must pass a value less than zero to use the default width.

Note: The width obtained is in screen coordinates. To convert the width value into drawing coordinates, use the sheet transformation matrix obtained using **ProDrawingSheetTrfGet()**.

The function **ProDrawingedgedisplayIsGlobal()** determines if the model edge display properties such as color, line font, and width have been applied globally to all the drawing views in the drawing sheet.

The function **ProDrawingedgedisplayGlobalSet()** sets the flag that assigns the model edge display properties such as color, line font, and width globally to all the drawing views in the drawing sheet.

Drawing Tables

A drawing table is identified by the DHandle *ProDwgtable* which is typedef to and inherited from *ProModelitem*. The “type” field in *ProDwgtable* has the value PRO_DRAW_TABLE.

Selecting Drawing Tables and Cells

Function Introduced:

- **ProSelectionDwgtblCellGet()**

In order to ask the user to select a table cell, use the option “dwg_table” as input to **ProSelect()**, and then use **ProSelectionModelitemGet()** to acquire the *ProDwgtable* handle to the table.

To select a table cell, use the option “table_cell” and call **ProSelectionModelitemGet()** to get the table handle, and the special function **ProSelectionDwgtblcellGet()** that returns the IDs of the selected table segment, column and row. The function **ProSelectionDwgtblcellGet()** returns row and column values starting from 0. To get the actual values of the rows and the columns, add 1 to the result, so that these can be used in other Pro/TOOLKIT functions.

Creating Drawing Tables

Functions Introduced:

- **ProDwgtabledataAlloc()**
- **ProDwgtabledataOriginSet()**
- **ProDwgtabledataSizetypeSet()**
- **ProDwgtabledataColumnsSet()**
- **ProDwgtabledataRowsSet()**
- **ProDrawingTableCreate()**
- **ProDwgtableTextEnter()**
- **ProDwgtableDisplay()**

The information required to build a table is contained in an opaque data structure, *ProDwgtabledata* that has to be allocated and filled before the table can be created. The function **ProDwgtabledataAlloc()** allocates the data.

ProDwgtabledataOriginSet() sets the position of the top left corner of a table in the *ProDwgtabledata* description.

ProDwgtabledataSizetypeSet() specifies whether the size of the columns and rows will be in screen coordinates, or as the number of text characters. It is usually more convenient to specify as numbers of characters.

ProDwgtabledataColumnsSet() sets the width of the columns as well as the default justifications of text in the columns.

ProDwgtabledataRowsSet() sets the height of the rows.

ProDwgtableTextEnter() adds the specified lines of text into the appropriate column and row of the table. You can write more than one line of text in a cell. If a line of text is too wide for the cell, then the text overlaps the neighboring cells.

ProDrawingTableCreate() creates the table in the Pro/ENGINEER drawing and optionally displays it. If your program is about to add rows, columns, or text to the table, it is usually better not to draw it immediately. It can be drawn later using **ProDwgtableDisplay()** and this will avoid multiple redrawing.

Reading Drawing Tables

Functions Introduced:

- **ProDrawingTableVisit()**
- **ProDrawingTablesCollect()**
- **ProDwgtableInfoGet()**
- **ProDwgtableColumnsCount()**
- **ProDwgtableRowsCount()**
- **ProDwgtableColumnSizeGet()**
- **ProDwgtableRowSizeGet()**
- **ProDwgtableCellNoteGet()**
- **ProDwgtableCelltextGet()**
- **ProDwgtableIsFromFormat()**
- **ProDwgtableRetrieve()**

ProDrawingTableVisit() visits all the tables in a specified drawing; it conforms to the usual standard for visit functions.

ProDrawingTablesCollect() is an alternative, and returns an array of *ProDwgtable* handles for a drawing.

A table may be divided into several segments, which are numbered sequentially from 0. The function **ProDwgtableInfoGet()** takes a *ProDwgtable* and a segment ID as input, and fills a data structure that describes the properties of the table. If the segment does not exist, it returns PRO_TK_NOT_EXIST. The properties of the table are as follows:

<pre>int rotation;</pre>	<pre>The number of 90 degree turns clockwise.</pre>

<code>double seg_origin[3];</code>	The screen coordinates of the top left corner of the segment.
<code>int nrows;</code>	The number of rows.
<code>int ncols;</code>	The number of columns.
<code>double outline[2][3];</code>	The outline of the segment.
<code>double seg_char_height;</code>	The text height used for the segment.
<code>double table_char_height;</code>	The text height used for the drawing.
<code>double char_width;</code>	The character width factor.

The functions **ProDwgtableRowsCount()** and **ProDwgtableColumnsCount()** return the number of rows and columns in a table respectively.

The functions **ProDwgtableColumnSizeGet()** and **ProDwgtableRowSizeGet()** give the column width and row height for a specified table column and row respectively.

The text item in each cell of a drawing table is stored as a detail note. If you need to modify the note in some way, for example the style, you can use the **ProDtlnote*()** functions described in the section on Detail Items. The function **ProDwgtableCellNoteGet()** returns the handle to the detail note that represents the text in a specified table cell.

The function **ProDwgtableCelltextGet()** places the text of the table into a string array.

The function **ProDwgtableIsFromFormat()** indicates whether a table was added to the table as a result of importing a format.

The function **ProDwgtableRetrieve()** retrieves a drawing table from a properly formatted Pro/ENGINEER table file, and places it in the specified drawing. It allows you to add a table to a drawing without having to specify all the table properties. This function also supports parameter tables exported in the CSV or TXT format from the Parameter dialog box or using **ProParameterTableExport()**. Refer to the Core: Parameters chapter for more information on parameters.

Modifying Drawing Tables

Functions Introduced:

- **ProDwgtableRowAdd()**
- **ProDwgtableColumnAdd()**
- **ProDwgtableRowDelete()**
- **ProDwgtableColumnDelete()**
- **ProDwgtableColumnWidthSet()**
- **ProDwgtableRowHeightSet()**
- **ProDwgtableCellsMerge()**
- **ProDwgtableCellsRemesh()**
- **ProDwgtableCelltextWrap()**
- **ProDwgtableSave()**
- **ProDwgtableRotate()**
- **ProDwgtableErase()**
- **ProDwgtableDelete()**

ProDwgtableRowAdd() and **ProDwgtableColumnAdd()** can add a row or a column before or after an existing row or column. The input arguments *height_in_chars* and *width_in_chars* specify the row height and column width in characters, respectively. Another input argument *display* specifies whether the updated table should be displayed. When making many changes to a table, it is advisable not to display them immediately, but to use **ProDwgtableDisplay()** to update the display later.

ProDwgtableRowDelete() and **ProDwgtableColumnDelete()** delete any specified row or column, including removing the text from the affected cells.

ProDwgtableColumnWidthSet() and **ProDwgtableRowHeightSet()** assign the width of a specified column and the height of a specified row, respectively, depending upon the size of the drawing table. The drawing table size given by the enumerated type *ProDwgtableSizetype* can be of the following types:

- **PRODWTABLESIZE_CHARACTERS**—Specifies the size in characters.
- **PRODWTABLESIZE_SCREEN**—Specifies the size in screen coordinates.

ProDwgtableCellsMerge() allows the merging of cells within a specified range of rows and columns, to form a single cell. The new cell can be addressed (for example, when using **ProDwgtableTextEnter()**, or other calls to **ProDwgtableCellsMerge()**) by the row and column number of the original top left cell. Rows below, and columns to the right, retain their original numbers. The function **ProDwgtableCellsRemesh()** unmerges all the merged cells in a specified range of rows and columns.

ProDwgtableCelltextWrap() wraps the text in a cell.

ProDwgtableSave() saves a drawing table in one of the formats listed below. The formats given by the enumerated type *ProDwgtableFormattype* can be of the following types:

- **PRODWGTABLEFORMAT_TBL**—Specifies the tabular format.
- **PRODWGTABLEFORMAT_TXT**—Specifies the text format.
- **PRODWGTABLEFORMAT_CSV**—Specifies the CSV format.

ProDwgtableRotate() rotates a table 90 degrees clockwise.

ProDwgtableErase() erases a drawing table.

ProDwgtableDelete() deletes a drawing table.

Notification Functions

Pro/TOOLKIT notifications are available when a drawing table or a row from a drawing table is deleted. The notification functions are established in a session using the function **ProNotificationSet()**.

- **ProDwgtableDeletePreAction()**
- **ProDwgtableDeletePostAction()**
- **ProDwgtableRowDeletePreAction()**
- **ProDwgtableRowDeletePostAction()**

The notification function **ProDwgtableDeletePreAction()** is called before deletion of a drawing table. This function is available by calling **ProNotificationSet()** with the value of the notify type as **PRO_DWGTABLE_DELETE_PRE**.

The notification function **ProDwgtableDeletePostAction()** is called after deletion of a drawing table. This function is available by calling **ProNotificationSet()** with the value of the notify type as **PRO_DWGTABLE_DELETE_POST**.

The notification function **ProDwgtableRowDeletePreAction()** is called before the deletion of a row from the selected drawing table. The input arguments for this function are as follows:

- *table*—The table containing the row to be deleted.
- *i_row*—The index of the row to be deleted.

Note: The index of the first row is 1.

This function is available by calling **ProNotificationSet()** with the value of the notify type as **PRO_DWGTABLE_ROW_DELETE_PRE**. The specified row is not deleted if the application returns an error from this callback. If the user cancels the deletion, an appropriate message should be displayed, if required.

The notification function **ProDwgtableRowDeletePostAction()** is called after the deletion of a row from the selected drawing table. This function is available by calling **ProNotificationSet()** with the value of the notify type as **PRO_DWGTABLE_ROW_DELETE_POST**.

Example 14: Creation of Table Listing Datum Points

The following example shows a command that creates a drawing table listing the datum points in a model shown in a drawing view.

```
#include <ProToolkit.h>
#include <ProDwgtable.h>
#include <ProPoint.h>
/*-----*\
  Data structure for information needed about datum points
\*-----*/
typedef struct point_t {
    ProFeature feature;
    ProVector  position;
    ProName    name;
} Point_t;

int UsrPointsCollect();
int UsrTableTextAdd();
/*=====*\
FUNCTION : UsrPointTable()
PURPOSE  : Command to create a table of datum points
\*=====*/
int UsrPointTable ()
{
    ProError status;
    ProSelection *sel;
    int n_sel, n_points, p;
```

```

ProSolid solid;
ProCharName name, type, csys_name;
ProName wname;
ProGeomitem csys_geom;
ProAsmcomppath csys_comppath;
ProMatrix from_csys, to_csys, trf;
ProGeomitemdata *gdata;
ProMouseButton button;
ProVector pos;
Point_t *points;
ProDwgtabledata tdata;
double widths[] = {8.0, 8.0, 10.0, 10.0, 10.0};
ProHorzJust justs[] = {
    PROHORZJUST_LEFT,
    PROHORZJUST_LEFT,
    PROHORZJUST_LEFT,
    PROHORZJUST_LEFT,
    PROHORZJUST_LEFT};

double *heights;
int last_feat_row;
ProDrawing drawing;
ProDwgtable table;
ProMdlldata mdata;
ProCharLine line;
ProFileName msgfil;
ProStringToWstring (msgfil, "msg_ugdrawing.txt");
/*-----*\
Select a coordinate system. This defines the model (the top one
in that view), and the reference for the datum point positions.
/*-----*\
ProMessageDisplay(msgfil, "USER Select csys");
status = ProSelect("csys", 1, NULL, NULL, NULL, NULL, &sel, &n_sel);
if(status != PRO_TK_NO_ERROR || n_sel < 1)
    return(0);
/*-----*\
Extract the csys handle, and assembly path.
/*-----*\
ProSelectionModelitemGet(sel[0], &csys_geom);
ProSelectionAsmcomppathGet(sel[0], &csys_comppath);
/*-----*\
Extract the csys location
/*-----*\
ProGeomitemdataGet(&csys_geom, &gdata);
ProMatrixInit(gdata->data.p_csys_data->x_vector,
    gdata->data.p_csys_data->y_vector,
    gdata->data.p_csys_data->z_vector,
    gdata->data.p_csys_data->origin,
    from_csys);
ProUtilMatrixInvert(from_csys, to_csys);
ProGeomitemdataFree(&gdata);
/*-----*\

```

```

Extract the cys name
/*-----*/
    ProModelitemNameGet(&csys_geom, wname);
    ProWstringToString(csys_name, wname);
/*-----*/
Get the root solid, and the transform from the root to the
component owning the csys
/*-----*/
    if(csys_comppath.table_num > 0)
    {
        solid = csys_comppath.owner;
        ProAsmcomppathTrfGet(&csys_comppath, PRO_B_FALSE, trf);
    }
    else
    {
        solid = csys_geom.owner;
        ProUtilMatrixCopy(NULL, trf);
    }
/*-----*/
Get a list of datum points in the model
/*-----*/
    UspointsCollect(solid, &points);
    ProArraySizeGet(points, &n_points);
    if(n_points < 1)
        return(0);
/*-----*/
Get the user to select the table position
/*-----*/
    ProMessageDisplay(msgfil, "USER Pick table position");
    if(ProMousePickGet(PRO_ANY_BUTTON, &button, pos) != PRO_TK_NO_ERROR)
        return(0);
/*-----*/
Setup the table data
/*-----*/
    ProDwgtabledataAlloc(&tdata);
    ProDwgtabledataOriginSet(tdata, pos);
    ProDwgtabledataSizetypeSet(tdata, PRODWGTABLESIZE_CHARACTERS);
    ProDwgtabledataColumnsSet(tdata, 5, widths, justs);

    heights = (double*)calloc(n_points+2, sizeof(double));
    for(p=0;p<n_points+2;p++)
        heights[p] = 1.0;
    ProDwgtabledataRowsSet(tdata, n_points+2, heights);
    free(heights);
/*-----*/
Create the table
/*-----*/
    ProMdlCurrentGet((ProMdl*)&drawing);
    ProDrawingTableCreate(drawing, tdata, PRO_B_FALSE, &table);
/*-----*/
Merge the top row cells to form the header

```

```

\*-----*/
    ProDwgTableCellsMerge(&table, 1, 1, 5, 1, PRO_B_FALSE);
\*-----*/
Write header text specifying model and csys
\*-----*/
    ProMdlDataGet(solid, &mdata);
    ProWstringToString(name, mdata.name);
    ProWstringToString(type, mdata.type);
    ProTKSprintf(line, "Datum points for %s.%s, w.r.t csys %s\n",
                    name, type, csys_name);
    UsrTableTextAdd(&table, 1, 1, line);
\*-----*/
Add subheadings to columns
\*-----*/
    UsrTableTextAdd(&table, 1, 2, "Feat id");
    UsrTableTextAdd(&table, 2, 2, "Point");
    UsrTableTextAdd(&table, 3, 2, "X");
    UsrTableTextAdd(&table, 4, 2, "Y");
    UsrTableTextAdd(&table, 5, 2, "Z");
\*-----*/
For each datum point...
\*-----*/
    for(p=0;p<n_points;p++)
    {
\*-----*/
        If the owning feature is the same as the last one, just
        merge column 1 with the cell above, else enter the feature id
\*-----*/
        if(p == 0 || points[p].feature.id != points[p-1].feature.id)
        {
            ProTKSprintf(name, "%d", points[p].feature.id);
            UsrTableTextAdd(&table, 1, p+3, name);
            last_feat_row = p+3;
        }
        else
        {
            ProDwgTableCellsMerge(&table, 1, last_feat_row, 1, p+3, PRO_B_FALSE);
\*-----*/
            Add the point name to column 2
\*-----*/
            ProWstringToString(name, points[p].name);
            UsrTableTextAdd(&table, 2, p+3, name);
\*-----*/
            Transform the location w.r.t to the csys
\*-----*/
            ProPntTrfEval(points[p].position, trf, points[p].position);
            ProPntTrfEval(points[p].position, to_csys, points[p].position);
\*-----*/
            Add the XYZ to column 3,4,5
\*-----*/
            ProTKSprintf(name, "%8.3f", points[p].position[0]);
            UsrTableTextAdd(&table, 3, p+3, name);

```

```

        ProTKSprintf(name,"%8.3f", points[p].position[1]);
        UstrTableTextAdd(&table, 4, p+3, name);
        ProTKSprintf(name,"%8.3f", points[p].position[2]);
        UstrTableTextAdd(&table, 5, p+3, name);
    }
/*-----*\
Display the table
\*-----*/
    ProDwgtableDisplay(&table);
    return(1);
}
/*=====*\
FUNCTION : UstrTableTextAdd()
PURPOSE  : Utility to add one text line to a table cell
\*=====*/
int UstrTableTextAdd(
    ProDwgtable *table,
    int col,
    int row,
    char *text)
{
    ProWstring *lines;
    ProArrayAlloc(1, sizeof(ProWstring), 1, (ProArray*)&lines);
    lines[0] = (wchar_t*)calloc(strlen(text) + 1, sizeof(wchar_t));
    ProStringToWstring(lines[0], text);
    ProDwgtableTextEnter(table, col, row, lines);
    ProArrayFree((ProArray*)&lines);
}
/*=====*\
FUNCTION : UstrPointAction()
PURPOSE  : Visit action function called for each datum point
\*=====*/
ProError UstrPointAction(
    ProGeomitem *geomitem,
    ProError filt_status,
    ProAppData data)
{
    Point_t point;
    ProPoint p;
/*-----*\
Find out which feature the datum point belongs to
\*-----*/
    ProGeomitemFeatureGet(geomitem, &point.feature);
/*-----*\
Get the point position
\*-----*/
    ProPointInit(geomitem->owner, geomitem->id, &p);
    ProPointCoordGet(p, point.position);
/*-----*\
Get the point name
\*-----*/

```

```

    ProModelitemNameGet(geomitem, point.name);
/*-----*\
Add the point to the array
\*-----*/
    ProArrayObjectAdd((ProArray*)data, -1, 1, &point);
    return(PRO_TK_NO_ERROR);
}
/*=====*\
FUNCTION : UsrFeatureAction()
PURPOSE  : Visit action function called for each feature
\*=====*/
ProError UsrFeatureAction(
    ProFeature *feature,
    ProError filt_status,
    ProAppData data)
{
    ProFeatStatus fstatus;
/*-----*\
If the feature is not active, skip it
\*-----*/
    ProFeatureStatusGet(feature, &fstatus);
    if(fstatus != PRO_FEAT_ACTIVE)
        return(PRO_TK_NO_ERROR);
/*-----*\
Visit the datum points in the feature
\*-----*/
    ProFeatureGeomitemVisit(feature, PRO_POINT, UsrPointAction,
                            NULL, data);
    return(PRO_TK_NO_ERROR);
}
/*=====*\
FUNCTION : UsrPointsCollect()
PURPOSE  : Collect an array of datum points in the solid
\*=====*/
int UsrPointsCollect(
    ProSolid solid,
    Point_t **points)
{
/*-----*\
Allocate the array
\*-----*/
    ProArrayAlloc(0, sizeof(Point_t), 1, (ProArray*)points);
/*-----*\
Visit the features
\*-----*/
    ProSolidFeatVisit(solid, UsrFeatureAction, NULL, points);
}

```


Drawing Table Segments

Functions Introduced:

- **ProDwgtableSegMove()**
- **ProDwgtableSegCount()**
- **ProDwgtableSegSheetGet()**
- **ProDwgtableSegmentOriginSet()**

Drawing tables can be constructed with one or more segments. Each segment can be independently placed.

Note: For these functions, pass -1 to refer to the only segment of a one-segment table.

Move a drawing table segment to new screen coordinates with the function **ProDwgtableSegMove()**. Pass the coordinates of the target position in format x, y, z=0 to this the function. It moves the table segment to the target position.

Determine the number of segments in a table with the function **ProDwgtableSegCount()**. Pass the name of the table to the function and it returns the number of table segments.

Use the function **ProDwgtableSegSheetGet()** to determine which sheet contains a specified drawing table segment.

Use the function **ProDwgtableSegmentOriginSet()** to assign the origin for a specified drawing table segment.

Repeat Regions

Functions Introduced:

- **ProDwgtableCellIsComment()**
- **ProDwgtableCellComponentGet()**
- **ProDwgtableCellRefmodelGet()**
- **ProDrawingTablesUpdate()**

The functions **ProDwgtableCellIsComment()**, **ProDwgtableCellComponentGet()**, **ProDwgtableCellRefmodelGet()**, and **ProDrawingTablesUpdate()** apply to repeat regions in drawing tables.

ProDwgtableCellIsComment() indicates whether a cell in a repeat region contains a comment.

ProDwgtableCellComponentGet() returns the full path to the component referenced in a cell in a repeat region of a drawing table. However, this function does not return a valid path if the cell has the attribute “NO DUPLICATE” or “NO DUPLICATE/LEVEL” since there is no unique path available. In this case, use **ProDwgtableCellRefmodelGet()** to return the reference assembly and component referred to by the cell in a repeat region. This function differs from **ProDwgtableCellComponentGet()** such that it returns reference objects, even if the cell attribute is set to “NO DUPLICATE” or “NO DUPLICATE/LEVEL”.

The function **ProDrawingTablesUpdate()** updates repeat regions in all tables to account for changes to the model or models. It is equivalent to the Pro/ENGINEER command **Table**, **Repeat Region**, or **Update**.

Drawing Dimensions

This section describes Pro/TOOLKIT functions that give access to the types of dimension that can be created in drawing mode. They do not apply to dimensions which are created in solid mode, either those created automatically as a result of feature creation, or reference dimensions created in a solid.

The ProDimension object is introduced in the section on Dimensions; read the explanation of ProDimension at the start of that section before reading further.

Dimensions created in drawing mode are stored either in the solid or in the drawing, depending upon the setting of the config.pro option CREATE_DRAWING_DIMS_ONLY. The default is “NO”, meaning that the dimensions will be stored in the solid. (Refer to the Pro/DRAWING User's Guide for more information on the various types of created dimension, and their behavior.)

The “owner” field in the ProDimension object always refers to the model in which the dimension is stored.

The function **ProDrawingDimensionVisit()**, described in the section on Dimensions, can be used to find all the dimensions stored in a drawing.

Drawing Dimension Attachments and Dimension Creation

Functions Introduced:

- **ProDrawingDimAttachsGet()**
- **ProDrawingDimAttachpointsGet()**
- **ProDrawingDimCreate()**

The function **ProDrawingDimAttachsGet()** provides information about the entities to which a dimension is attached.

Note: Dimensions created in solid mode are stored in a different way from those created in a drawing, because of their different role, and their attachments are not accessible to this function. If the function is called for a function created in solid mode, it will return an error.

The information about an attachment entity is given in terms of two variables of these types:

- *ProSelection*—Identifies the entity.
- *ProDimSense*—Gives more information about how the dimension attaches to the entity, for example, to what part of the entity, and in what direction the dimension runs.

Note: The above function **ProDrawingDimAttachsGet()** has been deprecated since it does not support drawing dimensions attached to intersections of geometric entities. Use the function **ProDrawingDimAttachpointsGet()** instead.

The function **ProDrawingDimAttachpointsGet()** retrieves the entities to which a dimension is attached and the type of attachments. This is applicable only for dimensions created in the drawing mode. This function supports drawing dimensions attached to intersections of geometric entities.

The information about the entities to which the dimension is attached is given by the following output arguments:

- *attachments_arr*—Specifies a *ProArray* of entities to which a dimension is attached. Each attachment point is described by two consecutive array elements, of which the second one may be NULL. If both elements are not NULL, then the attachment point refers to the intersection of the elements. If the second element is NULL, then the attachment point refers to the first one.

- *dsense_arr*—Specifies a *ProArray* of *ProDimSense* that gives more information about how the dimension attaches to the entities.

ProDimSense is declared in header file `ProDimension.h`. This is the declaration:

```
typedef struct pro_dim_sense {
    ProDimSenseType    type;
    int                sense;
    ProDimAngleSense   angle_sense;
    ProDimOrient        orient_hint;
} ProDimSense;
```

The “type” field indicates what type of information is being provided by the “sense” and/or “angle_sense” fields. The following sections list the values of *ProDimSenseType* with an explanation of the value of “sense” and “angle_sense” needed in each case.

Note: Some of the explanations below refer to the “direction” of an entity. Each entity (which includes 3D edges and curves, and 2D draft entities) has an inherent direction which is the direction in which its parameter “t” increases. In the data structure description of the entity, *ProCurvedata*, the first end specified is always the end at which t=0. For example, the direction of a line entity is from field “end1” to “end2” in *ProLinedata*.

- `PRO_DIM_SNS_TYP_NONE`—In this case, no other information is needed to describe the attachment points. For example, if there is a single attachment which is a straight line, the dimension is the length of the line; if the attachments are two parallel lines, the dimension is the distance between them.
- `PRO_DIM_SNS_TYP_PNT`—In this case the “sense” field is set to a value of the enum *ProPointType* (declared in the header `ProPoint.h`), which specifies the part of the entity to which the dimension is attached. The possible values are these:
 - `PRO_POINT_TYP_END1`—The first end (that is, where “t” = 0)
 - `PRO_POINT_TYP_END2`—The second end (that is, where “t” = 1.0)
 - `PRO_POINT_TYP_CENTER`—The center, if entity is an arc or a circle
 - `PRO_POINT_TYP_NONE`—This is equivalent to setting “type” to `PRO_DIM_SNS_TYP_NONE`.

- PRO_POINT_TYP_MIDPT—The midpoint of the entity (where “t” = 0.5)
- PRO_DIM_SNS_TYP_SPLN_PNT—This means that the attachment is to a point of a spline. The “sense” field is set to the index of the spline point.
- PRO_DIM_SNS_TYP_TGT_IDX —The dimension attaches to a tangent of the entity, which is an arc or circle. The “sense” field is set to the index of the tangent in a list of all possible tangents ordered by the “t” value at which they touch the entity.
- PRO_DIM_SNS_TYP_LIN_AOC_TGT—The dimension is the perpendicular distance between a line and a tangent to an arc or a circle which is parallel to the line. The value of “sense” is one of the values of the enum *ProDimLinAocTgtSense*. If the two possible tangents are on different sides of the line entity (because the distance from the line to the center is less than the radius) then the two tangents are distinguished as left or right of the line (with respect to its natural direction). If the two tangents are on the same side of the line (because the distance from the line to the center is more than the radius), the two tangents are distinguished as on the same side of the arc/circle center (0) or on the opposite side (1). There is an enum value for each of the four possible combinations of ways to identify a tangent, though of course only two are possible for a particular line and arc/circle pair.

The four values of *ProDimLinAocTgtSense* are:

- PRO_DIM_LIN_AOC_TGT_LEFT0—The tangent is to the left of the line, and on the same side of the center of the arc/circle as the line.
- PRO_DIM_LIN_AOC_TGT_RIGHT0 —The tangent is to the right of the line, and on the same side of the center of the arc/circle as the line
- PRO_DIM_LIN_AOC_TGT_LEFT1—The tangent is to the left of the line, and on the opposite side of the center of the arc/circle to the line.
- PRO_DIM_LIN_AOC_TGT_RIGHT1—The tangent is to the right of the line, and on the opposite side of the center of the arc/circle to the line.
- PRO_DIM_SNS_TYP_ANGLE—The dimension is the angle between two straight entities. The field “angle_sense” is given by the structure *ProDimAngleSense* which contains three boolean fields. They have the following meaning:

- `is_first`—Is set to TRUE if the angle dimension starts from this entity in a counterclockwise direction; FALSE if the dimension ends at this entity. The value must be TRUE for one entity, and FALSE for the other.
- `should_flip`—If “`should_flip`” is FALSE, and the entity's inherent direction is away from the angle vertex, then the dimension attaches directly to the entity. If the entity's direction is towards the angle vertex, the dimension is attached to a witness line which is in line with the entity but on the opposite side of the angle vertex—If “`should_flip`” is TRUE, then these cases are interchanged.
- `pic_vec_dir`—Reserved for future use.
- `PRO_DIM_SNS_TYP_PNT_ANGLE`—The dimension is the angle between a line entity and the tangent to a curved entity at one of its ends. The curve attachment is of this type. (The line attachment is of the type `PRO_DIM_SNS_TYP_PNT` described above.) In this case both the “`angle`” and “`angle_sense`” fields must be set: “`sense`” shows which end of the curve the dimension is attached to; “`angle_sense`” shows the direction in which the dimension rotates and in which side of the tangent it attaches.

The field “`orient_hint`” describes the orientation of the dimension in cases where this cannot be deduced from the attachments themselves. (When such a dimension is created interactively in Pro/ENGINEER, the user is prompted for the extra information.) For example, if the attachments are datum points that are not vertically or horizontally aligned, Pro/ENGINEER needs to know whether the dimension is to be horizontal, vertical, or slanted. The hint refers to the dimension itself, not the attachment, although it is a field in *ProDimSense*. Pro/ENGINEER looks at the value of “`orient_hint`” in the first item in the *ProDimSense* array you provide.

The values of *ProDimOrient* are:

- `PRO_DIM_ORNT_NONE`—No orientation information is needed or provided.
- `PRO_DIM_ORNT_HORIZ`—The dimension is horizontal
- `PRO_DIM_ORNT_VERT`—The dimension is vertical
- `PRO_DIM_ORNT_SLANTED`—The dimension is slanted
- `PRO_DIM_ORNT_ELPS_RAD1`—The major diameter of an ellipse

- **PRO_DIM_ORNT_ELPS_RAD2**—The minor diameter of an ellipse
- **PRO_DIM_ORNT_ARC_ANG**—The angle of an arc
- **PRO_DIM_ORNT_ARC_LENGTH**—The length of an arc
- **PRO_DIM_ORNT_LIN_TANCRV_ANG**—If the dimension is attached to a line and an end point of a curve, the default dimension will be a linear dimension showing the distance between the line and the curve point. If you want the dimension to show instead the angle between the line and the tangent at the curve point, set “orient_hint” to this value.

The function **ProDrawingDimCreate()** creates a dimension in a drawing. It takes as input an array of *ProSelection* objects and an array of *ProDimSense* structures that describe the required attachments. It will store the new dimension in the solid or the drawing depending upon the setting of the *config.pro* option **CREATE_DRAWING_DIMS_ONLY**.

The dimension will be added to the drawing view specified in the *ProSelection* object(s). If you want to build the attachment *ProSelection* object programmatically by calling **ProSelectionAlloc()**, rather than interactively using **ProSelect()**, call the function **ProSelectionViewSet()** to ensure that your *ProSelection* specifies the drawing view.

The function outputs a *ProDimension* object to identify the new dimension.

Ordinate Dimensions

Functions Introduced:

- **ProDrawingDimIsOrdinate()**
- **ProDrawingOrdbaselineCreate()**
- **ProDrawingDimToOrdinate()**
- **ProDrawingDimToLinear()**

The function **ProDrawingDimIsOrdinate()** tells you whether a particular dimension is an ordinate dimension. If so, it also outputs a *ProDimension* object to identify the baseline dimension being referenced.

The function **ProDrawingOrdbaselineCreate()** converts a specified dimension to an ordinate baseline dimension. The choice of which end of the dimension becomes the baseline is made by an input of type ProVector which should be close to the appropriate attachment entity, and be in 3d solid coordinates. The function outputs a new ProDimension object which is used to identify the baseline dimension when converting to ordinate other dimensions which should share that baseline.

The function **ProDrawingDimToOrdinate()** converts an existing linear dimension to ordinate. It requires as one of its inputs a ProDimension object that was output from **ProDrawingOrdbaselineCreate()** as input to identify the baseline.

The function **ProDrawingDimToLinear()** converts an existing ordinate dimension to linear.

The last three functions in this section all update the display of the dimension if it is currently displayed.

Example 15: Command Creation of Datum Point Table

The example below shows a command which creates vertical and horizontal ordinate dimensions from each datum point in a model in a drawing view to a selected coordinate system datum.

```
#include <ProToolkit.h>
#include <ProDrawing.h>
#include <ProDimension.h>
#include <ProArray.h>
#include <ProSolid.h>
#include <ProFeature.h>
#include <ProPoint.h>
#include <ProMessage.h>

int UsrPointGeomitemsCollect(ProSolid solid, ProGeomitem **points);

/*=====*\
FUNCTION : UsrDimPoints()
PURPOSE  : Command to create a table of datum points
\*=====*/
int UsrDimPoints() {
    ProError status;
    ProSelection *sel, csys_sel;
    int n_sel, n_points, p;
    ProSolid solid;
    ProGeomitem csys_geom;
    ProAsmcomppath csys_comppath;
    ProMatrix trf;
```



```

    ProVector csys_pos, csys_3dpos, pnt_pos, dim_pos, outline[2];
    ProGeomitem *points;
    ProDrawing drawing;
    ProSelection *attachments;
    ProDimSense *senses;
    ProDimension dimension, vbase_dim, hbase_dim;
    ProPoint point;
    ProView view;
    ProMdlCurrentGet((ProMdl*)&drawing);
/*-----*\
    Select a coordinate system. This defines the model (the top one
    in that view), and the common attachments for the dimensions
/*-----*\
    ProMessageDisplay(msgfil,"USER Select csys");
    status = ProSelect("csys",1,NULL,NULL,NULL,NULL,&sel,&n_sel);
    if(status != PRO_TK_NO_ERROR || n_sel < 1)
        return(0);
    ProSelectionCopy(sel[0], &csys_sel);
    ProSelectionModelitemGet(csys_sel, &csys_geom);
    ProSelectionAsmcomppathGet(csys_sel, &csys_comppath);
/*-----*\
Get the root solid
/*-----*\
    if(csys_comppath.table_num > 0)
        solid = csys_comppath.owner;
    else
        solid = csys_geom.owner;
/*-----*\
Get a list of datum points in the model
/*-----*\
    UsrPointGeomitemsCollect(solid, &points);
    ProArraySizeGet(points, &n_points);
    if(n_points < 1)
        return(0);
/*-----*\
    Calculate where the csys is located on the drawing
/*-----*\
    ProSelectionPoint3dGet(csys_sel, csys_pos);
    if(csys_comppath.table_num > 0)
    {
        ProAsmcomppathTrfGet(&csys_comppath, PRO_B_TRUE, trf);
        ProPntTrfEval(csys_pos, trf, csys_pos);
    }
    memcpy(csys_3dpos, csys_pos, sizeof(ProVector));
    ProSelectionViewGet(csys_sel, &view);
    ProDrawingViewTransformGet(drawing, view, PRO_B_TRUE, trf);
    ProPntTrfEval(csys_pos, trf, csys_pos);
/*-----*\
    Get the view outline
/*-----*\
    ProDrawingViewOutlineGet(drawing, view, outline);
/*-----*\

```

```

Allocate the attachment arrays
\*-----*/
    ProArrayAlloc(2, sizeof(ProSelection), 1, (ProArray*)&attachments);
    ProArrayAlloc(2, sizeof(ProDimSense), 1, (ProArray*)&senses);
\*-----*/
For each datum point...
\*-----*/
    for(p=0;p<n_points;p++)
    {

\*-----*/
Calculate the position of the point on the drawing
\*-----*/
ProPointInit(points[p].owner, points[p].id, &point);
    ProPointCoordGet(point, pnt_pos);
    ProPntTrfEval(pnt_pos, trf, pnt_pos);

\*-----*/
Set up the "sense" information for the point attachment
    (Specify a vertical dimension)
\*-----*/
    senses[0].type = PRO_DIM_SNS_TYP_PNT;
    senses[0].sense = PRO_POINT_TYP_CENTER;
    senses[0].orient_hint = PRO_DIM_ORNT_VERT;

\*-----*/
Set up the "sense" information for the csys attachment
\*-----*/
senses[1].type = PRO_DIM_SNS_TYP_PNT;
    senses[1].sense = PRO_POINT_TYP_CENTER;
    senses[1].orient_hint = PRO_DIM_ORNT_NONE;

\*-----*/
    Set the attachment information
\*-----*/
ProSelectionAlloc(NULL, &points[p], &attachments[0]);
    ProSelectionViewSet(view, &attachments[0]);
    ProSelectionCopy(csys_sel, &attachments[1]);

\*-----*/
    Calculate the dim position to be just to the left of the
        drawing view, midway between the point and csys
\*-----*/
dim_pos[0] = outline[0][0] - 20.0;
    dim_pos[1] = (csys_pos[1] + pnt_pos[1]) / 2.0;
    dim_pos[2] = 0.0;

\*-----*/
Create and display the dimension
\*-----*/
    ProDrawingDimCreate(drawing, attachments, senses, dim_pos,
        PRO_B_FALSE, &dimension);
    ProAnnotationShow ((ProAnnotation*)&dimension, NULL, view);

\*-----*/
        If this is the first vertical dim, create an ordinate base
            line from it, else just convert it to ordinate

```

```

/*-----*/
if(p==0)
    ProDrawingOrdbaselineCreate(drawing, &dimension,
                                csys_3dpos, &vbase_dim);
    else
        ProDrawingDimToOrdinate(drawing, &dimension, &vbase_dim);
/*-----*/
    Set this dimension to be horizontal
/*-----*/
    senses[0].orient_hint = PRO_DIM_ORNT_HORIZ;
/*-----*/
    Calculate the dim position to be just to the bottom of the
        drawing view, midway between the point and csys
/*-----*/
    dim_pos[0] = (csys_pos[0] + pnt_pos[0]) / 2.0;
    dim_pos[1] = outline[0][1] - 20.0;
/*-----*/
Create and display the dimension
/*-----*/
    ProDrawingDimCreate(drawing, attachments, senses, dim_pos,
                        PRO_B_FALSE, &dimension);
    ProAnnotationShow ((ProAnnotation*)&dimension, NULL, view);
/*-----*/
    If this is the first horizontal dim, create an ordinate base line
        from it, else just convert it to ordinate
/*-----*/
if(p==0)
    ProDrawingOrdbaselineCreate(drawing, &dimension,
                                csys_3dpos, &hbase_dim);
    else
        ProDrawingDimToOrdinate(drawing, &dimension, &hbase_dim);
/*-----*/
    Free the attachment selection objects
/*-----*/
ProSelectionFree(&attachments[0]);
    ProSelectionFree(&attachments[1]);
}
return(1);
}
/*=====*/
FUNCTION : UsrPointAddAction()
PURPOSE  : Visit action function called for each datum point
/*=====*/
ProError UsrPointAddAction((
    ProGeomitem *geomitem,
    ProError filt_status,
    ProAppData data) {
/*-----*/
    Add the point to the array
/*-----*/
ProArrayObjectAdd((ProArray*)data, -1, 1, geomitem);
    return(PRO_TK_NO_ERROR); }

```

```

/*=====*\
FUNCTION : UsrFeatureGetPointsAction()
PURPOSE: Visit action function called for each feature
\*=====*/
ProError UsrFeatureGetPointsAction(
    ProFeature *feature,
    ProError filt_status,
    ProAppData data)
{
    ProFeatStatus fstatus;
/*-----*\
If the feature is not active, skip it
\*-----*/
    ProFeatureStatusGet(feature, &fstatus);
    if(fstatus != PRO_FEAT_ACTIVE)
        return(PRO_TK_NO_ERROR);
/*-----*\
Visit the datum points in the feature
\*-----*/
    ProFeatureGeomitemVisit(feature, PRO_POINT, UsrPointAddAction, NULL,
        data);
    return(PRO_TK_NO_ERROR);
}
/*=====*\
FUNCTION : UsrPointGeomitemsCollect()
PURPOSE: Collect an array of datum points in the solid
\*=====*/
int UsrPointGeomitemsCollect(
    ProSolid solid,
    ProGeomitem **points)
{
/*-----*\
Allocate the array
\*-----*/
    ProArrayAlloc(0, sizeof(ProGeomitem), 1, (ProArray*)points);
/*-----*\
Visit the features
\*-----*/
    ProSolidFeatVisit(solid, UsrFeatureGetPointsAction, NULL, points);
}

```

Other Drawing Dimension Properties

Functions Introduced:

- **ProDrawingDimIsAssociative()**
- **ProDrawingDimensionIsDisplayed()**
- **ProDrawingDimensionIsToleranceDisplayed()**
- **ProDrawingDimensionViewGet()**
- **ProDrawingDimSwitchView()**
- **ProDrawingDimensionPosGet()**
- **ProDrawingDimensionMove()**
- **ProDrawingDualDimensionGet()**
- **ProDrawingDimensionPathGet()**

If you create a dimension which attaches only to draft entities in the drawing, the dimension may be associative or non-associative. This setting depends on the setting of the drawing setup option “associative_dimensioning”. The associative status of a dimension remains even when the setup option changes, so a drawing may contain dimensions of both types. The function

ProDrawingDimIsAssociative() tells you whether or not a particular dimension is associative. Refer to the Drawing User's Guide for a fuller description of associative draft dimensions.

The function **ProDrawingDimensionIsDisplayed()** identifies whether a drawing dimension is displayed in the drawing.

The function **ProDrawingDimensionIsToleranceDisplayed()** identifies whether the tolerance value is displayed on the drawing dimension.

The function **ProDrawingDimensionViewGet()** tells you what drawing view a dimension is being displayed in.

ProDrawingDimSwitchView() allows you to switch it's display to another view. Note that not all views will support the display of particular dimension.

The function **ProDrawingDimensionPosGet()** gives the position of the text of a specified dimension.

Note: This function has been deprecated from Pro/ENGINEER Wildfire 3.0 onwards. Use the function **ProDimlocationTextGet()** to get the location of the dimension text.

The function **ProDrawingDimensionMove()** allows you to move the dimension text to a new position on the drawing. Use the function **ProDimlocationTextGet()** to get the position before and after the move.

The function **ProDrawingDualDimensionGet()** identifies whether a drawing is using dual dimensioning, and also specifies the properties of the dual dimensioning.

The function **ProDrawingDimensionPathGet()** extracts the component path for a dimension displayed in a drawing.

Production Applications: Sheetmetal

This chapter describes the sheet metal geometry analysis and bend table functions. It also introduces and describes the feature element trees for the sheet metal features.

Topic	Page
Geometry Analysis	52 - 2
Bend Tables and Dimensions	52 - 9
Sheetmetal Flat Wall Feature	52 - 10
Sheetmetal Flange Wall Feature	52 - 30
Punch Form Feature	52 - 58
Quilt Form Feature	52 - 64

Geometry Analysis

Pro/TOOLKIT sheet metal geometry analysis functions provide for analysis of sheet metal part geometry and ensure effective customization of sheet metal parts. These analyses include extracting part thickness data and obtaining edge and surface data for sheet metal components.

In addition, sheet metal bend edge and bend surface functions support analyses that:

- Extract bend information associated with bend lines (K-factor, bend deduction, bend allowance).
- Find bend lines when a part is in a flat state.
- Map flat state IDs to bent state IDs.

Functions Introduced:

- **ProSmtPartThicknessGet()**
- **ProSmtSurfaceTypeGet()**
- **ProSmtedgeContourGet()**
- **ProSmtOppsurfGet()**
- **ProSmtOppedgeGet()**
- **ProSmtBendsrfParentGet()**
- **ProSmtBendsrfChildGet()**
- **ProSmtBendedgeChildGet()**
- **ProSmtBendedgeParentGet()**
- **ProSmtMdllsFlatStateInstance()**
- **ProFaminstancelsFlatState()**

The function **ProSmtPartThicknessGet()** returns the dimension that defines the thickness of the specified sheet metal component. If the model contains the thickness parameter, then this dimension cannot be modified directly. Use the function

ProParameterValueSet() to assign the value of the thickness parameter. If you specify a non sheet metal part,

ProSmtPartThicknessGet() returns PRO_TK_BAD_CONTEXT.

The function **ProSmtSurfaceTypeGet()** returns the type of the specified solid surface. This enables you to determine whether a surface is created by a sheet metal feature, and to distinguish among the different types of sheet metal surfaces, such as side, white, and green.

The possible values are as follows:

- **PRO_SMT_SURF_NON_SMT**—The surface was created by a solid feature.
- **PRO_SMT_SURF_SIDE**—The surface is a side surface created by a sheet metal feature.
- **PRO_SMT_SURF_FACE**—The surface is the face (green) surface created by a sheet metal feature.
- **PRO_SMT_SURF_OFFSET**—The surface is the offset (white) surface created by a sheet metal feature.

The function **ProSmtedgeContourGet()** returns a complete contour that contains the specified edge. This function returns **PRO_TK_BAD_CONTEXT** if the edge is not on the green or white side of the specified part.

The function **ProSmtOppsurfGet()** returns a surface that is opposite (offset to) the specified surface.

The function **ProSmtOppedgeGet()** returns the edge that is opposite (offset to) the specified edge. Edge data for function **ProSmtOppedgeGet()** uses the following definitions:

- An edge is lying on a green surface if one of its surfaces has **SHEETMETAL TYPE = FACE**.
- An edge is lying on a white surface if one of its surfaces has **SHEETMETAL TYPE = OFFSET**.
- The opposite edge to an edge must be on the surface opposite the original edge's surface and must be a geometrical offset of the original edge.
- An edge is in a peripheral contour if, and only if the following are true:
 - It is in the part geometry.
 - Exactly one of its surfaces is either **FACE** or **OFFSET**.

The function **ProSmtBendsrfParentGet()** returns the parent of the specified surface. For example, if the specified surface is in bent position, this function returns the surface that is the most recent, unbent equivalent of the specified surface. See notes below.

The function **ProSmtBendsrfChildGet()** returns the active (visible) child surface of the specified, inactive (invisible) surface. A surface is active (visible) if it is in the part geometry list. See notes below.

The function **ProSmtBendedgeParentGet()** returns the parent of the specified edge. For example, if the specified edge is in bent position, this function returns the edge that is the most recent, unbent equivalent of the specified edge. See notes below.

The function **ProSmtBendedgeChildGet()** returns the active (visible) child edge of the specified, inactive (invisible) edge. An edge is active (visible) if both its surfaces are active and the edge is contained in the contours of both surfaces. See notes below.

Notes:

- Edges and surfaces in quilt geometry are also visible, but they are invalid as input to sheet metal functions.
- Surface and edge parent and child functions use the following definitions:
 - An edge or surface has a parent if the edge or surface is a result of bending or unbending another edge or surface.
 - If an edge or surface is active and is a result of bending or unbending, any parent of this edge or surface that is in the chain of bends or unbends has this edge or surface as the active child.

The function **ProSmtMdlIsFlatStateInstance()** checks if the model is a flat state instance model.

The function **ProFaminstanceIsFlatState()** checks if the family instance of the model is a sheetmetal flat instance or not.

Example 1: Sheet Metal Geometry Analysis

```
#include "ProToolkit.h"
#include "ProMdl.h"
#include "ProSolid.h"
#include "ProSurface.h"
#include "ProSheetmetal.h"
#include "ProMode.h"

/*-----*\
C System includes
\*-----*/

/*-----*\
```

```

Application includes
/*-----*/
#include <TestError.h>
#include <TestFiletypes.h>
#include <UtilMessage.h>
#include <UtilNames.h>
#include <UtilString.h>
#include <UtilTypes.h>
#include <UtilVisit.h>

/*-----*\
Application macros
/*-----*/

/*-----*\
Application data types
/*-----*/

typedef struct testsmtlsurf
{
    ProMdl    *model;
    FILE      *fp;
} ProTestSmtlSurf;

/*-----*\
Application prototypes
/*-----*/
ProError ProTestCurveCompAct( ProCurve , ProCurve , int, ProBoolean,
    ProError, ProAppData);
ProError ProTestCurveAct( ProModelitem* , ProError, ProAppData);
int ProTestGeomTraverse(ProMdl *, int);

/*=====*\
    FUNCTION : ProTestSheetmetalTraverse()
    PURPOSE  : Traverse all sheetmetal surfaces.
/*=====*/
int ProTestSheetmetalTraverse(
    ProMdl *model)
{
    ProUtilCname fname;
    ProTestSmtlSurf app_data;
    ProError status;
    FILE *fp;
    ProSurface *surfaces;
    int surfaces_num, i;

    ProError ProTestShtmtlSurfAct(ProSurface surface, ProError instatus,
        ProAppData tmp_app_data);

```

```

/*-----*\
    Get the name of the output file
\*-----*/
    ProTestQcrName(model, TRAVERSAL, fname);
    fp = fopen(fname, "w");

    if (fp == NULL)
        return(-1);

/*-----*\
    Set up the general data with the model and the file
\*-----*/
    app_data.fp = fp;
    app_data.model = model;

/*-----*\
    Set up a header for the output file
\*-----*/
    fprintf(fp, "SHEETMETAL SURFACES:\n\n");
    fprintf(fp, "  Id      Surface Type    \n");

/*-----*\
    Visit all the solid surfaces
\*-----*/

    status = ProUtilCollectSolidSurfaces ((ProSolid) *model, &surfaces);
    if (status == PRO_TK_NO_ERROR)
    {
        status = ProArraySizeGet ((ProArray) surfaces, &surfaces_num);
        TEST_CALL_REPORT( "ProArraySizeGet()", "ProTestShtmlSurfAct()",
            status, status != PRO_TK_NO_ERROR );
        for (i = 0; i < surfaces_num; i++)
        {
            status = ProTestShtmlSurfAct (surfaces[i],
                PRO_TK_NO_ERROR, (ProAppData)&app_data);
        }
        status = ProArrayFree ((ProArray*)&surfaces);
        TEST_CALL_REPORT( "ProArrayFree()", "ProTestShtmlSurfAct()",
            status, status != PRO_TK_NO_ERROR );
    }

    if (status == PRO_TK_E_NOT_FOUND)
        fprintf(fp, " -- (No surfaces found) --\n");

    fclose(fp);

    return(0);
}

```

```

/*=====*\
    FUNCTION : ProTestShtmtlSurfAct()
    PURPOSE  : General action function for a surface
\*=====*/
ProError ProTestShtmtlSurfAct(
    ProSurface surface,
    ProError instatus,
    ProAppData tmp_app_data)
{
    ProTestSmtlSurf *app_data = (ProTestSmtlSurf *) tmp_app_data;
    ProSmtSurfType  smt_surf_type;
    ProMdl *model;
    int surf_id;
    ProError status;
    FILE *fp;

    fp      = app_data->fp;
    model   = app_data->model;

/*-----*\
    Get the surface id.
\*-----*/
    status = ProSurfaceIdGet(surface, &surf_id);
    TEST_CALL_REPORT("ProSurfaceIdGet()", "ProTestShtmtlSurfAct()",
        status, status != PRO_TK_NO_ERROR);

/*-----*\
    Get the sheetmetal surface type.
\*-----*/
    status = ProSmtSurfaceTypeGet ((ProPart)(*model), surface,
        &smt_surf_type);
    TEST_CALL_REPORT("ProSmtSurfaceTypeGet()", "ProTestShtmtlSurfAct()",
        status, status != PRO_TK_NO_ERROR);

/*-----*\
    Print the id and type of the sheetmetal surface.
\*-----*/
    switch (smt_surf_type)
    {
        case PRO_SMT_SURF_NON_SMT :
            fprintf(fp, " %d \t %s\n", surf_id,
                "PRO_SMT_SURF_NON_SMT - surface created by solid
feature.");
            break;
        case PRO_SMT_SURF_SIDE :
            fprintf(fp, " %d \t %s\n", surf_id,
                "PRO_SMT_SURF_SIDE - side surface.");
            break;
        case PRO_SMT_SURF_FACE :
            fprintf(fp, " %d \t %s\n", surf_id,
                "PRO_SMT_SURF_FACE - face (green) surface.");
    }
}

```

```

        break;
    case PRO_SMT_SURF_OFFSET :
        fprintf(fp, "    %d \t %s\n", surf_id,
            "PRO_SMT_SURF_OFFSET - offset (white) surface.");
        break;
    default ;;
}

return(PRO_TK_NO_ERROR);
}

/*=====*\
    FUNCTION : ProTestSheetmetalSurfs()
    PURPOSE  : Command Traversal for listing of all surfaces in a
                sheetmetal model and evaluation of surface types.
\*=====*/
int ProTestSheetmetalSurfs(
    ProMdl *model,
    int action)
{
    ProError status;
    ProMdlData mdata;
    ProMode curr_mode;
    ProUtilCname type;

/*-----*\
    Find out the current mode (works only in the sheetmetal mode)
\*-----*/
    status = ProModeCurrentGet(&curr_mode);
    TEST_CALL_REPORT("ProModeCurrentGet()", "ProTestSheetmetalSurfs()",
        status, status != PRO_TK_NO_ERROR);

    if (curr_mode != PRO_MODE_SHEET_METAL)
        return(-1);

/*-----*\
    Find out the model type
\*-----*/
    status = ProMdlDataGet(*model, &mdata);
    TEST_CALL_REPORT("ProMdlDataGet()", "ProTestSheetmetalSurfs()",
        status, status != PRO_TK_NO_ERROR);
    ProWstringToString(type, mdata.type);

/*-----*\
    Visit all surfaces in the sheetmetal model.
\*-----*/
    if(!ProUtilStrcmp(type, "PRT"))
        ProTestSheetmetalTraverse(model);

    return(0);
}

```

Bend Tables and Dimensions

Sheet metal bend table functions support reading in or removing bent table data for a sheet metal part or feature in the part.

Sheet metal dimension functions find or set whether or not developed length dimensions are driven.

Functions Introduced:

- **ProSmtPartBendtableApply()**
- **ProSmtPartBendtableRemove()**
- **ProSmtFeatureBendtableApply()**
- **ProSmtFeatureBendtableRemove()**
- **ProSmtFeatureDevldimsGet()**
- **ProSmtDevldimIsDriven()**
- **ProSmtDevldimDrivenSet()**

The function **ProSmtPartBendtableApply()** applies the specified bent table to the sheet metal part, and then regenerates the part. The input argument *from_file* specifies whether the bend table is to be applied from memory or from the specified file.

The function **ProSmtPartBendtableRemove()** removes the specified bend table from the sheet metal part, and then regenerates the part using the Y Factor.

The function **ProSmtFeatureBendtableApply()** applies the specified bent table to the sheet metal part feature, and then regenerates the part. The input argument *from_file* specifies whether the bend table is to be applied from memory or from the specified file.

The function **ProSmtFeatureBendtableRemove()** sets a sheet metal feature to use the part bend table instead of the feature bend table, and then regenerates the part.

The function **ProSmtFeatureDevldimsGet()** returns the developed length dimensions for the specified sheet metal bend or wall feature. It also returns the surfaces whose developed length these dimensions define.

The function **ProSmtDevldimIsDriven()** specifies whether a developed length dimension is driven or not. Use the function **ProSmtDevldimDrivenSet()** to set a developed length dimension to driven.

Sheetmetal Flat Wall Feature

A flat wall is a planar section of a sheet metal part. It can either be a primary wall (the first wall in the design), or a secondary wall (which is dependent on the primary wall). Flat walls can take any flat shape.

Pro/TOOLKIT supports flat walls that are created using the Fill Tool or the Flat Wall tool. Flat walls created using the Fill Tool are unattached and may be the primary wall. Wall created using the Flat Wall tool are secondary walls that are attached to existing wall edges, and may or may not have a bend applied.

Sheetmetal Flat Wall based on the Fill Tool

A sheetmetal flat wall created based on the fill tool shares most of the same elements as the standard fill feature. The element tree should include some of the following sheetmetal specific elements to generate a sheet metal feature:

- `PRO_E_IS_UNATTACHED_WALL`—Has a Boolean value that specifies whether the feature is actually a flat wall.
- `PRO_E_STD_DIRECTION`—Specifies the material creation direction of the sheetmetal flat wall, which allows you to control the thickness of the first sheetmetal wall.
- `PRO_E_STD_SMT_THICKNESS`—Has a double value that specifies the wall thickness. If this feature is not the first wall feature in the part, the thickness value is irrelevant and can be 0.0. The feature inherits the thickness of the first wall feature. This element is not required and cannot be modified if the sheet metal thickness parameter is already assigned in the model.
- `PRO_E_STD_SMT_SWAP_DRV_SIDE`—Specifies whether to swap the sides of the driving and the offset surfaces (the green and white surfaces of the wall).

For details on standard fill features, refer to the section Fill Feature.

Feature Element Tree for the Sheetmetal Flat Wall Feature

The element tree for Sheetmetal Flat Wall feature is documented in the header file *ProSmtFlatWall.h* and has a simple structure. The following figure demonstrates the feature element tree structure:

Figure 52-1: Feature Element tree for Sheetmetal Flat Wall Feature

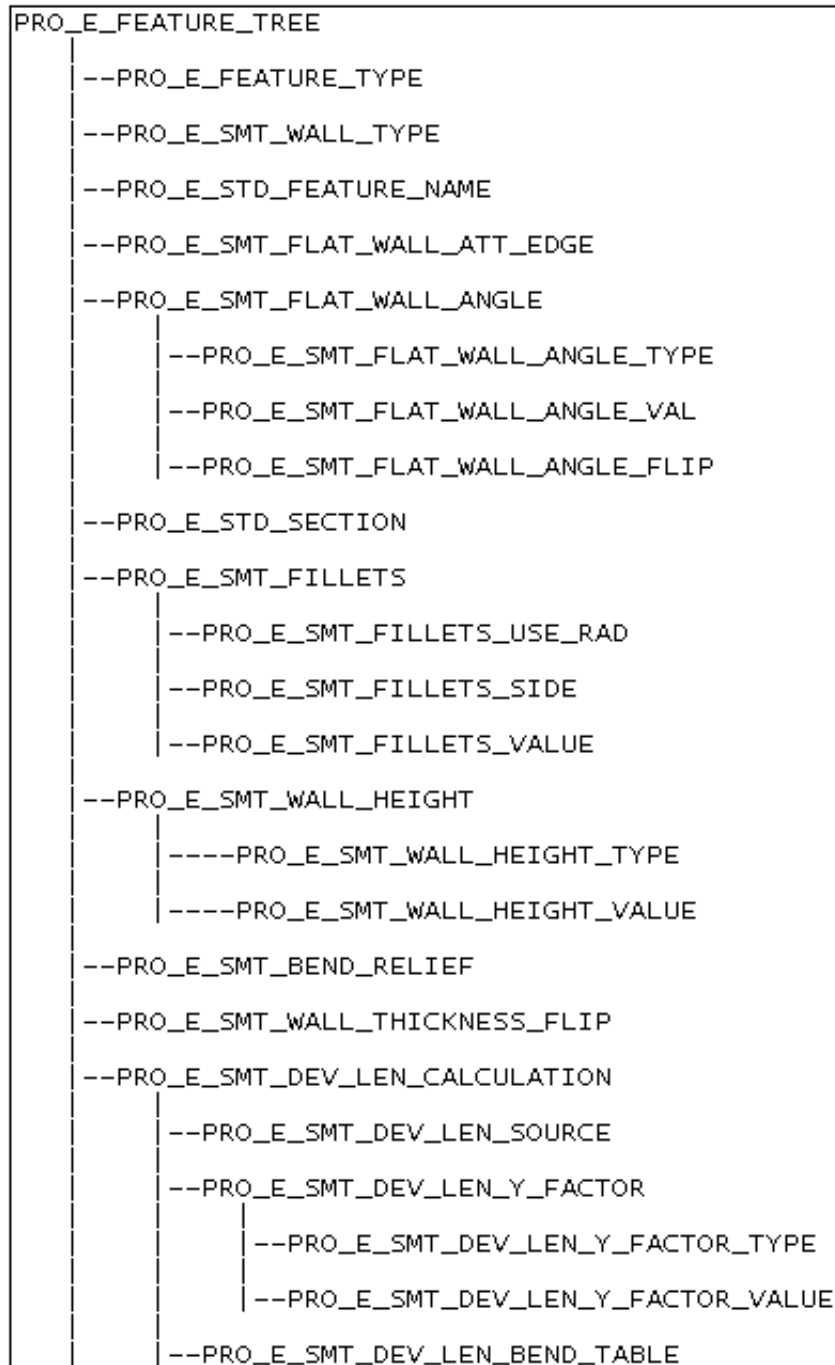
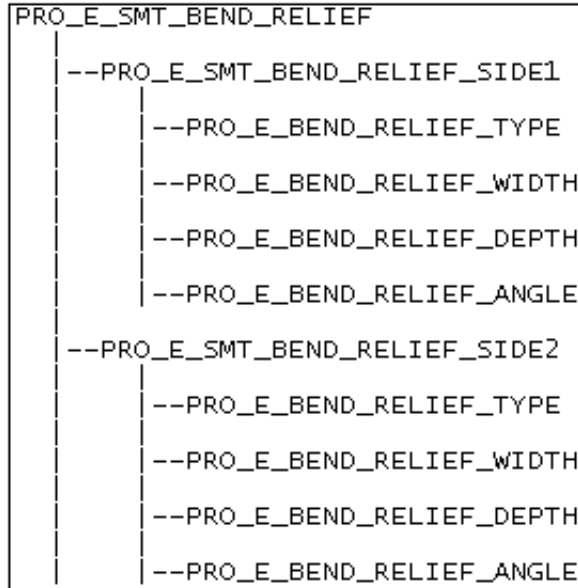


Figure 52-2: PRO_E_SMT_BEND_RELIEF



The feature element tree contains no non-standard element types. The following list details special information about some of the elements in this tree:

- PRO_E_FEATURE_TYPE—Specifies the feature type and should be PRO_FEAT_WALL.
- PRO_E_SMT_WALL_TYPE—Specifies the wall type. For Flat Walls, this should be PRO_SMT_WALL_TYPE_FLAT.
- PRO_E_STD_FEATURE_NAME—Specifies the name of the feature.
- PRO_E_SMT_FLAT_WALL_ATT_EDGE—Specifies the attachment edge.
- PRO_E_SMT_FLAT_WALL_ANGLE—Specifies the bend angle and consists of the following elements:
 - PRO_E_SMT_FLAT_WALL_ANGLE_TYPE specifies the angle type and is of the following types:
 - PRO_BND_ANGLE_VALUE—uses an indicated value.
 - PRO_BND_ANGLE_BY_PARAM—uses the default value of the sheet metal parameter "SMT_DFLT_BEND_ANGLE".

- PRO_BND_ANGLE_FLAT—uses no angle for the wall.
- PRO_E_SMT_FLAT_WALL_ANGLE_VAL—specifies the angle value.
- PRO_E_SMT_FLAT_WALL_ANGLE_FLIP—indicates whether or not to reverse the angle direction.
- PRO_E_STD_SECTION—Specifies the wall section. Wall sections can be standard or user-defined. Standard wall sections are stored in the location <Pro/ENGINEER Loadpoint>/text/smt. These standard sections can be retrieved; their dimensions modified, and can be added directly into the PRO_E_STD_SECTION element tree as the PRO_E_SKETCHER element. This does not require definition of a sketch plane or viewing direction, and it does not require an incomplete feature to be created as is described in the chapter Element Trees: Sketched Features. When standard sections are used to create the wall, the Pro/ENGINEER user interface will show the correct type of section in the drop down menu (for example: "Rectangle", "Trapezoid", "L", "T").

If a user-defined section is to be used to create a flat wall, it must conform to the following restrictions:

- It must be a 2D section
- It must not be named the same name as one of the default section types from the Pro/ENGINEER loadpoint.
- It must contain a horizontal centerline and 2 coordinate systems. The centerline represents the alignment with the attachment edge, and the coordinate systems represent the edge endpoints.
- The horizontal dimension must be specified.

When user-defined sections are used to create the wall, Pro/ENGINEER automatically creates necessary sketching planes for the section during creation. Therefore the section may be assigned directly into the PRO_E_SKETCHER element without defining the sketch plane and without creating the feature as incomplete. After the feature has been created, the 3D section can be extracted from the feature element tree and the section can be modified to include references to other geometric entities in the sheet metal part.

- PRO_E_SMT_FILLETS—Specifies the bend properties of the sheet metal wall:
 - PRO_E_SMT_FILLETS_USE_RAD—If true, a bend is applied, if false, no bend is used.

- PRO_E_SMT_FILLETS_SIDE—Specifies fillet side and has the following permitted values:
 - PRO_BEND_RAD_OUTSIDE—apply the bend radius to the outside of the bend.
 - PRO_BEND_RAD_INSIDE—apply the bend radius to the inside of the bend.
- PRO_E_SMT_FILLETS_VALUE—Specifies bend radius.
- PRO_E_SMT_WALL_HEIGHT—Specifies the height of the attachment wall. It has the following elements:
 - PRO_E_SMT_WALL_HEIGHT_TYPE specifies the manner in which the newly created wall feature attaches to the attachment edge. This element takes the following values:
 - PRO_SMT_WALL_HEIGHT_AUTO—Specifies that the wall feature attaches to the attachment edge by trimming the height of the attachment wall automatically.
 - PRO_SMT_WALL_HEIGHT_VALUE—Specifies that the wall feature attaches to the attachment edge by trimming the height of the attachment wall by a specified value.
 - PRO_SMT_WALL_HEIGHT_APP_BEND—Specifies that the wall feature appends to the attachment edge without trimming the height of the attachment wall.
 - PRO_E_SMT_WALL_HEIGHT_VALUE specifies the value of the height of the attachment wall.
- PRO_E_SMT_BEND_RELIEF—Specifies bend relief at the edges of the new wall feature. The relief can be specific differently on each side of the bend:

PRO_E_SMT_BEND_RELIEF_SIDE1—Specifies the first bend relief:

- PRO_E_BEND_RELIEF_TYPE specifies relief type and has the following values:
 - PRO_BEND_RLF_NONE specifies attachment of the wall using no relief.
 - PRO_BEND_RLF_RIP specifies ripping of the material at each attachment point.
 - PRO_BEND_RLF_STRETCH specifies stretching of the material for bend relief at wall attachment point.

- PRO_BEND_RLF_RECTANGULAR specifies adding a rectangular relief at each attachment point.
- PRO_BEND_RLF_OBROUND specifies adding an obround relief at each attachment point.
- PRO_E_BEND_RELIEF_WIDTH specifies the relief width (for rectangular and obround relief).
- PRO_E_BEND_RELIEF_DEPTH specifies relief depth (for rectangular and obround relief).
- PRO_E_BEND_RELIEF_ANGLE specifies relief angle (for stretch relief).

PRO_E_SMT_BEND_RELIEF_SIDE2— Includes an identical subtree for the relief applied to the second side of the wall.

- PRO_E_SMT_WALL_THICKNESS_FLIP—Indicates whether or not to flip the thickness direction of the new wall.
- PRO_E_SMT_DEV_LEN_CALCULATION—Specifies the method used to calculate the Developed Length dimensions for bends.

The Element Subtree for PRO_E_SMT_DEV_LEN_CALCULATION

- PRO_E_SMT_DEV_LEN_SOURCE— Specifies the development length source:
 - PRO_DVL_SRC_PART_YF_AND_BTAB—uses part Y-factor and applied bend table.
 - PRO_DVL_SRC_PART_YF_ONLY—uses the part Y-factor.
 - PRO_DVL_SRC_FEAT_YF_AND_BTAB—uses the feature specific Y-factor and bend table.
 - PRO_DVL_SRC_FEAT_BTAB_ONLY—uses the feature specific bend table.
 - PRO_DVL_SRC_FEAT_YF_ONLY—uses the feature specific y-factor.
- PRO_E_SMT_DEV_LEN_Y_FACTOR—Specifies the feature Y-factor and has the following elements:
 - PRO_E_SMT_DEV_LEN_Y_FACTOR_TYPE—Specifies the types of Y-factor and has the following values:
 - PRO_FACTOR_Y
 - PRO_FACTOR_K

- **PRO_E_SMT_DEV_LEN_Y_FACTOR_VALUE**—Specifies the value of Y- or K- factor.
 - **PRO_E_SMT_DEV_LEN_BEND_TABLE**—Specifies the development length bend table using the index of the bend table as loaded and stored in this model.
- Note:** Bend allowance is a method used to calculate the developed length of flat sheet metal required to make a bend of a specific radius and angle. The calculation accounts for the thickness of the sheet metal, bend radii, bend angles, and other material properties such as Y- and K-factors. Developed length fluctuates with different material types and thickness, and the bend table accounts for those variations.
- Note:** Y- and K-factors are part constants defined by the location of the sheet metal material's neutral bend line. The neutral bend line position is based on a numeric reference for the type of sheet metal material used in your design. The numeric references range from 0 to 1, with the lower numbers representing softer material. Both the Y- and K-factors are integral elements in calculating the developed length (the length of flat sheet metal required to make a bend of a specific radius and angle) in your design.

Creating a Sheetmetal Flat Wall Feature

Function Introduced

- **ProFeatureCreate()**

Use the function **ProFeatureCreate()** to create a Sheetmetal Flat Wall Feature based on element tree input. For more information about **ProFeatureCreate()**, refer to the section “Overview of Feature Creation” in the “Element Trees: Principles of Feature Creation” chapter.

Redefining a Sheetmetal Flat Wall Feature

Function Introduced

- **ProFeatureRedefine()**

Use the function **ProFeatureRedefine()** to redefine a Sheetmetal Flat Wall Feature based on the changes made in the element tree. For more information about **ProFeatureRedefine()**, refer to the section “Feature Redefine” in the “Element Trees: Principles of Feature Creation” chapter.

Accessing a Sheetmetal Flat Wall Feature

Function Introduced

- **ProFeatureElemtreeExtract()**

Use the function **ProFeatureElemtreeExtract()** to create a feature element tree that describes the contents of a Sheetmetal Flat Wall Feature and to retrieve the element tree description of a Sheetmetal Flat Wall Feature. For more information about **ProFeatureElemtreeExtract()**, refer to the section “Feature Inquiry” in the “Element Trees: Principles of Feature Creation” chapter.

Example 2: Creation of a Rectangular Flat Wall using a preselected edge

This example demonstrates creation of a rectangular flat wall using a preselected edge.

```
/*=====*\
FILE      : UgSmtFlatWallCreate.c
PURPOSE   : Sheetmetal flat wall feature creation example
\*=====*/

#include <ProToolkit.h>
#include <ProFeature.h>
#include <ProSelbuffer.h>
#include <ProReference.h>
#include <ProUtil.h>
#include <ProSmtFlatWall.h>
#include <ProSheetmetal.h>
#include <ProMenuBar.h>
#include <ProPopupMenu.h>
#include <ProMdl.h>
#include <PTAppIsUnicodeUtils.h>
#include <ProEdge.h>

/*=====*\
FUNCTION: UserSmtFlatWallCreate
PURPOSE:  Create sheetmetal flat wall features.
\*=====*/
int UserSmtFlatWallCreate (ProReference edge_ref,
    double bend_radius,
    ProSection section)
{
    ProElement pro_e_feature_tree;
    ProElement pro_e_feature_type;
    ProElement pro_e_smt_wall_type;
    ProElement pro_e_std_feature_name;
```

```

ProElement pro_e_smt_flat_wall_att_edge;
ProElement pro_e_smt_flat_wall_angle;
ProElement pro_e_smt_flat_wall_angle_type;
ProElement pro_e_smt_flat_wall_angle_val;
ProElement pro_e_smt_flat_wall_angle_flip;
ProElement pro_e_std_section;
ProElement pro_e_sketcher;
ProElement pro_e_smt_fillet;
ProElement pro_e_smt_fillet_use_rad;
ProElement pro_e_smt_fillet_side;
ProElement pro_e_smt_fillet_value;
ProElement pro_e_smt_wall_height;
ProElement pro_e_smt_wall_height_type;
ProElement pro_e_smt_wall_height_value;
ProElement pro_e_smt_bend_relief;
ProElement pro_e_smt_bend_relief_side1;
ProElement pro_e_bend_relief_type;
ProElement pro_e_bend_relief_width;
ProElement pro_e_bend_relief_depth;
ProElement pro_e_bend_relief_angle;
ProElement pro_e_smt_bend_relief_side2;
ProElement pro_e_smt_wall_thickness_flip;
ProElement pro_e_smt_dev_len_calculation;
ProElement pro_e_smt_dev_len_source;
ProElement pro_e_smt_dev_len_y_factor;
ProElement pro_e_smt_dev_len_y_factor_value;
ProElement pro_e_smt_dev_len_y_factor_type;
ProElement pro_e_smt_dev_len_bend_table;
ProMdl model;
ProModelitem model_item;
ProFeatureCreateOptions opts [1];
ProSelection model_sel;
ProFeature feature;
ProErrorlist errors;
ProError status;
ProSelection feat_sel;

/*-----*\
Processing Element PRO_E_FEATURE_TREE
/*-----*/
    status = ProElementAlloc ( PRO_E_FEATURE_TREE, &pro_e_feature_tree );

/*-----*\
Processing Element PRO_E_FEATURE_TYPE
/*-----*/
    status = ProElementAlloc ( PRO_E_FEATURE_TYPE, &pro_e_feature_type );
    status = ProElementIntegerSet ( pro_e_feature_type, PRO_FEAT_WALL );
    status = ProElementtreeElementAdd ( pro_e_feature_tree, NULL,
        pro_e_feature_type );

```



```

/*-----*\
Processing Element PRO_E_SMT_WALL_TYPE
\*-----*/
status = ProElementAlloc ( PRO_E_SMT_WALL_TYPE, &pro_e_smt_wall_type );
status = ProElementIntegerSet ( pro_e_smt_wall_type,
PRO_SMT_WALL_TYPE_FLAT );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_smt_wall_type );

/*-----*\
Processing Element PRO_E_SMT_FLAT_WALL_ATT_EDGE
\*-----*/
status = ProElementAlloc ( PRO_E_SMT_FLAT_WALL_ATT_EDGE,
&pro_e_smt_flat_wall_att_edge );
status = ProElementReferenceSet( pro_e_smt_flat_wall_att_edge, edge_ref
);
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_smt_flat_wall_att_edge );

/*-----*\
Processing Element PRO_E_SMT_FLAT_WALL_ANGLE
\*-----*/
status = ProElementAlloc ( PRO_E_SMT_FLAT_WALL_ANGLE,
&pro_e_smt_flat_wall_angle );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_smt_flat_wall_angle );

/*-----*\
Processing Element PRO_E_SMT_FLAT_WALL_ANGLE_TYPE
\*-----*/
status = ProElementAlloc ( PRO_E_SMT_FLAT_WALL_ANGLE_TYPE,
&pro_e_smt_flat_wall_angle_type );
status = ProElementIntegerSet ( pro_e_smt_flat_wall_angle_type,
PRO_BND_ANGLE_VALUE);
status = ProElemtreeElementAdd ( pro_e_smt_flat_wall_angle, NULL,
pro_e_smt_flat_wall_angle_type );

/*-----*\
Processing Element PRO_E_SMT_FLAT_WALL_ANGLE_VAL
\*-----*/
status = ProElementAlloc ( PRO_E_SMT_FLAT_WALL_ANGLE_VAL,
&pro_e_smt_flat_wall_angle_val );
status = ProElementDoubleSet ( pro_e_smt_flat_wall_angle_val, 90.000000
);
status = ProElemtreeElementAdd ( pro_e_smt_flat_wall_angle, NULL,
pro_e_smt_flat_wall_angle_val );

/*-----*\
Processing Element PRO_E_SMT_FLAT_WALL_ANGLE_FLIP
\*-----*/
status = ProElementAlloc ( PRO_E_SMT_FLAT_WALL_ANGLE_FLIP,

```

```

        &pro_e_smt_flat_wall_angle_flip );
status = ProElementIntegerSet ( pro_e_smt_flat_wall_angle_flip,
                                PRO_B_FALSE);
status = ProElemtreeElementAdd ( pro_e_smt_flat_wall_angle, NULL,
                                pro_e_smt_flat_wall_angle_flip );

/*-----*\
Processing Element PRO_E_STD_SECTION
\*-----*/
status = ProElementAlloc ( PRO_E_STD_SECTION, &pro_e_std_section );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_std_section );

/*-----*\
Processing Element PRO_E_SKETCHER
\*-----*/
status = ProElementAlloc ( PRO_E_SKETCHER, &pro_e_sketcher );
status = ProElementSpecialvalueSet ( pro_e_sketcher, section );
status = ProElemtreeElementAdd ( pro_e_std_section, NULL,
                                pro_e_sketcher );

/*-----*\
Processing Element PRO_E_SMT_FILLETS
\*-----*/
status = ProElementAlloc ( PRO_E_SMT_FILLETS, &pro_e_smt_filletts );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_smt_filletts );

/*-----*\
Processing Element PRO_E_SMT_FILLETS_USE_RAD
\*-----*/
status = ProElementAlloc ( PRO_E_SMT_FILLETS_USE_RAD,
                            &pro_e_smt_filletts_use_rad );
status = ProElementIntegerSet ( pro_e_smt_filletts_use_rad, PRO_B_TRUE);
status = ProElemtreeElementAdd ( pro_e_smt_filletts, NULL,
                                pro_e_smt_filletts_use_rad );

/*-----*\
Processing Element PRO_E_SMT_FILLETS_SIDE
\*-----*/
status = ProElementAlloc ( PRO_E_SMT_FILLETS_SIDE,
                            &pro_e_smt_filletts_side );
status = ProElementIntegerSet ( pro_e_smt_filletts_side,
                                PRO_BEND_RAD_INSIDE );
status = ProElemtreeElementAdd ( pro_e_smt_filletts, NULL,
                                pro_e_smt_filletts_side );

/*-----*\
Processing Element PRO_E_SMT_FILLETS_VALUE
\*-----*/
status = ProElementAlloc ( PRO_E_SMT_FILLETS_VALUE,

```

```

        &pro_e_smt_filletlets_value );
    status = ProElementDoubleSet ( pro_e_smt_filletlets_value, bend_radius );
    status = ProElemtreeElementAdd ( pro_e_smt_filletlets, NULL,
        pro_e_smt_filletlets_value );

/*-----*\
Processing Element PRO_E_SMT_WALL_HEIGHT
\*-----*/
    status = ProElementAlloc ( PRO_E_SMT_WALL_HEIGHT, &pro_e_smt_wall_height
);
    status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
        pro_e_smt_wall_height );

/*-----*\
Processing Element PRO_E_SMT_WALL_HEIGHT_TYPE
\*-----*/
    status = ProElementAlloc ( PRO_E_SMT_WALL_HEIGHT_TYPE,
        &pro_e_smt_wall_height_type );
    status = ProElementIntegerSet ( pro_e_smt_wall_height_type,
        PRO_SMT_WALL_HEIGHT_NONE );
    status = ProElemtreeElementAdd ( pro_e_smt_wall_height, NULL,
        pro_e_smt_wall_height_type );

/*-----*\
Processing Element PRO_E_SMT_BEND_RELIEF
\*-----*/
    status = ProElementAlloc ( PRO_E_SMT_BEND_RELIEF, &pro_e_smt_bend_relief
);
    status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
        pro_e_smt_bend_relief );

/*-----*\
Processing Element PRO_E_SMT_BEND_RELIEF_SIDE1
\*-----*/
    status = ProElementAlloc ( PRO_E_SMT_BEND_RELIEF_SIDE1,
        &pro_e_smt_bend_relief_side1 );
    status = ProElemtreeElementAdd ( pro_e_smt_bend_relief, NULL,
        pro_e_smt_bend_relief_side1 );

/*-----*\
Processing Element PRO_E_BEND_RELIEF_TYPE
\*-----*/
    status = ProElementAlloc ( PRO_E_BEND_RELIEF_TYPE,
        &pro_e_bend_relief_type );
    status = ProElementIntegerSet ( pro_e_bend_relief_type, PRO_BEND_RLF_RIP
);
    status = ProElemtreeElementAdd ( pro_e_smt_bend_relief_side1, NULL,
        pro_e_bend_relief_type );

/*-----*\
Processing Element PRO_E_SMT_BEND_RELIEF_SIDE2

```

```

\*-----*/
status = ProElementAlloc ( PRO_E_SMT_BEND_RELIEF_SIDE2,
    &pro_e_smt_bend_relief_side2 );
status = ProElementtreeElementAdd ( pro_e_smt_bend_relief, NULL,
    pro_e_smt_bend_relief_side2 );

/*-----*\
Processing Element PRO_E_BEND_RELIEF_TYPE
\*-----*/
status = ProElementAlloc ( PRO_E_BEND_RELIEF_TYPE,
    &pro_e_bend_relief_type );
status = ProElementIntegerSet ( pro_e_bend_relief_type, PRO_BEND_RLF_RIP
);
status = ProElementtreeElementAdd ( pro_e_smt_bend_relief_side2, NULL,
    pro_e_bend_relief_type );

/*-----*\
Processing Element PRO_E_SMT_WALL_THICKNESS_FLIP
\*-----*/
status = ProElementAlloc ( PRO_E_SMT_WALL_THICKNESS_FLIP,
    &pro_e_smt_wall_thickness_flip );
status = ProElementIntegerSet ( pro_e_smt_wall_thickness_flip,
    PRO_B_FALSE );
status = ProElementtreeElementAdd ( pro_e_feature_tree, NULL,
    pro_e_smt_wall_thickness_flip );

/*-----*\
Processing Element PRO_E_SMT_DEV_LEN_CALCULATION
\*-----*/
status = ProElementAlloc ( PRO_E_SMT_DEV_LEN_CALCULATION,
    &pro_e_smt_dev_len_calculation );
status = ProElementtreeElementAdd ( pro_e_feature_tree, NULL,
    pro_e_smt_dev_len_calculation );

/*-----*\
Processing Element PRO_E_SMT_DEV_LEN_SOURCE
\*-----*/
status = ProElementAlloc ( PRO_E_SMT_DEV_LEN_SOURCE,
    &pro_e_smt_dev_len_source );
status = ProElementIntegerSet ( pro_e_smt_dev_len_source,
    PRO_DVL_SRC_FEAT_YF_ONLY );
status = ProElementtreeElementAdd ( pro_e_smt_dev_len_calculation, NULL,
    pro_e_smt_dev_len_source );

/*-----*\
Processing Element PRO_E_SMT_DEV_LEN_Y_FACTOR
\*-----*/
status = ProElementAlloc ( PRO_E_SMT_DEV_LEN_Y_FACTOR,
    &pro_e_smt_dev_len_y_factor );
status = ProElementtreeElementAdd ( pro_e_smt_dev_len_calculation, NULL,
    pro_e_smt_dev_len_y_factor );

```

```

/*-----*\
Processing Element PRO_E_SMT_DEV_LEN_Y_FACTOR_VALUE
\*-----*/
status = ProElementAlloc ( PRO_E_SMT_DEV_LEN_Y_FACTOR_VALUE,
    &pro_e_smt_dev_len_y_factor_value );
status = ProElementDoubleSet ( pro_e_smt_dev_len_y_factor_value,
    0.320000 );
status = ProElementtreeElementAdd ( pro_e_smt_dev_len_y_factor, NULL,
    pro_e_smt_dev_len_y_factor_value );

/*-----*\
Create the feature using the element tree
\*-----*/
status = ProMdlCurrentGet (&model);
status = ProMdlToModelitem( model, &model_item );
status = ProSelectionAlloc (NULL, &model_item, &model_sel);

opts[0] = PRO_FEAT_CR_NO_OPTS;
status = ProFeatureCreate (model_sel, pro_e_feature_tree, opts, 1,
    &feature, &errors);

if (status == PRO_TK_NO_ERROR)
{
    ProSelectionAlloc (NULL, &feature, &feat_sel);
    ProSelbufferClear ();
    ProSelbufferSelectionAdd (feat_sel);
    ProSelectionFree (&feat_sel);
}

return (PRO_TK_NO_ERROR);
}

/*=====*\
FUNCTION: UserSmtFlatWallCreate_OA_CreateISec
PURPOSE: Create sheetmetal flat wall features using Object/Action
         selection and Pro/E's predefined rectangular section.
\*=====*/
int UserSmtFlatWallCreate_OA_CreateRectSec ()
{
    ProSelection* sel;
    ProReference reference;
    ProPart part;
    ProDimension thickness_dim;
    double thickness;
    ProSection rect_wall_sec;
    ProPath w_mdldpath;
    ProCharPath mdlpath, secpath;
    char* prodir;

```

```

/*-----*\
Extract the current selected edge or edge chain as the wall attachment
\*-----*/
    status = ProSelbufferSelectionsGet (&sel);

    status = ProSelectionToReference ( sel[0], &reference );

/*-----*\
Extract the wall thickness to use for the bend radius
\*-----*/
    status = ProMdlCurrentGet ((ProMdl*)&part);
    status = ProSmtPartThicknessGet(part, &thickness_dim);
    status = ProDimensionValueGet (&thickness_dim, &thickness);

/*-----*\
Retrieve the Pro/ENGINEER default section for rectangular flat walls
\*-----*/
    prodir = getenv ("PRO_DIRECTORY");
    if (prodir == NULL)
return PRO_TK_INVALID_DIR;

    ProTKSprintf (secpath, "%s/text/smt/", prodir);
    ProTKSprintf (mdlpath, "%srect_flat_wall.sec", secpath);
    ProStringToWstring (w_mdlpath, mdlpath);
    status = ProMdlLoad (w_mdlpath, PRO_VALUE_UNUSED, PRO_B_FALSE,
        (ProMdl*)&rect_wall_sec);

/*-----*\
Create the feature with the extracted information
\*-----*/
    UserSmtFlatWallCreate (reference, thickness, rect_wall_sec);

    ProReferenceFree (reference);

    return (PRO_TK_NO_ERROR);
}

/*=====*\
FUNCTION: UserSectionOpen
PURPOSE: Retrieve a section using a File Open dialog.
\*=====*/
ProError UserSectionOpen (ProSection* sec)
{
    ProName label;
    ProLine filter;
    ProPath* shortcut_path_arr;
    ProName* shortcut_name;
    ProPath path;

/*-----*\
Prompt to select a section using the Open dialog box

```

```

\*-----*/
ProStringToWstring (label, "Select section");
ProStringToWstring (filter, "*.sec");
ProArrayAlloc (0, sizeof (ProPath), 1, (ProArray*) &shortcut_path_arr);
ProArrayAlloc (0, sizeof (ProName), 1, (ProArray*) &shortcut_name);

status = ProFileOpen (label, filter, shortcut_path_arr, shortcut_name,
NULL, NULL, path);

if (status == PRO_TK_NO_ERROR)
{
\*-----*\
Retrieve the selected section
\*-----*/
status = ProMdlLoad (path, PRO_VALUE_UNUSED, PRO_B_FALSE,
(ProMdl*)sec);
}

ProArrayFree ((ProArray*)&shortcut_path_arr);
ProArrayFree ((ProArray*)&shortcut_name);

return (status);
}

\*=====*\
FUNCTION: UserSmtFlatWallCreate_OA_CreateUserDefSec
PURPOSE: Create sheetmetal flat wall features using Object/Action
selection and a user-selected 2D section.
\*=====*/
int UserSmtFlatWallCreate_OA_CreateUserDefSec ()
{
ProSelection* sel;
ProReference reference;
ProPart part;
ProDimension thickness_dim;
double thickness;
ProSection userdef_sec;
ProPath w_mdldpath;
ProCharPath mdldpath, secpath;
char* prodir;

\*-----*\
Prompt the user to select a section for the flat wall
\*-----*/
status = UserSectionOpen (&userdef_sec);

if (status != PRO_TK_NO_ERROR)
return (status);

\*-----*\
Extract the current selected edge or edge chain as the wall attachment

```

```

\*-----*/
    status = ProSelbufferSelectionsGet (&sel);

    status = ProSelectionToReference ( sel[0], &reference );

/*-----*\
Extract the wall thickness to use for the bend radius
\*-----*/
    status = ProMdlCurrentGet ((ProMdl*)&part);
    status = ProSmtPartThicknessGet(part, &thickness_dim);
    status = ProDimensionValueGet (&thickness_dim, &thickness);

/*-----*\
Create the feature with the extracted information
\*-----*/
    UserSmtFlatWallCreate (reference, thickness, userdef_sec);

    ProReferenceFree (reference);
    ProMdlErase (userdef_sec);

    return (PRO_TK_NO_ERROR);
}

/*=====*\
FUNCTION: UserEdgeIsSmtFaceEdge
PURPOSE: Returns TRUE if the edge borders a sheetmetal face.
\*=====*/
ProBoolean UserEdgeIsSmtFaceEdge (ProGeomitem* edge)
{
    ProEdge edge_ptr;
    ProEdge edges [2];
    ProSurface surfs [2];
    ProSmtSurfType srf_type, srf_type2;

/*-----*\
Obtain the neighboring surfaces for the given edge
\*-----*/
    ProGeomitemToEdge (edge, &edge_ptr);

    ProEdgeNeighborsGet (edge_ptr, &edges[0], &edges[1],
        &surfs [0], &surfs[1]);

/*-----*\
Test the surfaces
\*-----*/
    ProSmtSurfaceTypeGet (edge->owner, surfs [0], &srf_type);
    ProSmtSurfaceTypeGet (edge->owner, surfs [1], &srf_type2);

    if (srf_type == PRO_SMT_SURF_FACE ||
        srf_type2 == PRO_SMT_SURF_FACE ||
        srf_type == PRO_SMT_SURF_OFFSET ||

```



```

        srf_type2 == PRO_SMT_SURF_OFFSET)
    return PRO_B_TRUE;
else
    return PRO_B_FALSE;
}

#define OA 1 /* Standard OA in the menu system, typically gray out
             the button when not usable*/
#define PM 2 /* OA in a popup menu, typically remove the button when
             not usable */

/*=====*\
FUNCTION: UserSmtFlatWallCreate_TestLow
PURPOSE: Access function for the button for smt flat wall feature
creation
\*=====*/
static uiCmdAccessState UserSmtFlatWallCreate_TestLow (ProSelection*
                                                       sels, int mode)
{
    uiCmdAccessState access_result;
    ProBoolean should_free = PRO_B_FALSE;
    ProError status;
    ProMdl mdl;
    ProMdlType type;
    ProMdlsubtype subtype;
    ProReference reference;
    ProModelitem item;
    ProEdge edge;
    ProEnttype ent_type;
    int size;

/*-----*\
    Set the default return if the button is unusable.
\*-----*/
    if (mode == OA)
        access_result = ACCESS_UNAVAILABLE;
    else
        access_result = ACCESS_REMOVE;

    status = ProMdlCurrentGet (&mdl);

    if (status != PRO_TK_NO_ERROR)
        return (access_result);

    status = ProMdlTypeGet (mdl, &type);

    status = ProMdlSubtypeGet (mdl, &subtype);

    if (type != PRO_MDL_PART || subtype != PROMDLSTYPE_PART_SHEETMETAL)
        return (access_result);

```

```

/*-----*\
  If called without selections, extract the current selections from
  the buffer.
/*-----*/
  if (sels == NULL)
  {
    status = ProSelbufferSelectionsGet (&sels);

    if (status != PRO_TK_NO_ERROR)
return access_result;

    if (sels == NULL)
return access_result;

    should_free = PRO_B_TRUE;
  }

/*-----*\
  This command allows only one selection.
/*-----*/
  status = ProArraySizeGet (sels, &size);

  if (status != PRO_TK_NO_ERROR)
    return access_result;

  if (size == 1)
  {
/*-----*\
Extract the reference edge selected by the user
/*-----*/
    status = ProSelectionToReference (sels [0], &reference);
    ProTKPrintf ("Status of ProSelectionCollectionGet: %d\n\n", status);

    if (status == PRO_TK_NO_ERROR)
    {
/*-----*\
Check the reference to ensure that the reference is an edge
which borders a sheetmetal face.
/*-----*/
      ProReferenceTypeGet (reference, &item.type);

      if (item.type == PRO_EDGE)
      {

        ProReferenceOwnerGet (reference, &item.owner);
        ProReferenceIdGet (reference, &item.id);

        ProReferenceFree (reference);

        if (UserEdgeIsSmtFaceEdge (&item))
      {

```

```

/*-----*\
Flat wall attachment edge must be straight
\*-----*/
    ProGeomitemToEdge (&item, &edge);

    ProEdgeTypeGet (edge, &ent_type);

    if (ent_type == PRO_ENT_LINE)
    {
/*-----*\
If reference is OK, the button should be active.
\*-----*/
        access_result = ACCESS_AVAILABLE;
    }
}
    }
}

    if (should_free)
        ProSelectionarrayFree (sels);

    return access_result;
}

/*=====*\
FUNCTION: UserSmtFlatWallCreate_TestMdlTree
PURPOSE: Access function for the model tree popup menu button.
\*=====*/
uiCmdAccessState UserSmtFlatWallCreate_TestMdlTree (uiCmdAccessMode mode)
{
    return UserSmtFlatWallCreate_TestLow (NULL, PM);
}

/*=====*\
FUNCTION: UserSmtFlatWallCreate_TestPM
PURPOSE: Access function for the graphics window popup menu button.
\*=====*/
uiCmdAccessState UserSmtFlatWallCreate_TestPM (uiCmdCmdId id,
                                                ProAppData data, ProSelection* sels)
{
    return UserSmtFlatWallCreate_TestLow (sels, PM);
}

```

Sheetmetal Flange Wall Feature

Flange wall features may be either swept or extruded.

A swept flange wall follows the trajectory formed by the chain of tangent attachment edges. You can sketch a cross section along the attachment edge and the wall sweeps along that edge. The attachment edge need not be linear and the adjacent surface need not have to be planar.

An extruded flange wall extends from one linear edge into space. You can sketch the side section of the wall and project it to a certain length in both directions.

Feature Element Tree for the Sheetmetal Flange Wall Feature

The element tree for Sheetmetal Flange Wall feature is documented in the header file *ProSmtFlangeWall.h*, and has a simple structure. The following figure demonstrates the feature element tree structure:

Figure 52-3: Feature Element tree for Sheetmetal Flange Wall Feature

```

PRO_E_FEATURE_TREE
|--PRO_E_FEATURE_TYPE
|--PRO_E_SMT_WALL_TYPE
|--PRO_E_STD_FEATURE_NAME
|--PRO_E_SMT_FLANGE_TYPE
|--PRO_E_STD_CURVE_COLLECTION_APPL
|--PRO_E_SMT_FLANGE_TRAJ_CRV_NORM
|--PRO_E_STD_SECTION
|--PRO_E_SMT_WALL_SHARPS_TO_BENDS
|--PRO_E_SMT_FLANGE_SEC_FLIP
|--PRO_E_SMT_FLANGE_DEPTH
|--PRO_E_SMT_FILLETS
|   |--PRO_E_SMT_FILLETS_USE_RAD
|   |--PRO_E_SMT_FILLETS_SIDE
|   |--PRO_E_SMT_FILLETS_VALUE
|--PRO_E_SMT_WALL_HEIGHT
|   |--PRO_E_SMT_WALL_HEIGHT_TYPE
|   |--PRO_E_SMT_WALL_HEIGHT_VALUE
|--PRO_E_SMT_BEND_RELIEF
|--PRO_E_SMT_WALL_THICKNESS_FLIP
|--PRO_E_SMT_CORNER_RELIEF
|--PRO_E_SMT_MTR_CUTS
|--PRO_E_SMT_AUTO_EXLD_EDGE
|--PRO_E_SMT_CORNERS_ARR
|--PRO_E_SMT_DEV_LEN_CALCULATION
    
```

Figure 52-4: PRO_E_SMT_FLANGE_DEPTH

```
PRO_E_SMT_FLANGE_DEPTH
|--PRO_E_SMT_FLANGE_SIDE_1_DEPTH
|   |--PRO_E_SMT_FLANGE_DEPTH_TYPE
|   |--PRO_E_SMT_FLANGE_DEPTH_OFFSET
|   |--PRO_E_SMT_FLANGE_DEPTH_REF
|--PRO_E_SMT_FLANGE_SIDE_2_DEPTH
|   |--PRO_E_SMT_FLANGE_DEPTH_TYPE
|   |--PRO_E_SMT_FLANGE_DEPTH_OFFSET
|   |--PRO_E_SMT_FLANGE_DEPTH_REF
```

Figure 52-5: PRO_E_SMT_BEND_RELIEF

```
PRO_E_SMT_BEND_RELIEF
|--PRO_E_SMT_BEND_RELIEF_SIDE1
|   |--PRO_E_BEND_RELIEF_TYPE
|   |--PRO_E_BEND_RELIEF_WIDTH
|   |--PRO_E_BEND_RELIEF_DEPTH
|   |--PRO_E_BEND_RELIEF_ANGLE
|--PRO_E_SMT_BEND_RELIEF_SIDE2
|   |--PRO_E_BEND_RELIEF_TYPE
|   |--PRO_E_BEND_RELIEF_WIDTH
|   |--PRO_E_BEND_RELIEF_DEPTH
|   |--PRO_E_BEND_RELIEF_ANGLE
```

Figure 52-6: PRO_E_SMT_CORNER_RELIEF

```

PRO_E_SMT_CORNER_RELIEF
|
|--PRO_E_SMT_CORNER_RELIEF_TYPE
|
|--PRO_E_SMT_CORNER_RELIEF_WIDTH
|   |
|   |--PRO_E_SMT_CORNER_RELIEF_WIDTH_TYPE
|   |
|   |--PRO_E_SMT_CORNER_RELIEF_WIDTH_VAL
|
|--PRO_E_SMT_CORNER_RELIEF_DEPTH
|   |
|   |--PRO_E_SMT_CORNER_RELIEF_DEPTH_TYPE
|   |
|   |--PRO_E_SMT_CORNER_RELIEF_DEPTH_VAL
|

```

Production Applications:
Sheetmetal

Figure 52-7: PRO_E_SMT_MTR_CUTS

```

PRO_E_SMT_MTR_CUTS
|
|--PRO_E_SMT_MTR_CUTS_ADD
|
|--PRO_E_SMT_MTR_CUTS_KEEP_DEF_AREAS
|
|--PRO_E_SMT_MTR_CUTS_WIDTH
|   |
|   |--PRO_E_SMT_MTR_CUTS_WIDTH_TYPE
|   |
|   |--PRO_E_SMT_MTR_CUTS_WIDTH_VAL
|
|--PRO_E_SMT_MTR_CUTS_OFFSET
|   |
|   |--PRO_E_SMT_MTR_CUTS_OFFSET_TYPE
|   |
|   |--PRO_E_SMT_MTR_CUTS_OFFSET_VAL
|

```

Figure 52-8: PRO_E_SMT_CORNERS_ARR

```

PRO_E_SMT_CORNERS_ARR
|
|--PRO_E_SMT_CORNER
|
|   |--PRO_E_SMT_EDGE_RIP
|   |
|   |   |--PRO_E_SMT_EDGE_RIP_TYPE
|   |   |
|   |   |--PRO_E_SMT_EDGE_RIP_ADD_GAP
|   |   |
|   |   |--PRO_E_SMT_EDGE_RIP_DIM_1
|   |   |
|   |   |   |--PRO_E_SMT_EDGE_RIP_DIM_1_TYPE
|   |   |   |
|   |   |   |--PRO_E_SMT_EDGE_RIP_DIM_1_VAL
|   |   |   |
|   |   |--PRO_E_SMT_EDGE_RIP_DIM_2
|   |   |
|   |   |   |--PRO_E_SMT_EDGE_RIP_DIM_2_TYPE
|   |   |   |
|   |   |   |--PRO_E_SMT_EDGE_RIP_DIM_2_VAL
|   |   |   |
|   |   |--PRO E SMT EDGE RIP FLIP

```

Figure 52-9: PRO_E_SMT_DEV_LEN_CALCULATION

```

PRO_E_SMT_DEV_LEN_CALCULATION
|
|--PRO_E_SMT_DEV_LEN_SOURCE
|
|--PRO_E_SMT_DEV_LEN_Y_FACTOR
|   |
|   |--PRO_E_SMT_DEV_LEN_Y_FACTOR_TYPE
|   |
|   |--PRO_E_SMT_DEV_LEN_Y_FACTOR_VALUE
|   |
|--PRO E SMT DEV LEN BEND TABLE

```


Apart from the usual element for the tree root, a Sheetmetal Flange Wall feature contains the following elements:

- `PRO_E_FEATURE_TYPE`—Specifies the feature type and should be `PRO_FEAT_WALL`.
- `PRO_E_SMT_WALL_TYPE`—Specifies the wall type and must be
 - `PRO_SMT_WALL_TYPE_FLANGE`
- `PRO_E_STD_FEATURE_NAME`— Specifies the name of the feature.
- `PRO_E_SMT_FLANGE_TYPE`—Specifies the type of flange wall:
 - `PRO_FLANGE_WALL_TYPE_2D_SWEPT`—a swept flange wall in the default orientation.
 - `PRO_FLANGE_WALL_TYPE_3D_SWEPT`—a swept flange wall with non-default directions, orientations, and start points.
 - `PRO_FLANGE_WALL_TYPE_EXTRUDE`—a flange wall extruded from a sketching plane.
- `PRO_E_STD_CURVE_COLLECTION_APPL`—Specifies the attachment edge chain. If the type is swept, this can reference multiple non-tangent edges selected as one by one or using any of the other instruction types. If the flange wall type is to be "Extruded", this must contain a One by One chain with a linear edge.
- `PRO_E_SMT_FLANGE_TRAJ_CRV_NORM`—Specifies whether the flange wall should progress along the default edge chain direction or in the opposite direction.
- `PRO_E_STD_SECTION`— Specifies the wall section.

Wall sections can be standard or user-defined. Standard wall sections are stored in the location `<Pro/ENGINEER Loadpoint>/text/smt`. These standard sections can be retrieved, their dimensions modified, and can be added directly into the `PRO_E_STD_SECTION` element tree as the `PRO_E_SKETCHER` element. This does not require definition of a sketch plane or viewing direction, and it does not require an incomplete feature to be created as is described in the chapter "Creating Sketched Features". When standard sections are used to create the wall, the `Pro/ENGINEER` user interface will show the correct type of section in the drop down menu (for example: "I", "Arc", etc.).

If a user-defined section is to be used to create a flat wall, it must conform to the following restrictions:

- It must be a 2D section.
- It must not be named the same name as one of the default section types from the Pro/ENGINEER loadpoint.
- It must contain a horizontal centerline with a coordinate system located on it. The centerline represents the alignment with the attachment wall, and the coordinate system represents the attachment point for the section.
- The section may optionally contain bent or straight edges. It may also contain sheet metal section entities that assist in constructing the correct swept geometry.

When user-defined sections are used to create the wall, Pro/ENGINEER automatically creates necessary sketching planes for the section during creation. Therefore the section may be assigned directly into the PRO_E_SKETCHER element without defining the sketch plane and without creating the feature as incomplete. After the feature has been created, if the wall type is 3D swept or Extruded, the 3D section can be extracted from the feature element tree and the section can be modified to include references to other geometric entities in the sheet metal part.

- PRO_E_SMT_WALL_SHARPS_TO_BENDS—If PRO_B_TRUE then Pro/ENGINEER attempts to convert sharp edges in the section to bends.
- PRO_E_SMT_FLANGE_SEC_FLIP—Specifies whether or not to flip the direction of the section for user-defined sections.
- PRO_E_SMT_FLANGE_DEPTH—Specifies the depth of the flange (the extent of the flange cover. This element governs the results for extruded flange walls only. For swept walls, the extents are governed by the rules in the element PRO_E_STD_CURV_COLLECTION_APPL, which might include trim values and boundary geometry.
- PRO_E_SMT_FILLETS—Specifies the bend properties of the sheet metal wall.
 - PRO_E_SMT_FILLETS_USE_RAD—If true, a bend is applied, if false, no bend is used.
 - PRO_E_SMT_FILLETS_SIDE—Specifies fillet side and has the following permitted values:
 - PRO_BEND_RAD_OUTSIDE—apply the bend radius to the outside of the bend.

- PRO_BEND_RAD_INSIDE—apply the bend radius to the inside of the bend.
 - PRO_E_SMT_FILLETS_VALUE—the bend radius.
- PRO_E_SMT_WALL_HEIGHT—Specifies the height of the attachment wall. It has the following elements:
 - PRO_E_SMT_WALL_HEIGHT_TYPE specifies the manner in which the newly created wall feature attaches to the attachment edge. This element takes the following values:
 - PRO_SMT_WALL_HEIGHT_AUTO—Specifies that the wall feature attaches to the attachment edge by trimming the height of the attachment wall automatically.
 - PRO_SMT_WALL_HEIGHT_VALUE—Specifies that the wall feature attaches to the attachment edge by trimming the height of the attachment wall by a specified value.
 - PRO_SMT_WALL_HEIGHT_APP_BEND—Specifies that the wall feature appends to the attachment edge without trimming the height of the attachment wall.
 - PRO_E_SMT_WALL_HEIGHT_VALUE specifies the value of the height of the attachment wall.
- PRO_E_SMT_BEND_RELIEF—Specifies bend relief. Refer to the section Feature Element Tree for the Sheetmetal Flat Wall Feature for more information on this element subtree.
- PRO_E_SMT_WALL_THICKNESS_FLIP—Indicates whether or not to flip the thickness direction of the new wall.
- PRO_E_SMT_CORNER_RELIEF—Indicates a compound element representing corner relief. Corner relief is added at each intersection of a pair of bends.
- PRO_E_SMT_MTR_CUTS—Indicates a compound element representing miter cuts.
- PRO_E_SMT_AUTO_EXLD_EDGE—Specifies whether to set automatic exclusion of edges. Pro/ENGINEER uses the following set of rules and logic in order to execute the automatic wall segment excluding:
 - Long wall segment has lower excluding priority than short wall segment
 - Small wall segment that is neighbor to long wall has high excluding priority than other short wall segments

- A wall segment whose overlapping area at the intersection of bend surfaces of neighborhood wall segments is maximum has the highest excluding priority
- PRO_E_SMT_CORNERS_ARR—Specifies the edge transitions.
- PRO_E_SMT_DEV_LEN_CALCULATION—Specifies the properties used to calculate the development length. See the section on this element subtree in the section “Flat Wall” for more information.

The Element Subtree for PRO_E_SMT_FLANGE_DEPTH

PRO_E_SMT_FLANGE_DEPTH has the following elements:

- PRO_E_SMT_FLANGE_SIDE_1_DEPTH— Specifies first side of the flange extents and has the following elements:
 - PRO_WALL_LEN_TYPE_NONE—the flange does not extend in this direction.
 - PRO_WALL_LEN_TYPE_BLIND—the flange extends a specified length value in this direction.
 - PRO_WALL_LEN_TYPE_BLIND_SYM—the flange extends a symmetric length value in both direction. If this is used for side 1, side 2 must use PRO_WALL_LEN_TYPE_NONE.
 - PRO_WALL_LEN_TYPE_TO_REF—the flange extends to a selected geometric reference.
 - PRO_WALL_LEN_TYPE_TO_END—the flange extends to the end of the attachment reference.
 - PRO_E_SMT_FLANGE_DEPTH_OFFSET—specifies the depth offset for blind and symmetric blind extents.
 - PRO_E_SMT_FLANGE_DEPTH_REF—specifies the depth placement reference for "to ref" extents.
- PRO_E_SMT_FLANGE_SIDE_2_DEPTH— Specifies side2 the second side of the flange. This subtree is identical to the first side.

The Element Subtree for PRO_E_SMT_CORNER_RELIEF

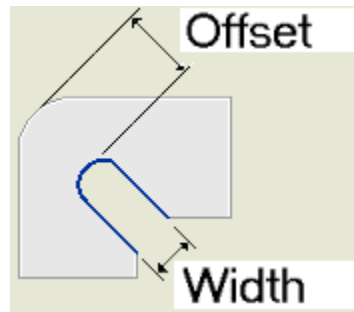
Corner relief is required when multiple non-tangent edges are used for attachment of the flange wall. The element PRO_E_SMT_CORNER_RELIEF represents corner relief in the feature. It has the following properties:

- PRO_E_SMT_CORNER_RELIEF_TYPE specifies the types of corner reliefs—
 - PRO_CORNER_RELIEF_NO—Pro/ENGINEER does not add relief and generates square corners.
 - PRO_CORNER_RELIEF_V_NOTCH—Pro/ENGINEER adds a V notch shape cut at the corners.
 - PRO_CORNER_RELIEF_CIRCULAR—Pro/ENGINEER adds a circular shape relief at the corners with a radius dimension.
 - PRO_CORNER_RELIEF_OBROUND—Pro/ENGINEER adds an obround relief at the corners with a specified diameter and depth.
 - PRO_CORNER_RELIEF_RECTANGULAR—Pro/ENGINEER adds a rectangular relief at the corners with a specified width and depth.
- PRO_E_SMT_CORNER_RELIEF_WIDTH a compound element with the following elements—
 - PRO_E_SMT_CORNER_RELIEF_WIDTH_TYPE—This is one of the members of *ProSmdRelType*. See table Relation Value Types for the list of value types permitted.
 - PRO_E_SMT_CORNER_RELIEF_WIDTH_VAL—This is the value for the dimension, if the width type is PRO_DIM_ENTER.
- PRO_E_SMT_CORNER_RELIEF_DEPTH a compound element with the following elements—
 - PRO_E_SMT_CORNER_RELIEF_DEPTH_TYPE—This is one of the members of *ProSmdRelType*. See table Relation Value Types for the list of value types permitted.
 - PRO_E_SMT_CORNER_RELIEF_DEPTH_VAL—This is the value for the dimension, if the width type is PRO_DIM_ENTER.

The Element Subtree for PRO_E_SMT_MTR_CUTS

A miter cut removes material from any profile intersecting wall segments. The miter cut is controlled by two dimensions—width and offset as shown in the figure below. Offset is the distance between the end of the miter cut to the placement chain. Pro/ENGINEER uses half of the specified width value to cut the material of each side of the centerline of the miter cut.

Figure 52-10: Miter Cut



It has the following elements:

- PRO_E_SMT_MTR_CUTS_ADD—Specifies whether to add miter cuts.
- PRO_E_SMT_MTR_CUTS_KEEP_DEF_AREAS—Specifies the deformation area of the miter cut. A deformation area is a section of sheet metal that helps to accurately stretch the material when you unbend the sheet metal part.
- PRO_E_SMT_MTR_CUTS_WIDTH
 - PRO_E_SMT_MTR_CUTS_WIDTH_TYPE—This is one of the members of *ProSmdRelType*. See table Relation Value Types for the list of value types permitted.
 - PRO_E_SMT_MTR_CUTS_WIDTH_VAL—This is the value for the dimension, if the width type is PRO_DIM_ENTER.
- PRO_E_SMT_MTR_CUTS_OFFSET
 - PRO_E_SMT_MTR_CUTS_OFFSET_TYPE—This is one of the members of *ProSmdRelType*. See table Relation Value Types for the list of value types permitted.

- PRO_E_SMT_MTR_CUTS_OFFSET_VAL—This is the value for the dimension, if the width type is PRO_DIM_ENTER.

The Element Subtree for PRO_E_SMT_CORNERS_ARR

When the flange is attached to multiple non-tangent edges, it is possible to define edge transitions for each such intersection. The members of the array element PRO_E_SMT_CORNERS_ARR each define the edge transition for a particular corner intersection. Each member has a subelement called PRO_E_SMT_EDGE_RIP which contains the following:

- PRO_E_SMT_EDGE_RIP_TYPE specifies edge treatment types and can be as follows:
 - PRO_EDGE_RIP_OPEN
 - PRO_EDGE_RIP_BLIND
 - PRO_EDGE_RIP_MITER_CUT
 - PRO_EDGE_RIP_OVERLAP
 - PRO_EDGE_RIP_CONNECT
- PRO_E_SMT_EDGE_RIP_ADD_GAP specifies whether to add a gap.
- PRO_E_SMT_EDGE_RIP_DIM_1 specifies the first side's properties.
 - PRO_E_SMT_EDGE_RIP_DIM_1_TYPE—This is one of the members of *ProSmdRelType*. See table Relation Value Types for the list of value types permitted.
 - PRO_E_SMT_EDGE_RIP_DIM_1_VAL—This is the value for the dimension, if the width type is PRO_DIM_ENTER.
- PRO_E_SMT_EDGE_RIP_DIM_2 specifies the second side's properties.
 - PRO_E_SMT_EDGE_RIP_DIM_2_TYPE—This is one of the members of *ProSmdRelType*. See table Relation Value Types for the list of value types permitted.
 - PRO_E_SMT_EDGE_RIP_DIM_2_VAL—This is the value for the dimension, if the width type is PRO_DIM_ENTER.
- PRO_E_SMT_EDGE_RIP_FLIP specifies whether to flip the overlapping side.

Relation Value Types

ProSmdRelType	Description
PRO_DIM_THICK	The part thickness
PRO_DIM_DOUBLE_THICK	2 x the part thickness
PRO_DIM_ENTER	A user-defined value
PRO_DIM_DEF_CRN_REL_WIDTH	The value of the Sheetmetal Parameter SMT_DFLT_CRNR_REL_WIDTH, only allowed for corner relief width.
PRO_DIM_DEF_CRN_REL_DEPTH	The value of the Sheetmetal Parameter SMT_DFLT_CRNR_REL_DEPTH, only allowed for corner relief depth.
PRO_DIM_MINUS_THICK	-1 x the part thickness
PRO_DIM_MINUS_DOUBLE_THICK	-2 x the part thickness
PRO_DIM_DFLT_EDGE_TREAT_WIDTH	The value of the Sheetmetal Parameter SMT_DFLT_EDGE_TREAT_WIDTH, only allowed for edge transition width
PRO_DIM_DFLT_MITER_CUT_WIDTH	The value of the Sheetmetal Parameter SMT_DFLT_MITER_CUT_WIDTH, only allowed for miter cut width
PRO_DIM_DFLT_MITER_CUT_OFFSET	The value of the Sheetmetal Parameter SMT_DFLT_MITER_CUT_OFFSET, only allowed for miter cut offset
PRO_DIM_THICK_1_1	1.1 x the part thickness
PRO_DIM_THICK_05	0.5 x the part thickness
PRO_DIM_SMT_GAP	The value of the Sheetmetal Parameter SMT_GAP
PRO_DIM_MINUS_THICK_05	-0.5 x the part thickness

Creating a Sheetmetal Flange Wall Feature

Function Introduced

- **ProFeatureCreate()**

Use the function **ProFeatureCreate()** to create a Sheetmetal Flange Wall Feature based on element tree input. For more information about **ProFeatureCreate()**, refer to the section “Overview of Feature Creation” in the “Element Trees: Principles of Feature Creation” chapter.

Redefining a Sheetmetal Flange Wall Feature

Function Introduced

- **ProFeatureRedefine()**

Use the function **ProFeatureRedefine()** to redefine a Sheetmetal Flange Wall Feature based on the changes made in the element tree. For more information about **ProFeatureRedefine()**, refer to the section “Feature Redefine” in the “Element Trees: Principles of Feature Creation” chapter.

Accessing a Sheetmetal Flange Wall Feature

Function Introduced

- **ProFeatureElemtreeExtract()**

Use the function **ProFeatureElemtreeExtract()** to create a feature element tree that describes the contents of a Sheetmetal Flange Wall Feature and to retrieve the element tree description of a Sheetmetal Flange Wall Feature. For more information about **ProFeatureElemtreeExtract()**, refer to the section “Feature Inquiry” in the “Element Trees: Principles of Feature Creation” chapter.

Example 3: Creation of Sheetmetal Flange Wall feature using Pro/TOOLKIT

The example demonstrates how to create a sheetmetal flange wall feature using Pro/TOOLKIT. The feature is created using two external properties: the wall thickness determines the bend radius, and the attachment edge(s) are obtained from the currently selected edges in the Pro/ENGINEER model. If a single edge is selected, it is converted to a tangent chain before it is used to create the feature. The section is the 2d section handle for the I-wall section provided with Pro/ENGINEER.

```
/*=====*\
FILE      : UgSmtFlgWallCreate.c
PURPOSE   : Sheetmetal flange wall feature creation example
\*=====*/

#include <ProToolkit.h>
#include <ProFeature.h>
#include <ProSelbuffer.h>
#include <ProReference.h>
#include <ProCrvcollection.h>
#include <ProUtil.h>
#include <ProSmtFlangeWall.h>
```

```

#include <ProSheetmetal.h>
#include <ProMenuBar.h>
#include <ProPopupMenu.h>
#include <ProMdl.h>

/*=====*\
FUNCTION: UserSmtFlgWallCreate
PURPOSE: Create sheetmetal flange wall features.
\*=====*/
int UserSmtFlgWallCreate (ProCollection ref_edge_collection,
    double bend_radius,
    ProSection section)
{
    ProElement pro_e_feature_tree;
    ProElement pro_e_feature_type;
    ProElement pro_e_smt_wall_type;
    ProElement pro_e_std_feature_name;
    ProElement pro_e_smt_flange_type;
    ProElement pro_e_std_curve_collection_appl;
    ProElement pro_e_smt_flange_traj_crv_norm;
    ProElement pro_e_std_section;
    ProElement pro_e_sketcher;
    ProElement pro_e_smt_wall_sharps_to_bends;
    ProElement pro_e_smt_flange_sec_flip;
    ProElement pro_e_smt_flange_depth;
    ProElement pro_e_smt_flange_side_1_depth;
    ProElement pro_e_smt_flange_depth_type;
    ProElement pro_e_smt_flange_depth_offset;
    ProElement pro_e_smt_flange_side_2_depth;
    ProElement pro_e_smt_fillet;
    ProElement pro_e_smt_fillet_use_rad;
    ProElement pro_e_smt_fillet_side;
    ProElement pro_e_smt_fillet_value;
    ProElement pro_e_smt_wall_height;
    ProElement pro_e_smt_wall_height_type;
    ProElement pro_e_smt_wall_height_value;
    ProElement pro_e_smt_bend_relief;
    ProElement pro_e_smt_bend_relief_side1;
    ProElement pro_e_bend_relief_type;
    ProElement pro_e_bend_relief_width;
    ProElement pro_e_bend_relief_depth;
    ProElement pro_e_bend_relief_angle;
    ProElement pro_e_smt_bend_relief_side2;
    ProElement pro_e_smt_wall_thickness_flip;
    ProElement pro_e_smt_dev_len_calculation;
    ProElement pro_e_smt_dev_len_source;
    ProElement pro_e_smt_dev_len_y_factor;
    ProElement pro_e_smt_dev_len_y_factor_value;
    ProElement pro_e_smt_dev_len_y_factor_type;
    ProElement pro_e_smt_dev_len_bend_table;
    ProMdl model;

```

```

ProModelitem model_item;
ProFeatureCreateOptions opts [1];
ProSelection model_sel;
ProFeature feature;
ProErrorlist errors;
ProError status;
ProSelection feat_sel;

/*-----*\
Processing Element PRO_E_FEATURE_TREE
\*-----*/
    status = ProElementAlloc ( PRO_E_FEATURE_TREE, &pro_e_feature_tree );

/*-----*\
Processing Element PRO_E_FEATURE_TYPE
\*-----*/
    status = ProElementAlloc ( PRO_E_FEATURE_TYPE, &pro_e_feature_type );
    status = ProElementIntegerSet ( pro_e_feature_type, PRO_FEAT_WALL );
    status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
        pro_e_feature_type );

/*-----*\
Processing Element PRO_E_SMT_WALL_TYPE
\*-----*/
    status = ProElementAlloc ( PRO_E_SMT_WALL_TYPE, &pro_e_smt_wall_type );
    status = ProElementIntegerSet ( pro_e_smt_wall_type,
        PRO_SMT_WALL_TYPE_FLANGE );
    status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
        pro_e_smt_wall_type );

/*-----*\
Processing Element PRO_E_SMT_FLANGE_TYPE
\*-----*/
    status = ProElementAlloc ( PRO_E_SMT_FLANGE_TYPE, &pro_e_smt_flange_type
);
    status = ProElementIntegerSet ( pro_e_smt_flange_type,
        PRO_FLANGE_WALL_TYPE_2D_SWEPT );
    status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
        pro_e_smt_flange_type );

/*-----*\
Processing Element PRO_E_STD_CURVE_COLLECTION_APPL
\*-----*/
    status = ProElementAlloc ( PRO_E_STD_CURVE_COLLECTION_APPL,
        &pro_e_std_curve_collection_appl );
    status = ProElementCollectionSet( pro_e_std_curve_collection_appl,
        ref_edge_collection);
    status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
        pro_e_std_curve_collection_appl );

/*-----*\

```

```

Processing Element PRO_E_SMT_FLANGE_TRAJ_CRV_NORM
/*-----*/
    status = ProElementAlloc ( PRO_E_SMT_FLANGE_TRAJ_CRV_NORM,
        &pro_e_smt_flange_traj_crv_norm );
    status = ProElementIntegerSet ( pro_e_smt_flange_traj_crv_norm,
        PRO_FLANGE_TRAJ_NORM_NO_FLIP);
    status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
        pro_e_smt_flange_traj_crv_norm );

/*-----*\
Processing Element PRO_E_STD_SECTION
/*-----*/
    status = ProElementAlloc ( PRO_E_STD_SECTION, &pro_e_std_section );
    status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
        pro_e_std_section );

/*-----*\
Processing Element PRO_E_SKETCHER
/*-----*/
    status = ProElementAlloc ( PRO_E_SKETCHER, &pro_e_sketcher );
    status = ProElementSpecialvalueSet ( pro_e_sketcher, section );
    status = ProElemtreeElementAdd ( pro_e_std_section, NULL,
        pro_e_sketcher );

/*-----*\
Processing Element PRO_E_SMT_WALL_SHARPS_TO_BENDS
/*-----*/
    status = ProElementAlloc ( PRO_E_SMT_WALL_SHARPS_TO_BENDS,
        &pro_e_smt_wall_sharps_to_bends );
    status = ProElementIntegerSet ( pro_e_smt_wall_sharps_to_bends,
        PRO_B_TRUE );
    status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
        pro_e_smt_wall_sharps_to_bends );

/*-----*\
Processing Element PRO_E_SMT_FLANGE_SEC_FLIP
/*-----*/
    status = ProElementAlloc ( PRO_E_SMT_FLANGE_SEC_FLIP,
        &pro_e_smt_flange_sec_flip );
    status = ProElementIntegerSet ( pro_e_smt_flange_sec_flip,
        PRO_B_FALSE );
    status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
        pro_e_smt_flange_sec_flip );

/*-----*\
Processing Element PRO_E_SMT_FLANGE_DEPTH
/*-----*/
    status = ProElementAlloc ( PRO_E_SMT_FLANGE_DEPTH,
        &pro_e_smt_flange_depth );
    status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
        pro_e_smt_flange_depth );

```

```

/*-----*\
Processing Element PRO_E_SMT_FLANGE_SIDE_1_DEPTH
\*-----*/
    status = ProElementAlloc ( PRO_E_SMT_FLANGE_SIDE_1_DEPTH,
        &pro_e_smt_flange_side_1_depth );
    status = ProElemtreeElementAdd ( pro_e_smt_flange_depth, NULL,
        pro_e_smt_flange_side_1_depth );

/*-----*\
Processing Element PRO_E_SMT_FLANGE_DEPTH_TYPE
\*-----*/
    status = ProElementAlloc ( PRO_E_SMT_FLANGE_DEPTH_TYPE,
        &pro_e_smt_flange_depth_type );
    status = ProElementIntegerSet ( pro_e_smt_flange_depth_type,
        PRO_WALL_LEN_TYPE_TO_END);
    status = ProElemtreeElementAdd ( pro_e_smt_flange_side_1_depth, NULL,
        pro_e_smt_flange_depth_type );

/*-----*\
Processing Element PRO_E_SMT_FLANGE_SIDE_2_DEPTH
\*-----*/
    status = ProElementAlloc ( PRO_E_SMT_FLANGE_SIDE_2_DEPTH,
        &pro_e_smt_flange_side_2_depth );
    status = ProElemtreeElementAdd ( pro_e_smt_flange_depth, NULL,
        pro_e_smt_flange_side_2_depth );

/*-----*\
Processing Element PRO_E_SMT_FLANGE_DEPTH_TYPE
\*-----*/
    status = ProElementAlloc ( PRO_E_SMT_FLANGE_DEPTH_TYPE,
        &pro_e_smt_flange_depth_type );
    status = ProElementIntegerSet ( pro_e_smt_flange_depth_type,
        PRO_WALL_LEN_TYPE_TO_END );
    status = ProElemtreeElementAdd ( pro_e_smt_flange_side_2_depth, NULL,
        pro_e_smt_flange_depth_type );

/*-----*\
Processing Element PRO_E_SMT_FILLETS
\*-----*/
    status = ProElementAlloc ( PRO_E_SMT_FILLETS, &pro_e_smt_filletts );
    status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
        pro_e_smt_filletts );

/*-----*\
Processing Element PRO_E_SMT_FILLETS_USE_RAD
\*-----*/
    status = ProElementAlloc ( PRO_E_SMT_FILLETS_USE_RAD,
        &pro_e_smt_filletts_use_rad );
    status = ProElementIntegerSet ( pro_e_smt_filletts_use_rad, PRO_B_TRUE );
    status = ProElemtreeElementAdd ( pro_e_smt_filletts, NULL,

```

```

    pro_e_smt_filletlets_use_rad );

/*-----*\
Processing Element PRO_E_SMT_FILLETS_SIDE
\*-----*/
status = ProElementAlloc ( PRO_E_SMT_FILLETS_SIDE,
    &pro_e_smt_filletlets_side );
status = ProElementIntegerSet ( pro_e_smt_filletlets_side,
    PRO_BEND_RAD_INSIDE );
status = ProElementtreeElementAdd ( pro_e_smt_filletlets, NULL,
    pro_e_smt_filletlets_side );

/*-----*\
Processing Element PRO_E_SMT_FILLETS_VALUE
\*-----*/
status = ProElementAlloc ( PRO_E_SMT_FILLETS_VALUE,
    &pro_e_smt_filletlets_value );
status = ProElementDoubleSet ( pro_e_smt_filletlets_value, bend_radius );
status = ProElementtreeElementAdd ( pro_e_smt_filletlets, NULL,
    pro_e_smt_filletlets_value );

/*-----*\
Processing Element PRO_E_SMT_WALL_HEIGHT
\*-----*/
status = ProElementAlloc ( PRO_E_SMT_WALL_HEIGHT, &pro_e_smt_wall_height
);
status = ProElementtreeElementAdd ( pro_e_feature_tree, NULL,
    pro_e_smt_wall_height );

/*-----*\
Processing Element PRO_E_SMT_WALL_HEIGHT_TYPE
\*-----*/
status = ProElementAlloc ( PRO_E_SMT_WALL_HEIGHT_TYPE,
    &pro_e_smt_wall_height_type );
status = ProElementIntegerSet ( pro_e_smt_wall_height_type,
    PRO_SMT_WALL_HEIGHT_NONE);
status = ProElementtreeElementAdd ( pro_e_smt_wall_height, NULL,
    pro_e_smt_wall_height_type );

/*-----*\
Processing Element PRO_E_SMT_BEND_RELIEF
\*-----*/
status = ProElementAlloc ( PRO_E_SMT_BEND_RELIEF, &pro_e_smt_bend_relief
);
status = ProElementtreeElementAdd ( pro_e_feature_tree, NULL,
    pro_e_smt_bend_relief );

/*-----*\
Processing Element PRO_E_SMT_BEND_RELIEF_SIDE1
\*-----*/
status = ProElementAlloc ( PRO_E_SMT_BEND_RELIEF_SIDE1,

```

```

        &pro_e_smt_bend_relief_side1 );
status = ProElementtreeElementAdd ( pro_e_smt_bend_relief, NULL,
    pro_e_smt_bend_relief_side1 );

/*-----*\
Processing Element PRO_E_BEND_RELIEF_TYPE
\*-----*/
status = ProElementAlloc ( PRO_E_BEND_RELIEF_TYPE,
    &pro_e_bend_relief_type );
status = ProElementIntegerSet ( pro_e_bend_relief_type, PRO_BEND_RLF_RIP
);
status = ProElementtreeElementAdd ( pro_e_smt_bend_relief_side1, NULL,
    pro_e_bend_relief_type );

/*-----*\
Processing Element PRO_E_SMT_BEND_RELIEF_SIDE2
\*-----*/
status = ProElementAlloc ( PRO_E_SMT_BEND_RELIEF_SIDE2,
    &pro_e_smt_bend_relief_side2 );
status = ProElementtreeElementAdd ( pro_e_smt_bend_relief, NULL,
    pro_e_smt_bend_relief_side2 );

/*-----*\
Processing Element PRO_E_BEND_RELIEF_TYPE
\*-----*/
status = ProElementAlloc ( PRO_E_BEND_RELIEF_TYPE,
    &pro_e_bend_relief_type );
status = ProElementIntegerSet ( pro_e_bend_relief_type, PRO_BEND_RLF_RIP
);
status = ProElementtreeElementAdd ( pro_e_smt_bend_relief_side2, NULL,
    pro_e_bend_relief_type );

/*-----*\
Processing Element PRO_E_SMT_WALL_THICKNESS_FLIP
\*-----*/
status = ProElementAlloc ( PRO_E_SMT_WALL_THICKNESS_FLIP,
    &pro_e_smt_wall_thickness_flip );
status = ProElementIntegerSet ( pro_e_smt_wall_thickness_flip,
    PRO_B_FALSE );
status = ProElementtreeElementAdd ( pro_e_feature_tree, NULL,
    pro_e_smt_wall_thickness_flip );

/*-----*\
Processing Element PRO_E_SMT_DEV_LEN_CALCULATION
\*-----*/
status = ProElementAlloc ( PRO_E_SMT_DEV_LEN_CALCULATION,
    &pro_e_smt_dev_len_calculation );
status = ProElementtreeElementAdd ( pro_e_feature_tree, NULL,
    pro_e_smt_dev_len_calculation );

/*-----*\

```

```

Processing Element PRO_E_SMT_DEV_LEN_SOURCE
/*-----*/
status = ProElementAlloc ( PRO_E_SMT_DEV_LEN_SOURCE,
    &pro_e_smt_dev_len_source );
status = ProElementIntegerSet ( pro_e_smt_dev_len_source,
    PRO_DVL_SRC_FEAT_YF_ONLY );
status = ProElemtreeElementAdd ( pro_e_smt_dev_len_calculation, NULL,
    pro_e_smt_dev_len_source );

/*-----*\
Processing Element PRO_E_SMT_DEV_LEN_Y_FACTOR
/*-----*/
status = ProElementAlloc ( PRO_E_SMT_DEV_LEN_Y_FACTOR,
    &pro_e_smt_dev_len_y_factor );
status = ProElemtreeElementAdd ( pro_e_smt_dev_len_calculation, NULL,
    pro_e_smt_dev_len_y_factor );

/*-----*\
Processing Element PRO_E_SMT_DEV_LEN_Y_FACTOR_VALUE
/*-----*/
status = ProElementAlloc ( PRO_E_SMT_DEV_LEN_Y_FACTOR_VALUE,
    &pro_e_smt_dev_len_y_factor_value );
status = ProElementDoubleSet ( pro_e_smt_dev_len_y_factor_value,
    0.320000 );
status = ProElemtreeElementAdd ( pro_e_smt_dev_len_y_factor, NULL,
    pro_e_smt_dev_len_y_factor_value );

/*-----*\
Create the feature using the element tree
/*-----*/
status = ProMdlCurrentGet (&model);
status = ProMdlToModelitem( model, &model_item );
status = ProSelectionAlloc (NULL, &model_item, &model_sel);

opts[0] = PRO_FEAT_CR_NO_OPTS;
status = ProFeatureCreate (model_sel, pro_e_feature_tree, opts, 1,
    &feature, &errors);

if (status == PRO_TK_NO_ERROR)
{
    ProSelectionAlloc (NULL, &feature, &feat_sel);
    ProSelbufferClear ();
    ProSelbufferSelectionAdd (feat_sel);
    ProSelectionFree (&feat_sel);
}

return (PRO_TK_NO_ERROR);
}

/*=====*\
FUNCTION: UserSmtFlgWallCreate_OA_CreateISec

```



```

PURPOSE:  Create sheetmetal flange wall features using Object/Action
          selection and Pro/E's predefined I-section.
\*=====*/
int UserSmtFlgWallCreate_OA_CreateISec ()
{
    ProSelection* sel;
    ProCollection collection;
    ProCrvcollinstr instr;
    ProReference reference;
    ProPart part;
    ProDimension thickness_dim;
    double thickness;
    ProSection i_wall_sec;
    ProPath w_mdldpath;
    ProCharPath mdldpath, secpath;
    char* prodir;

/*-----*\
Extract the current selected edge or edge chain as the wall attachment
\*-----*/
    status = ProSelbufferSelectionsGet (&sel);

    status = ProSelectionCollectionGet (sel[0], &collection);

/*-----*\
If a single edge is selected, allocate the chain as a tangent chain
\*-----*/
    if (status != PRO_TK_NO_ERROR)
    {
        status = ProCrvcollectionAlloc ( &collection );
        status = ProCrvcollinstrAlloc ( PRO_CURVCOLL_TAN_INSTR, &instr );
        status = ProSelectionToReference ( sel[0], &reference );
        status = ProCrvcollinstrReferenceAdd ( instr, reference );
        status = ProCrvcollectionInstructionAdd ( collection, instr );

        ProReferenceFree (reference);
    }

/*-----*\
Extract the wall thickness to use for the bend radius
\*-----*/
    status = ProMdlCurrentGet ((ProMdl*)&part);
    status = ProSmtPartThicknessGet(part, &thickness_dim);
    status = ProDimensionValueGet (&thickness_dim, &thickness);

/*-----*\
Retrieve the Pro/ENGINEER default section for "I" walls
\*-----*/
    prodir = getenv ("PRO_DIRECTORY");
    if (prodir == NULL)
return PRO_TK_INVALID_DIR;

```

```

ProTKSprintf (secpath, "%s/text/smt/", prodir);
ProTKSprintf (mdlpath, "%si_wall.sec", secpath);
ProStringToWstring (w_mdlpath, mdlpath);
status = ProMdlLoad (w_mdlpath, PRO_VALUE_UNUSED, PRO_B_FALSE,
    (ProMdl*)&i_wall_sec);

/*-----*\
Create the feature with the extracted information
\*-----*/
UserSmtFlgWallCreate (collection, thickness, i_wall_sec);

ProCollectionFree (&collection);

return (PRO_TK_NO_ERROR);
}

/*=====*\
FUNCTION: UserSmtFlgWallCreate_OA_CreateUserDefSec
PURPOSE: Create sheetmetal flange wall features using Object/Action
         selection and a user-selection 2D section.
\*=====*/
int UserSmtFlgWallCreate_OA_CreateUserDefSec ()
{
    ProSelection* sel;
    ProCollection collection;
    ProCrvcollinstr instr;
    ProReference reference;
    ProPart part;
    ProDimension thickness_dim;
    double thickness;
    ProSection userdef_sec;
    ProPath w_mdlpath;
    ProCharPath mdlpath, secpath;
    char* prodir;

/*-----*\
Prompt the user to select a section for the flange wall
\*-----*/
    status = UserSectionOpen (&userdef_sec);

    if (status != PRO_TK_NO_ERROR)
        return (status);

/*-----*\
Extract the current selected edge or edge chain as the wall attachment
\*-----*/
    status = ProSelbufferSelectionsGet (&sel);

    status = ProSelectionCollectionGet (sel[0], &collection);

```

```

if (status != PRO_TK_NO_ERROR)
{
    status = ProCrvcollectionAlloc ( &collection );
    status = ProCrvcollinstrAlloc ( PRO_CURVCOLL_TAN_INSTR, &instr );
    status = ProSelectionToReference ( sel[0], &reference );
    status = ProCrvcollinstrReferenceAdd ( instr, reference );
    status = ProCrvcollectionInstructionAdd ( collection, instr );

    ProReferenceFree (reference);
}

/*-----*\
Extract the wall thickness to use for the bend radius
\*-----*/
status = ProMdlCurrentGet ((ProMdl*)&part);
status = ProSmtPartThicknessGet(part, &thickness_dim);
status = ProDimensionValueGet (&thickness_dim, &thickness);

/*-----*\
Create the feature with the extracted information
\*-----*/
UserSmtFlgWallCreate (collection, thickness, userdef_sec);

ProMdlErase (userdef_sec);

ProCollectionFree (&collection);

return (PRO_TK_NO_ERROR);
}

#define OA 1 /* Standard OA in the menu system, typically gray out
the button when not usable*/
#define PM 2 /* OA in a popup menu, typically remove the button when
not usable */

/*=====*\
FUNCTION: UserSmtFlgWallCreate_TestLow
PURPOSE: Access function for the button for smt flange wall feature
creation
\*=====*/
static uiCmdAccessState UserSmtFlgWallCreate_TestLow (ProSelection* sels,
int mode)
{
    uiCmdAccessState access_result;
    ProBoolean should_free = PRO_B_FALSE;
    ProError status;
    ProMdl mdl;
    ProMdlType type;
    ProMdlsubtype subtype;
    int size, refs_size;
    ProCollection collection;

```

```

ProCollectionType coll_type;
ProCrvcollinstr* instrs;
int i, j;
ProCrvcollinstrType instr_type;
ProReference* refs;
ProModelitem item;
ProBoolean invalid_ref;

/*-----*\
Set the default return if the button is unusable.
\*-----*/
if (mode == OA)
    access_result = ACCESS_UNAVAILABLE;
else
    access_result = ACCESS_REMOVE;

status = ProMdlCurrentGet (&mdl);

if (status != PRO_TK_NO_ERROR)
    return (access_result);

status = ProMdlTypeGet (mdl, &type);

status = ProMdlSubtypeGet (mdl, &subtype);

if (type != PRO_MDL_PART || subtype != PROMDLSTYPE_PART_SHEETMETAL)
    return (access_result);

/*-----*\
If called without selections, extract the current selections from
the buffer.
\*-----*/
if (sels == NULL)
{
    status = ProSelbufferSelectionsGet (&sels);

    if (status != PRO_TK_NO_ERROR)
return access_result;

    if (sels == NULL)
return access_result;

    should_free = PRO_B_TRUE;
}

/*-----*\
This command allows only one selection.
\*-----*/
status = ProArraySizeGet (sels, &size);

if (status != PRO_TK_NO_ERROR)

```

```

        return access_result;

    if (size == 1)
    {
/*-----*\
Extract the crv collection selected by the user
\*-----*/
        status = ProSelectionCollectionGet (sels [0], &collection);
        ProTKPrintf ("Status of ProSelectionCollectionGet: %d\n\n", status);

        if (status == PRO_TK_NO_ERROR)
        {
            status = ProCollectionTypeGet (collection, &coll_type);

            if (coll_type == PRO_COLLECTION_CRV_COLL)
            {
/*-----*\
Check the instructions assigned to the curve collection.
Exclude this collection if the instruction contains a type not
supported for this feature
\*-----*/
                status = ProCrvcollectionInstructionsGet (collection, &instrs);

                status = ProArraySizeGet (instrs, &size);

                for (i = 0; i < size; i++)
                {
                    status = ProCrvcollinstrTypeGet (instrs[i], &instr_type);

                    if (instr_type != PRO_CURV_COLL_ADD_ONE_INSTR &&
                        instr_type != PRO_CURV_COLL_TAN_INSTR &&
                        instr_type != PRO_CURV_COLL_TRIM_INSTR &&
                        instr_type != PRO_CURV_COLL_START_PNT_INSTR &&
                        instr_type != PRO_CURV_COLL_ADD_TANGENT_INSTR)
                        break;

/*-----*\
Check the references to ensure that the reference is an edge
which borders a sheetmetal face.
\*-----*/
                    status = ProCrvcollinstrReferencesGet (instrs[i], &refs);

                    status = ProArraySizeGet (refs, &refs_size);

                    invalid_ref = PRO_B_FALSE;
                    for (j = 0; j < refs_size; j++)
                    {
                        ProReferenceTypeGet (refs[j], &item.type);

                        if (item.type != PRO_EDGE)
                            break;

```

```

        ProReferenceOwnerGet (refs[j], &item.owner);
        ProReferenceIdGet (refs[j], &item.id);

        ProReferenceFree (refs[j]);

        if (!UserEdgeIsSmtFaceEdge (&item))
            invalid_ref = PRO_B_TRUE;
    }
    ProReferencearrayFree (refs);

    if (invalid_ref)
        break;

/*-----*\
If all instructions and references passed, the button should be active.
\*-----*/
    if (i == size -1)
        access_result = ACCESS_AVAILABLE;
    }
    ProCrvcollinstrArrayFree (instrs);
}
ProCollectionFree (&collection);
}

    else
{
    status = ProSelectionModelitemGet (sels [0], &item);

/*-----*\
    If the selected type is a single edge, it must border a smt face surface.
\*-----*/
    if (item.type == PRO_EDGE)
    {
        if (UserEdgeIsSmtFaceEdge (&item)) access_result = ACCESS_AVAILABLE;
    }
}

    if (should_free)
        ProSelectionarrayFree (sels);

    return access_result;
}

/*=====*\
FUNCTION: UserSmtFlgWallCreate_TestMdlTree
PURPOSE: Access function for the model tree popup menu button.
\*=====*/
uiCmdAccessState UserSmtFlgWallCreate_TestMdlTree (uiCmdAccessMode mode)
{
    return UserSmtFlgWallCreate_TestLow (NULL, PM);
}

```

```

}

/*=====*\
FUNCTION: UserSmtFlgWallCreate_TestPM
PURPOSE: Access function for the graphics window popup menu button.
\*=====*/
uiCmdAccessState UserSmtFlgWallCreate_TestPM (uiCmdCmdId id,
        ProAppData data,
        ProSelection* sels)
{
    return UserSmtFlgWallCreate_TestLow (sels, PM);
}

```

Sheetmetal Extruded Wall Features

An extruded wall is drawn as a cross-section extruded in the specified direction. You can specify the wall thickness, generate bends, and assign a development length calculation to the wall.

In Pro/TOOLKIT an extruded wall shares the same element tree as the basic Extrude feature. The element tree should include some of the following sheetmetal-specific elements to generate a sheetmetal extruded wall feature:

- **PRO_E_STD_SMT_THICKNESS**—Has a double value that specifies the wall thickness. If this feature is not the first wall feature in the part, the thickness value is irrelevant and can be 0.0. The feature inherits the thickness of the first wall feature. This element is not required and cannot be modified if the sheetmetal thickness parameter is already assigned in the model.
- **PRO_E_STD_SMT_SWAP_DRV_SIDE**—Specifies sheet metal swap sides to switch sides of driving and offset sides.
- **PRO_E_SMT_WALL_SHARPS_TO_BENDS**—Converts any sharp edges in the section to appropriate bends.
- **PRO_E_SMT_FILLETS**—Refer to the section Feature Element Tree for the Sheetmetal Flat Wall Feature for the description of the element.
- **PRO_E_SMT_DEV_LEN_CALCULATION**—Refer to the section Feature Element Tree for the Sheetmetal Flat Wall Feature for the description of the element.

For details on the basic extrude Feature, refer to the chapter Element Trees: Extrude and Revolve.

Sheetmetal Cut Features

An extruded sheetmetal cut removes material from the walls it encounters.

In Pro/TOOLKIT a sheetmetal cut feature shares the same element tree as the basic extrude feature. The element tree should include some of the following sheetmetal cut-specific elements to generate a sheetmetal cut feature:

- **PRO_E_IS_SMT_CUT**—If true this feature is a sheet metal cut, otherwise it is a solid cut. It is applicable only in sheet metal parts.
- **PRO_E_SMT_CUT_NORMAL_DIR**—This element defines to which surface (green or white) the section projection will be normal.

Punch Form Feature

A Punch Form is a sheetmetal wall molded by a template (reference part). Merging the geometry of a reference part with the sheetmetal part creates the Punch Form feature.

Feature Element Tree for the Punch Form Feature

The element tree for the Punch Form feature is documented in the header file *ProSmtForm.h*. The following figure shows the feature element tree structure:

Figure 52-11: Feature Element Tree for Punch Form Feature

```

PRO_E_FEATURE_TREE
|
|-PRO_E_FEATURE_TYPE
|-PRO_E_STD_FEATURE_NAME
|-PRO_E_GMRG_SMT_TYPE
|-PRO_E_GMRG_FEAT_TYPE
|-PRO_E_DSF_REF_MDL
|   |--PRO_E_DSF_SEL_REF_MDL
|-PRO_E_COMP_PLACE_INTERFACE
|-PRO_E_COMPONENT_CONSTRAINTS
|-PRO_E_GMRG_VARIED_ITEMS
|-PRO_E_DSF_DEPENDENCY
|-PRO_E_FORM_PUNCH_SIDE
|-PRO_E_STD_SURF_COLLECTION_APPL
|-PRO_E_FORM_TOOL_CSYS
|-PRO_E_FORM_TOOL_NAME
|-PRO_E_GMRG_FORM_AUTO_ROUNDS
|-PRO_E_SMT_FILLET_INTERSECT
|   |--PRO_E_SMT_FILLET_RADIUS_USEFLAG
|   |--PRO_E_SMT_FILLET_RADIUS_SIDE
|   |--PRO_E_SMT_FILLET_RADIUS_VALUE

```

Production Applications:
Sheetmetal

Figure 52-12: PRO_SMT_SURF_FACE

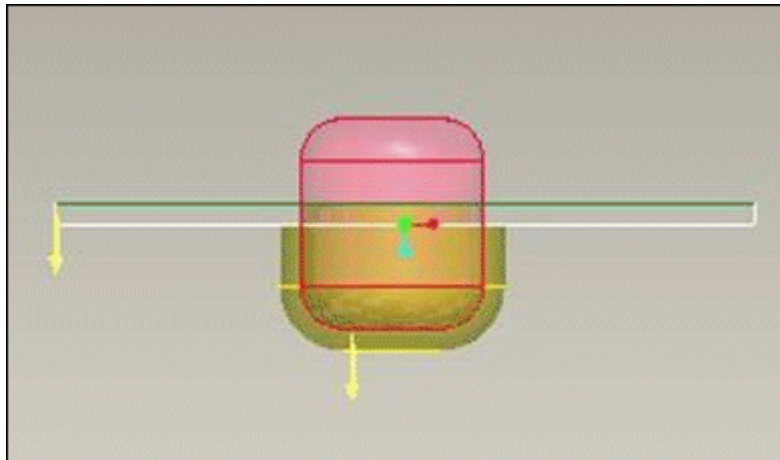
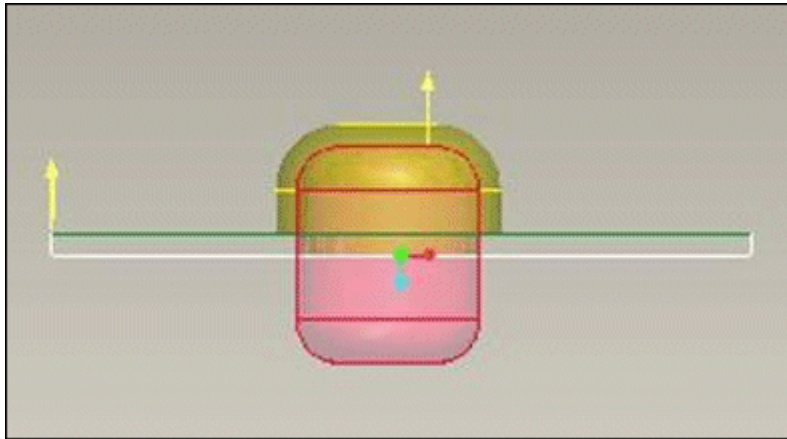


Figure 52-13: PRO_SMT_SURF_OFFSET



The elements in this tree are as follows:

- PRO_E_FEATURE_TYPE—Specifies the feature type and should be PRO_FEAT_GEN_MERGE.
- PRO_E_STD_FEATURE_NAME—Specifies the name of the feature.
- PRO_E_GMRG_SMT_TYPE—Specifies the type of sheetmetal feature. The values for this element are specified by the enumerated type *ProGenMergeSmtType*. It should be PRO_GEN_MERGE_SMT_TYPE_FORM for a Punch Form feature and PRO_GEN_MERGE_SMT_TYPE_NOT_SMT for a Merge, Inheritance, or any other type of General Merge feature.
- PRO_E_GMRG_FEAT_TYPE—Specifies the type of General Merge feature. The types applicable for the Punch Form feature depend upon the value of the PRO_E_DSF_DEPENDENCY element. The valid combinations for the Punch Form feature are:
 - The General Merge feature is of the type PRO_GEN_MERGE_TYPE_MERGE and the element PRO_E_DSF_DEPENDENCY is set to PRO_DSF_DEPENDENT.
 - The General Merge feature is of the type PRO_GEN_MERGE_TYPE_INHERITANCE and the element PRO_E_DSF_DEPENDENCY is set to PRO_DSF_INDEPENDENT.

For more information on the types of General Merge features, refer the “General Merge (Merge, Cutout and Inheritance Feature)” section in the “Assembly: Data Sharing Features” chapter.

- PRO_E_DSF_REF_MDL—Specifies the punch model used to create the Punch Form feature. It consists of the following element:
 - PRO_E_DSF_SEL_REF_MDL—Specifies the selected punch model.
 - PRO_E_COMP_PLACE_INTERFACE—Specifies the assembly component interfaces used to define the placement of the punch model in the Sheetmetal geometry. For more information on the elements contained by the placement interfaces element, refer the “Placement via Interface” section in the “Assembly: Assembling Components” chapter.
 - PRO_E_COMPONENT_CONSTRAINTS—Specifies the assembly component constraints used to define the placement of the punch model in the sheetmetal geometry. For more information on the component constraints elements, refer the “Placement Constraints” section in the “Assembly: Assembling Components” chapter.
- Note:** The placement by coordinate system option in the Pro/ENGINEER user interface for the Punch Form feature is not available via Pro/TOOLKIT. To place the punch model in the sheetmetal geometry using a coordinate system, define a coordinate system feature in the sheetmetal model and use it for placement.
- PRO_E_GMRG_VARIED_ITEMS—Specifies a pointer element that defines the Inheritance feature varied items and their values. This element is available only when the Punch Form feature is of the type PRO_GEN_MERGE_TYPE_INHERITANCE. For more information on this element, refer the “Inheritance Feature and Flexible Component Variant Items” section in the “Assembly: Data Sharing Features” chapter.
 - PRO_E_DSF_DEPENDENCY—Specifies the dependency flag. The values for this element, specified by the enumerated type *ProDsfDependency*, are as follows:

- PRO_DSF_DEPENDENT specifies that the geometry of the Punch Form feature depends upon the geometry of the punch model used during feature creation. The Punch Form feature reflects all the changes made in the punch model. This value is available only when the Punch Form feature is of the type PRO_GEN_MERGE_TYPE_MERGE.
- PRO_DSF_INDEPENDENT specifies that the geometry of the Punch Form feature is independent of the geometry of the punch model used during feature creation. If you update the punch model, the Punch Form feature does not change. This value is available only when the Punch Form feature is of the type PRO_GEN_MERGE_TYPE_INHERITANCE.
- PRO_E_FORM_PUNCH_SIDE—Specifies the punch direction. The punch direction specifies a side of a wall from which the punch model penetrates the sheetmetal geometry. The values for this element are specified by the enumerated type *ProSmtSurfType*. The surface types are as follows:
 - PRO_SMT_SURF_FACE—The punch model moves in a direction from the green side to the white side of the sheetmetal model. Refer to the Figure 52-12: PRO_SMT_SURF_FACE.
 - PRO_SMT_SURF_OFFSET—The punch model moves in a direction from the white side to the green side of the sheetmetal model. Refer to the Figure 52-13: PRO_SMT_SURF_OFFSET.
- PRO_E_STD_SURF_COLLECTION_APPL—Specifies a collection of surfaces to be excluded from the Punch model during feature creation.
- PRO_E_FORM_TOOL_CSYS—Specifies the reference coordinate system used during the manufacturing process. The coordinate system in the Punch model is used by default.
- PRO_E_FORM_TOOL_NAME—Specifies the name of the manufacturing tool used to create the Punch model. The name specified by the SMT_FORM_TOOL_NAME parameter in the Punch model is used by default.
- PRO_E_GMRG_FORM_AUTO_ROUNDS—Specifies the *ProBoolean* option to round the non-placement sharp edges (that do not lie on the placement surface). This optional element uses a constant outside radius of value equal to the thickness of the original sheetmetal part.

- **PRO_E_SMT_FILLET_INTERSECT**—Specifies the option to round the placement sharp edges (that lie on the placement surface and intersect the sheetmetal geometry). This optional element consists of the following elements:
 - **PRO_E_SMT_FILLET_RADIUS_USEFLAG**—Specifies whether a fillet radius is used.
 - **PRO_E_SMT_FILLET_RADIUS_SIDE**—Specifies the radius direction (outside or inside). The values for this element, specified by the enumerated type *ProSmdRadType*, are as follows:
 - **PRO_BEND_RAD_OUTSIDE**—The radius is applied to the outside of the sheetmetal geometry.
 - **PRO_BEND_RAD_INSIDE**—The radius is applied to the inside of the sheetmetal geometry.
 - **PRO_E_SMT_FILLET_RADIUS_VALUE**—Specifies the radius value.

Creating a Punch Form Feature

Function Introduced

- **ProFeatureCreate()**

Use the function **ProFeatureCreate()** to create a Punch Form feature based on the element tree definition. For more information about **ProFeatureCreate()**, refer the “Overview of Feature Creation” section in the “Element Trees: Principles of Feature Creation” chapter.

Redefining a Punch Form Feature

Function Introduced

- **ProFeatureRedefine()**

Use the function **ProFeatureRedefine()** to redefine a Punch Form feature based on the changes made in the element tree. For more information about **ProFeatureRedefine()**, refer the “Feature Redefine” section in the “Element Trees: Principles of Feature Creation” chapter.

Accessing a Punch Form Feature

Function Introduced

- **ProFeatureElemtreeExtract()**

Use the function **ProFeatureElemtreeExtract()** to create a feature element tree that describes the contents of a Punch Form feature and to retrieve the element tree description of the feature. For more information about **ProFeatureElemtreeExtract()**, refer the “Feature Inquiry” section in the “Element Trees: Principles of Feature Creation” chapter.

Quilt Form Feature

The Quilt Form feature molds the sheetmetal wall using a referenced closed or open datum quilt.

Feature Element Tree for the Quilt Form Feature

The element tree for the Quilt Form feature is documented in the header file *ProSmtPunchQuilt.h*. The following figure shows the feature element tree structure:

Figure 52-14: Feature Element Tree for Quilt Form Feature

```

PRO_E_FEATURE_TREE
|
|--PRO_E_FEATURE_TYPE
|
|--PRO_E_STD_FEATURE_NAME
|
|--PRO_E_FORM_PUNCH_QUILT
|
|--PRO_E_FORM_PUNCH_SIDE
|
|--PRO_E_FORM_PUNCH_MATERIAL_SIDE
|
|--PRO_E_STD_SURF_COLLECTION_APPL
|
|--PRO_E_SMT_FILLET_INTERSECT
|   |
|   |--PRO_E_SMT_FILLET_RADIUS_USEFLAG
|   |
|   |--PRO_E_SMT_FILLET_RADIUS_SIDE
|   |
|   |--PRO_E_SMT_FILLET_RADIUS_VALUE
|
|--PRO_E_SMT_FILLET_QUILT
|   |
|   |--PRO_E_SMT_FILLET_RADIUS_USEFLAG
|   |
|   |--PRO_E_SMT_FILLET_RADIUS_SIDE
|   |
|   |--PRO_E_SMT_FILLET_RADIUS_VALUE
|
|--PRO_E_FORM_PUNCH_HIDE_QUILT

```

The elements in this tree are as follows:

- PRO_E_FEATURE_TYPE—Specifies the feature type and should be PRO_FEAT_PUNCH_QUILT.
- PRO_E_STD_FEATURE_NAME—Specifies the name of the feature.
- PRO_E_FORM_PUNCH_QUILT—Specifies the referenced datum quilt. The geometry of the referenced quilt is merged with the sheetmetal part.

- **PRO_E_FORM_PUNCH_SIDE**—Specifies the punch direction. The punch direction specifies a side of a wall from which the punch model penetrates the sheetmetal geometry. The values for this element are specified by the enumerated type *ProSmtSurfType*. The surface types are as follows:
 - **PRO_SMT_SURF_FACE**—The punch model moves in a direction from the green side to the white side of the sheetmetal model. Refer to the Figure 52-12: **PRO_SMT_SURF_FACE**.
 - **PRO_SMT_SURF_OFFSET**—The punch model moves in a direction from the white side to the green side of the sheetmetal model. Refer to the Figure 52-13: **PRO_SMT_SURF_OFFSET**.
- **PRO_E_FORM_PUNCH_MATERIAL_SIDE**—Specifies whether the resultant sheetmetal geometry lies inside or outside the referenced quilt after the feature is created. The geometry is placed inside or outside the referenced quilt using the thickness value of the original sheetmetal part. The values for this element, specified by the enumerated type *ProSmdPunchMatSide*, are as follows:
 - **PRO_SMT_PUNCH_MAT_INSIDE**
 - **PRO_SMT_PUNCH_MAT_OUTSIDE**
- **PRO_E_STD_SURF_COLLECTION_APPL**—Specifies a collection of surfaces to be excluded from the referenced quilt when the feature is created.
- **PRO_E_SMT_FILLET_INTERSECT**—Specifies the set of fillets that are added to the contours created by the intersection of the referenced quilt with the sheetmetal part. This element consists of the following elements:
 - **PRO_E_SMT_FILLET_RADIUS_USEFLAG**—Specifies whether a fillet radius is used.
 - **PRO_E_SMT_FILLET_RADIUS_SIDE**—Specifies the radius direction (outside or inside). The values for this element, specified by the enumerated type *ProSmdRadType*, are as follows:
 - **PRO_BEND_RAD_OUTSIDE**—The radius is applied to the outside of the sheetmetal geometry.
 - **PRO_BEND_RAD_INSIDE**—The radius is applied to the inside of the sheetmetal geometry.
 - **PRO_E_SMT_FILLET_RADIUS_VALUE**—Specifies the radius value.

- **PRO_E_SMT_FILLET_QUILT**—Specifies the set of fillets that are added to sharps (edges between non-tangent geometries) on the referenced quilt. This element consists of the same set of elements as **PRO_E_SMT_FILLET_INTERSECT**.
- **PRO_E_FORM_PUNCH_HIDE_QUILT**—Specifies whether the referenced quilt will be hidden in the feature. The values for this element, specified by the enumerated type *ProSmdPunchHideQuilt*, are as follows:
 - **PRO_SMT_PUNCH_HIDE_ORIGINAL**
 - **PRO_SMT_PUNCH_KEEP_ORIGINAL**

Creating a Quilt Form Feature

Function Introduced

- **ProFeatureCreate()**

Use the function **ProFeatureCreate()** to create a Quilt Form feature based on the element tree definition. For more information about **ProFeatureCreate()**, refer the “Overview of Feature Creation” section in the “Element Trees: Principles of Feature Creation” chapter.

Redefining a Quilt Form Feature

Function Introduced

- **ProFeatureRedefine()**

Use the function **ProFeatureRedefine()** to redefine a Quilt Form feature based on the changes made in the element tree. For more information about **ProFeatureRedefine()**, refer the “Feature Redefine” section in the “Element Trees: Principles of Feature Creation” chapter.

Accessing a Quilt Form Feature

Function Introduced

- **ProFeatureElemtreeExtract()**

Use the function **ProFeatureElemtreeExtract()** to create a feature element tree that describes the contents of a Quilt Form feature and to retrieve the element tree description of the feature. For more information about **ProFeatureElemtreeExtract()**, refer the “Feature Inquiry” section in the “Element Trees: Principles of Feature Creation” chapter.

Production Applications: Manufacturing

This chapter describes the Pro/TOOLKIT functions for manufacturing operations. Familiarity with Pro/NC functions simplifies the use of these manufacturing functions.

Topic	Page
Manufacturing Models	53 - 2
Creating a Manufacturing Model	53 - 3
Analyzing a Manufacturing Model	53 - 3
Creating Manufacturing Objects	53 - 9
Analyzing Manufacturing Features	53 - 45

Manufacturing Models

Functions Introduced:

- **ProMfgAssemGet()**
- **ProMfgTypeGet()**
- **ProMfgCreate()**

You can use the function **ProSolidFeatVisit()** to visit all the components in a manufacturing model. However, this function requires a *ProSolid* (or one of its instances, *ProPart* or *ProAssembly*) handle to the model as input. The function **ProMfgAssemGet()** outputs a *ProAssembly* handle to the top-level assembly in the manufacturing model, given its *ProMfg* handle. This assembly handle can then be passed to **ProSolidFeatVisit()**.

Manufacturing models, like other models in Pro/ENGINEER, are uniquely identified by name and type. However, there are several different varieties of manufacturing models. For example, part machining, assembly machining, sheetmetal manufacturing, and mold manufacturing are all types of manufacturing models. The *ProMfg* object is a general purpose, opaque handle used to represent any of the different manufacturing model varieties. To distinguish between the different types of manufacturing model, there are manufacturing subtypes. The complete list of subtypes is as follows:

- PRO_MFGTYPE_MACH_PART
- PRO_MFGTYPE_MACH_ASSEM
- PRO_MFGTYPE_SHEET_METAL
- PRO_MFGTYPE_MOLD
- PRO_MFGTYPE_CAST
- PRO_MFGTYPE_CMM

The function **ProMfgTypeGet()** outputs the subtype, given the *ProMfg* handle to the manufacturing object.

Creating a Manufacturing Model

Function Introduced

- **ProMfgCreate()**

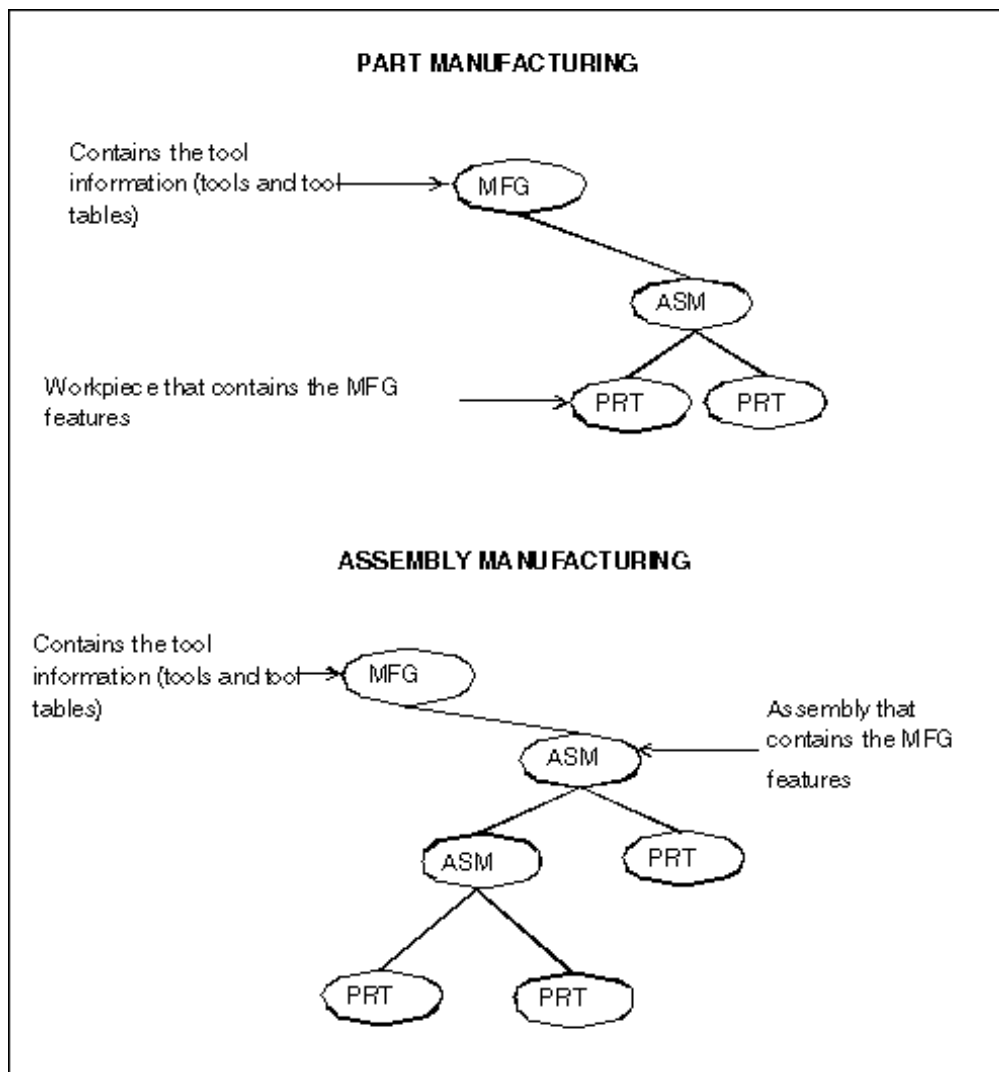
The function **ProMfgCreate()** outputs an initialized *ProMfg* object handle, given the model name, manufacturing subtype, and name of the reference model. For part machining, the reference model is the design part; for sheetmetal manufacturing, this should be the sheetmetal workpiece. For all other subtypes, the reference model argument is ignored.

Analyzing a Manufacturing Model

Pro/NC has two modes of operation—part and assembly manufacturing. In both cases, the top-level model is an assembly that contains the description of the tools. The following diagram shows the hierarchy of part and assembly manufacturing models.

Production Applications:
Manufacturing

Figure 53-1: Part and Assembly Manufacturing Model Hierarchy



In part manufacturing, the storage solid is a part that represents the workpiece or stock, and the design piece is another component at the same level. In assembly manufacturing, the storage solid is the actual assembly representing the design model. The workpiece can be at any level inside this assembly. In both types of manufacturing, the manufacturing operations are described as features of the storage solid.

The important tasks for Pro/TOOLKIT are to traverse the manufacturing assembly components, identify the storage solid that contains the manufacturing operations as its features, and list the manufacturing tools.

This section contains the following subsections:

- Traversing Manufacturing Components
- Identifying the Storage Solid
- Visiting Manufacturing Tools

Traversing Manufacturing Components

Function Introduced:

- **ProAsmcompTypeGet()**

You can visit the components in a manufacturing assembly using the same functions that enable you to visit the components of a regular assembly. For a full description of these functions, see the 'Assembly: Basic Assembly Access' chapter.

The components within a manufacturing assembly perform a variety of different roles. The function **ProAsmcompTypeGet()** provides the role of any model under a manufacturing assembly. The possible roles are as follows:

- PRO_ASM_COMP_TYPE_NONE—A regular component (no special manufacturing role)
- PRO_ASM_COMP_TYPE_WORKPIECE—A workpiece
- PRO_ASM_COMP_TYPE_REF_MODEL—A reference model
- PRO_ASM_COMP_TYPE_FIXTURE—A fixture
- PRO_ASM_COMP_TYPE_MOLD_BASE—A mold base
- PRO_ASM_COMP_TYPE_MOLD_COMP—A mold component
- PRO_ASM_COMP_TYPE_MOLD_ASSEM—A mold assembly
- PRO_ASM_COMP_TYPE_GEN_ASSEM—A general assembly
- PRO_ASM_COMP_TYPE_CAST_ASSEM—A cast assembly
- PRO_ASM_COMP_TYPE_DIE_BLOCK—A die block
- PRO_ASM_COMP_TYPE_DIE_COMP—A die component
- PRO_ASM_COMP_TYPE_SAND_CORE—A sand core
- PRO_ASM_COMP_TYPE_CAST_RESULT—A cast result

Identifying the Storage Solid

Functions Introduced:

- **ProMfgSolidGet()**
- **ProMfgFeatureOwnerGet()**

Another important task in using Pro/TOOLKIT for accessing Pro/NC models is to find the storage solid inside the manufacturing model. This model is important because the manufacturing operations—workcells, NC sequences, and so on—are represented as features within it.

Manufacturing features are treated like all other features in Pro/TOOLKIT. This enables you to search for NC sequences as you would for solid features. For example, to visit all the workcells of a manufacturing solid, you can use the **ProSolidFeatVisit()** function and filter out any model items not of type PRO_E_WCELL.

The function **ProMfgSolidGet()** returns the handle to the storage solid, and **ProMfgFeatureOwnerGet()** returns the component path to the solid from the top-level assembly.

Example 1: Identifying Workcell Features of a Pro/NC Model

The following example shows how to identify workcell features.

```
/*=====*\
FUNCTION: UserMfgWcellCollect
PURPOSE:  Add a workcell feature pointer to an array of workcells.
\*=====*/
ProError UserMfgWcellCollect (
    ProFeature *wcell,
    ProError    err,
    ProAppData  wcell_array)
{
    err = ProArrayObjectAdd((ProArray*)wcell_array,
        PRO_VALUE_UNUSED, 1, (void*)wcell);
    return err;
}
/*=====*\
FUNCTION: UserFeatNonWcellFilter
PURPOSE:  Filter out non-workcell features.
\*=====*/
ProError UserFeatNonWcellFilter (
    ProFeature *feature,
    ProAppData  unused)
{
    ProError    status = PRO_TK_NO_ERROR;
    ProFeattype feat_type;
```



```

    status = ProFeatureTypeGet (feature, &feat_type);

    if (status != PRO_TK_NO_ERROR || feat_type != PRO_FEAT_WORKCELL)
        return PRO_TK_CONTINUE;    /* Don't visit this item. */
    else
        return PRO_TK_NO_ERROR;    /* Do visit this one! */
}
/*=====*\
FUNCTION: UserMfgWcellList
PURPOSE:  Collect an array of workcells that belong to a
          manufacturing model.
\*=====*/
int UserMfgWcellList (
    ProAppData    mfg_model,
    int           dummy)
{
    ProError      err = PRO_TK_NO_ERROR;
    ProFeature    *wcell_array = (ProFeature*)NULL;
    ProSolid      mfg_solid;
    int           num_wcells = -1;

    err = ProArrayAlloc (0, sizeof (ProFeature*), 1,
        (ProArray*)&wcell_array);

    if (err == PRO_TK_NO_ERROR)
    {
        err = ProMfgSolidGet (*(ProMfg*)mfg_model, &mfg_solid);

        if (err == PRO_TK_NO_ERROR)
            err = ProSolidFeatVisit (mfg_solid,
                UserMfgWcellCollect, UserFeatNonWcellFilter,
                (ProAppData)&wcell_array);

        if (err == PRO_TK_NO_ERROR)
        {
            err = ProArraySizeGet ((ProArray)wcell_array, &num_wcells);

            if (err == PRO_TK_NO_ERROR)
                err = ProMessageDisplay (MSG_FIL,
                    "USER There are %0d workcells in this model",
                    &num_wcells);
        }

        err = ProArrayFree ((ProArray*)&wcell_array);
    }
    if (num_wcells == -1)
        err = ProMessageDisplay (MSG_FIL,
            "USER Error determining the number of workcells");

    return ((int)err);
}

```

Visiting Manufacturing Tools

Functions Introduced:

- **ProMfgToolVisit()**
- **ProToolTypeGet()**
- **ProToolModelGet()**
- **ProToolParamGet()**
- **ProToolAttributesGet()**
- **ProToolVerify()**

Tools are not stored as features of the manufacturing solid, nor as components in the assembly; they are special objects within the top-level manufacturing model, so they need their own visit function. This function is **ProMfgToolVisit()**, which has the same form as other visit functions in Pro/TOOLKIT, except it does not offer the option of a user-defined filter function. You call the action function with an input argument of type *ProTool* to identify the tool.

The other functions analyze the tool being visited.

The function **ProToolTypeGet()** returns the tool type. Examples of the types are as follows:

- PRO_TOOL_DRILL
- PRO_TOOL_TURN
- PRO_TOOL_SAW
- PRO_TOOL_MILL

The **ProToolModelGet()** function outputs the model name and type of a tool, given its *ProTool* handle.

The function **ProToolParamGet()** retrieves the value of a specified tool parameter. This yields a value in terms of the type *ProParamvalue* (see the 'Core: Parameters' chapter for more information).

The function **ProToolAttributesGet()** provides the current setting of several Boolean attributes of the tool in the form of an integer bitmap. Currently, the attributes define whether a solid tool is to be by reference or by copy. See the section 'Creating NC Sequences' for more information on creating tools.

The function **ProToolVerify()** returns a Boolean showing whether a specified tool handle corresponds to an existing tool.

Creating Manufacturing Objects

This section explains how to create all types of manufacturing feature. This section assumes you have an understanding of element trees for feature creation and Pro/NC. For an introduction to element trees, see 'Element Trees: Principles of Feature Creation'.

Note: You must have a Pro/NC license to create manufacturing features using Pro/TOOLKIT.

An important principle of creating manufacturing features is that all the elements required to be defined interactively must also be defined when you create that feature using Pro/TOOLKIT.

As with creating solid features, manufacturing features use element trees to define the feature before you call

ProFeatureCreate() to create the feature. However, the method of creating tools is slightly different, as described in the following sections.

Creating Tools

Functions Introduced:

- **ProToolinputAlloc()**
- **ProToolinputTypeSet()**
- **ProToolElemParamAdd()**
- **ProToolElemModelSet()**
- **ProToolinputElemAdd()**
- **ProToolInit()**
- **ProToolCreate()**
- **ProToolinputFree()**

In Pro/ENGINEER, and therefore in Pro/TOOLKIT, tools are not features, and must be created in a slightly different manner for solid and manufacturing features.

Tool creation involves initializing an input structure using a call to **ProToolinputAlloc()**.

You set the tool type (for example, center drill or ream) directly in the input structure using the function **ProToolinputTypeSet()**.

You can then add tool elements to this input structure using a three-step process. First, initialize each element using the function **ProElementAlloc()**. Next, add data to this element using an element-specific function. Finally, add the element to the tool input structure using **ProToolinputElemAdd()**.

As in Pro/ENGINEER, tools can be defined by parameter or by model. To add a parameter to a tool, first allocate the space for a parameter element using a call such as this:

```
status = ProElementAlloc (PRO_E_PARAM, &element);
```

Next, add the parameter to the element using the function **ProToolElemParamAdd()**, then add the element itself to the input structure using the function **ProToolinputElemAdd()**.

The following table lists the parameters required to be defined for each turning tool.

Parameter	Turn	Turn Groove
NOSE_RADIUS	•	•
TOOL_WIDTH	•	•
SIDE_WIDTH	•	
LENGTH	•	•
SIDE_ANGLE	•	•
END_ANGLE	•	•
GAUGE_X_LENGTH	•	•
GAUGE_Z_LENGTH	•	•
TOOL_MATERIAL	•	•
HOLDER_TYPE	•	

The following table lists the parameters required to be defined for milling tools.

Parameter	Mill	Side Mill	Thread Mill	Groove
CUTTER_DIAM	•	•	•	•
CORNER_RADIUS	•	•		•
CUTTER_WIDTH		•		
SHANK_DIAM		•		
LENGTH	•	•	•	•
INSERT_LENGTH			•	
END_OFFSET			•	
SIDE_ANGLE	•	•		
GAUGE_X_LENGTH	•	•		
GAUGE_Z_LENGTH	•	•		•
NUM_OF_TEETH	•	•	•	
TOOL_MATERIAL	•	•		

The following table lists the parameters required to be defined for auxiliary and contouring tools.

Parameter	Auxiliary	Contouring
CUTTER_DIAM	•	•
LENGTH	•	•

The following table lists the parameters required to be defined for holmaking tools.

Parameter	Drill	Csink	Tap	Ream	Center Drill	Bore	Back Spot
CUTTER_DIAM	•	•	•	•	•	•	•
POINT_DIAMETER		•	•				
DRILL_DIAMETER					•		
BODY_DIAMETER							•
LENGTH	•	•	•	•	•	•	•
CHAMFER_LENGTH			•				
DRILL_LENGTH					•		
INSERT_LENGTH							•
TIP_OFFSET		•					
GAUGE_OFFSET		•					
CUTTING_OFFSET							•
SIDE_ANGLE				•			
POINT_ANGLE	•	•			•		
CSINK_ANGLE					•		•
GAUGE_X_LENGTH	•	•	•	•	•	•	•
GAUGE_Z_LENGTH	•	•	•	•	•	•	•
TOOL_MATERIAL	•	•	•	•	•	•	•

For more information, see the *Pro/NC User's Guide*.

Creating a tool using a tool model is similar to the previous process. First, allocate space for an element of type PRO_E_TOOL_MODEL. Set the model in the element using the function **ProToolElemModelSet()**, then add it to the input structure using **ProToolinputElemAdd()**. As in Pro/NC, you must specify the required number of dimensions within the tool model.

Creating the tool requires two steps. First, initialize a tool handle using **ProToolInit()**. This creates a tool identifier that uniquely defines the tool and is used to reference that tool within the manufacturing model. You pass this identifier, together with the completed input structure, to the function **ProToolCreate()**, which actually creates the tool.

Once the tool has been created, release the memory used by the tool input structure and its associated elements using the function **ProToolinputFree()**.

The following table shows the elements required for tool creation. In this table, the Value column specifies whether the element is required (R) or optional (O).

Element	Description	Value
Name	The name used to identify the tool	R
Type	Mill, drill, and so on	R
Parameters (for parameter-driven tools only)	Tool parameters	R
Model (for solid tools only)	Model that represents the tool	R

Example 2: Creating a Tool from a Solid Model

The following example code shows how to create a tool from solid model.

```

/*=====*\
FUNCTION: UserSolidToolCreate
PURPOSE: Create a tool from a tool part.
\*=====*/
ProErr UserSolidToolCreate (
    ProMfg          mfg_model,
    ProToolType     tool_type,
    wchar_t         *tool_id,
    wchar_t         *part_name,
    ProTool         *tool)
{
    ProToolElem     tool_elem;
    ProToolinputPtr tool_input;
    ProMdl          tool_part;
    ProErr          status;
    ProErrorlist    errors;

    status = ProMdlRetrieve (part_name, PRO_PART, &tool_part);
    if (status != PRO_TK_NO_ERROR)
        return status;

```

```

    status = ProToolinputAlloc (&tool_input);
    if (status == PRO_TK_NO_ERROR)
    {
/*-----*\
        Set the tool type.
/*-----*\
        status = ProToolinputTypeSet (tool_input, tool_type);
        if (status == PRO_TK_NO_ERROR)
            return status;
/*-----*\
        Allocate the space for the tool model and to the
        tool_input.
/*-----*\
        status = ProElementAlloc (PRO_E_TOOL_MODEL, &tool_elem);

        if (status == PRO_TK_NO_ERROR)
        {
            status = ProToolElemModelSet (tool_elem, &tool_part);
        }

        if (status == PRO_TK_NO_ERROR)
        {
            status = ProToolinputElemAdd (tool_input, tool_elem);
        }
/*-----*\
        Create the tool.
/*-----*\
        if (status == PRO_TK_NO_ERROR)
        {
            status = ProToolInit (tool_id, mfg_model, tool);
            if (status == PRO_TK_NO_ERROR)
            {
                status = ProToolCreate (tool, tool_input, &errors);
            }
        }
/*-----*\
        Free the input structure
/*-----*\
        status = ProToolinputFree (&tool_input);

    }
    return status;
}

```


Example 3: Creating a Parameter-Driven Tool

The following example code shows a program that creates a drilling tool from parameters.

```
/*=====*\
FUNCTION: UserParamToolCreate
PURPOSE: Create a simple drilling tool from parameters.
\*=====*/
ProError UserParamToolCreate (
    ProMfg          mfg_model,
    ProToolType     tool_type,
    wchar_t         *tool_id,
    double          tool_diam,
    double          tool_length,
    double          point_angle,
    ProTool         *tool)
{
    #define NUM_PARAMS    3
    ProElement        tool_elem;
    ProToolinputPtr   tool_input;
    ProError          status;
    ProErrorlist      errors;
    ProParamvalue     param_val;

    char    par_name[NUM_PARAMS][PRO_NAME_SIZE] =
                {"CUTTER_DIAM", "LENGTH", "POINT_ANGLE"};
    double  par_vals[NUM_PARAMS];
    int     i;

    par_vals[0] = tool_diam;
    par_vals[1] = tool_length;
    par_vals[2] = point_angle;
/*-----*\
    Allocate the input structure.
\*-----*/
    status = ProToolinputAlloc (&tool_input);

    if (status == PRO_TK_NO_ERROR)
    {
/*-----*\
        Set the type.
\*-----*/
        status = ProToolinputTypeSet (tool_input, tool_type);

        if (status != PRO_TK_NO_ERROR)
            return status;
/*-----*\
        Add tool parameters
\*-----*/
        status = ProElementAlloc (PRO_E_PARAMS, &tool_elem);
```

```

    param_val.type = PRO_PARAM_DOUBLE;

    for ( i = 0; i < NUM_PARAMS; i++)
    {
        param_val.value.d_val = par_vals[i];
        status = ProToolElemParamAdd (tool_elem, &param_val,
                                      par_name[i]);
    }

    status = ProToolinputElemAdd (tool_input, tool_elem);
/*-----*\
    Create the tool.
\*-----*/
    status = ProToolInit (tool_id, mfg_model, tool);

    if (status == PRO_TK_NO_ERROR)
    {
        status = ProToolCreate (tool, tool_input, &errors);
    }
/*-----*\
    Free the input structure.
\*-----*/
    status = ProToolinputFree (&tool_input);
}

return status;
}

```

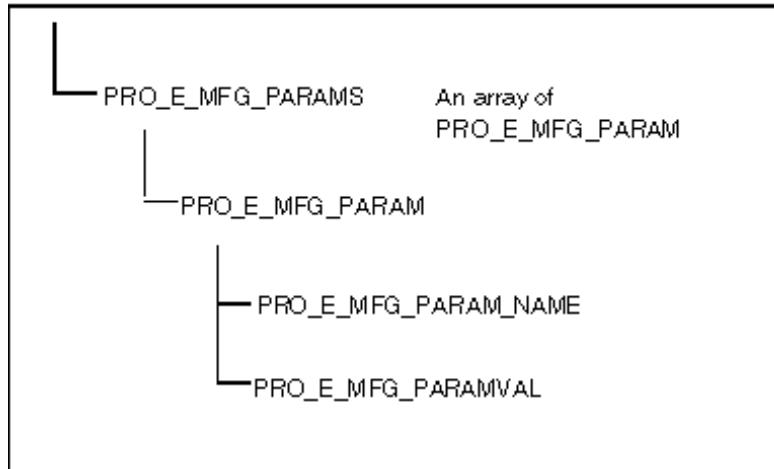
Manufacturing Parameters

In general, there are a number of parameters that are mandatory for a given tool type or NC sequence, and others that are optional. For example, a milling tool requires that its length and diameter be specified, while other parameters such as the number of teeth, or tool material are optional.

The addition of manufacturing parameters to both workcells and operations is optional.

The following figure shows a parameter element subtree.

Figure 53-2: Parameter Element Subtree



The process of creating a parameter element subtree is the same for workcells, operations, and NC sequences. First, allocate the space for the `PRO_E_MFG_PARAMS` array element. The simplest method of creating the tree is to delay adding the `PRO_E_MFG_PARAMS` element to its parent until you have fully defined the tree. As you define each `PRO_E_MFG_PARAM` element, add it to the `PRO_E_MFG_PARAMS` array using **ProElemtreeElementAdd()**. Use `NULL` for the element path as each `PRO_E_MFG_PARAM` element is added to the parameter element tree.

The `PRO_E_MFG_PARAM` element itself is a compound element and requires two children to be defined. One is the `PRO_E_MFG_PARAM_NAME` element, a string (*not* a wide string) that represents the parameter name to define. The other is a `PRO_E_MFG_PARAMVAL` element, which represents the value of the parameter. Depending on the context, this might be an integer, double, or wide string. For example, `CUT_FEED` is represented by a double, whereas `NUMBER_OF_ARC_PNTS` is an integer. For further explanation of manufacturing parameters, see the *Pro/NC User's Guide*.

Example 4: Creating a Parameter Tree

This example illustrates the creation of a parameter element tree.

```
/*=====*\
FUNCTION: UserMfgParamsSet
PURPOSE: Demonstrate how to set a few required NC sequence
         parameters.
\*=====*/
ProError UserMfgParamsSet (ProElement *mfg_params_elem)
{
    ProError      status = PRO_TK_NO_ERROR;
    ProElement     mfg_param_elem, name_elem, value_elem;
    typedef struct {char          param_name[PRO_NAME_SIZE];
                   double         param_value; } ParameterTable;
    ParameterTable seq_params[] = {"CUT_FEED",          16.0 },
                                   {"TOLERANCE",         0.0001},
                                   {"STEP_OVER",         4.0  },
                                   {"SPINDLE_SPEED",     550.0 },
                                   {"CLEAR_DIST",        v  1.0  }};

    int seq_param_size = sizeof(seq_params)/sizeof(ParameterTable), i;
    ProError UserElementBuild();

    for (i = 0; i < seq_param_size && status == PRO_TK_NO_ERROR; i++)
    {
/*-----*\
        Allocate the parent element.
\*-----*/
        {
            status = ProElementAlloc (PRO_E_MFG_PARAM,
                                     &mfg_param_elem);
        }

/*-----*\
        Add the name element to the parent.
\*-----*/
        if (status == PRO_TK_NO_ERROR)
        {
            status = ProElementAlloc (PRO_E_MFG_PARAM_NAME,
                                     &name_elem);
        }

        if (status == PRO_TK_NO_ERROR)
            status = UserElementBuild(PRO_VALUE_TYPE_STRING,
                                     seq_params[i].param_name, &name_elem);

        if (status == PRO_TK_NO_ERROR)
        {
            status = ProElementtreeElementAdd (mfg_param_elem, NULL,
                                               name_elem);
        }
    }
}
```

```

/*-----*\
    Add the value element to the parent.
\*-----*/
    if (status == PRO_TK_NO_ERROR)
    {
        status = ProElementAlloc(PRO_E_MFG_PARAMVAL,
                                &value_elem);
    }

    if (status == PRO_TK_NO_ERROR)
        status = UserElementBuild(PRO_VALUE_TYPE_DOUBLE,
                                &(seq_params[i].param_value), &value_elem);

    if (status == PRO_TK_NO_ERROR)
    {
        status = ProElementtreeElementAdd (mfg_param_elem, NULL,
                                value_elem);
    }
/*-----*\
    Add the parent to the tree.
\*-----*/
    if (status == PRO_TK_NO_ERROR)
    {
        status = ProElementtreeElementAdd (*mfg_params_elem, NULL,
                                mfg_param_elem);
    }
}

return status;
}
/*=====*\
FUNCTION: UserElementBuild
PURPOSE:  Given a value type and a value, return the element.
\*=====*/
ProError UserElementBuild (
    ProValueDataType data_type,
    void             *data,
    ProElement       *element)
{
    ProError      status = PRO_TK_NO_ERROR;
    ProValue      value;
    ProValueData  value_data;
    ProSelection  *s;
    int           *i;
    double        *d;
    char          *c;
    wchar_t       *w;

    value_data.type = data_type;
/*-----*\
    Based on the data type, set the correct union member.

```

```

\*-----*/
switch (data_type)
{
    case PRO_VALUE_TYPE_INT:
        i = (int*) data;
        value_data.v.i = *i;
        break;
    case PRO_VALUE_TYPE_DOUBLE:
        d = (double*) data;
        value_data.v.d = *d;
        break;
    case PRO_VALUE_TYPE_STRING:
        value_data.v.s = (char*) calloc (1, sizeof(ProCharName));
        if (value_data.v.s)
        {
            c = (char*) data;
            strcpy (value_data.v.s, c);
        }
        else
            status = PRO_TK_OUT_OF_MEMORY;
        break;
    case PRO_VALUE_TYPE_WSTRING:
        value_data.v.w = (wchar_t*) calloc (1, sizeof(ProLine));
        if (value_data.v.w)
        {
            w = (wchar_t*) data;
            ProUtilWstrcpy(value_data.v.w, w);
        }
        else
            status = PRO_TK_OUT_OF_MEMORY;
        break;
    case PRO_VALUE_TYPE_SELECTION:
        status = ProSelectionCopy(*((ProSelection*)data),
                                &(value_data.v.r));
        break;
    default:
        return PRO_TK_BAD_INPUTS;
}
/*-----*\
Allocate and set the element value.
\*-----*/
if (status == PRO_TK_NO_ERROR)
{
    status = ProValueAlloc (&value);
}

if (status == PRO_TK_NO_ERROR)
{
    status = ProValueDataSet (value, &value_data);
}

```

```

if (status == PRO_TK_NO_ERROR)
{
    status = ProElementValueSet (*element, value);
}

return status;
}

```

Using External Functions to Define Parameters in the Manufacturing Step Table

Pro/TOOLKIT provides the capability to define customized functions that are available from within the Relations user interface and can be used to define relations for a given item. This capability is described in the section 'Adding a Customized Function to the Relations Dialog in Pro/ENGINEER' in the chapter 'Core: Relations'. External relation functions can also be used within relations stored in steps in the Manufacturing Step Table.

External functions can be used to define new parameters for the members of the step table. For example, if a parameter 'sample_parameter' is defined as follows:

```
sample_parameter = protk_user_defined_function (list of arguments)
```

The definition suggests that the value of the parameter 'sample_parameter' is set as a result of the calculation done using the function 'protk_user_defined_function'.

The following steps are required to define new parameters using external relation functions:

1. Within the Pro/TOOLKIT application, register an appropriate external function using **ProRelationFunctionRegister()**.
2. In the user interface for the step table, define the relation calling the externally registered functions.

Pro/TOOLKIT relation functions called from an entry in the Step Table will have one of two owners for the relation set (*ProRelSet*) depending on the following cases:

Case 1: If the step has been applied

The owner will be the manufacturing feature created by this step table entry.

Case 2: If the step has not been applied

The owner will be an object whose type is PRO_NC_STEP_OBJECT.

Using external functions, it is possible to interact with an external database or application, in order to set the value of some parameters in the Process Table. Because relations may be reevaluated many times during a regeneration cycle, PTC recommends that the external functions contain some control that prevents reconnection to the external database on each invocation.

Note: If no Pro/TOOLKIT application has registered the needed external function, any relation using the function cannot be evaluated and is shown as an error. However, when defining Global Relations, even if an undefined function is encountered, the relation will pass the validation, but, the relation will not be applied to the steps because the function is not found.

Example 5: Using External Functions

This example illustrates how to use external functions for relation calculation.

```
#include <ProToolkit.h>
#include <ProRelSet.h>
#include <ProArray.h>
#include <math.h>

typedef struct
{
    double input_param;
    double output_param;
} PTAFExLookup;

/* Stores a table of input/output pairs so that we don't have to retrieve
them from a database for every recalculation */

static PTAFExLookup* lookup_table = NULL;
#define INPUT_EPSILON 0.01

/*=====*\
FUNCTION :   PTAFExCalcFeedCoeffFind()
PURPOSE  :   Check the cache to see if the coefficient has already been
              retrieved.
\*=====*/
PTAFExLookup* PTAFExCalcFeedCoeffFind (double input)
{
    int i, array_size;
    double delta;
    status = ProArraySizeGet (lookup_table, &array_size);
```



```

PT_TEST_LOG_SUCC ("ProArraySizeGet()");
if (status != PRO_TK_NO_ERROR)
    return (NULL);
for (i = 0; i < array_size; i++)

{
    delta = fabs (lookup_table[i].input_param - input);
    if ( delta < INPUT_EPSILON)
    {
        return (&lookup_table[i]);
    }
}
return (NULL);
}
/*=====*\
FUNCTION :   PTAFExCalcFeedGetCoeffFromDB()
PURPOSE :   Go to the database to get the needed coefficient.
\*=====*/
double PTAFExCalcFeedGetCoeffFromDB (double input)
{
/*-----*\
Typically, this function would connect to an external database to lookup
the needed coefficient. Rather than actually set this up here, this
example calculates the result from a mathematical formula instead.
\*-----*/
    double output = input * 2.2;
    return output;
}
/*=====*\
FUNCTION :   PTAFExCalcFeedItemAdd()
PURPOSE :   Cache the value of the input and output coefficients that
            we looked up.
\*=====*/
void PTAFExCalcFeedItemAdd (double input, double output)
{
    PTAFExLookup lookup;
    lookup.input_param = input;
    lookup.output_param = output;
    if (lookup_table == NULL)
    {
        status = ProArrayAlloc (1, sizeof (PTAFExLookup), 10,
        (ProArray*)&lookup_table);
        PT_TEST_LOG_SUCC ("ProArrayAlloc()");
        lookup_table [0].input_param = lookup.input_param;
        lookup_table [0].output_param = lookup.output_param;
    }
    else
    {
        status = ProArrayObjectAdd ((ProArray*)&lookup_table, -1, 1,
        &lookup);
        PT_TEST_LOG_SUCC ("ProArrayObjectAdd()");
    }
}

```

```

    }
    return;
}
/*=====*\
FUNCTION :   PTAFExCalculateFeedFunction()
PURPOSE  :   This the relation calculation function.
\*=====*/
ProError PTAFExCalculateFeedFunction (ProRelset* relset,
                                     ProMdl owner,
                                     char* function_name,
                                     ProParamvalue* inputs,
                                     ProAppData app_data,
                                     ProParamvalue* output)
{
    ProModelitem item;
    double input = inputs[0].value.d_val;
    double output_val;
    PTAFExLookup* lookup = NULL;
    ProParameter cutfeed_param, cutdia_param;
    ProParamvalue cutfeed_pvalue, cutdia_pvalue;
    double cut_feed, cut_dia;
    double coeff;
    status = ProRelsetToModelitem (relset, &item);
    PT_TEST_LOG ("ProRelsetToModelitem()", status,
                status != PRO_TK_NO_ERROR && status != PRO_TK_INVALID_ITEM);
/*-----*\
Obtain the value of the CUT_FEED_01 parameter.
\*-----*/
    if (status == PRO_TK_NO_ERROR)
    {
        status = ProParameterInit (&item, L"CUT_FEED_01", &cutfeed_param);
        PT_TEST_LOG_SUCC ("ProParameterInit");
    }
    if (status == PRO_TK_NO_ERROR)
    {
        status = ProParameterValueGet (&cutfeed_param, &cutfeed_pvalue);
        PT_TEST_LOG_SUCC ("ProParameterValueGet");

        cut_feed = cutfeed_pvalue.value.d_val;
    }
/*-----*\
Obtain the value of the CUT_DIAMETER_01 parameter.
\*-----*/
    if (status == PRO_TK_NO_ERROR)
    {
        status = ProParameterInit (&item, L"CUTTER_DIA_01",
                                   &cutdia_param);
        PT_TEST_LOG_SUCC ("ProParameterInit");
    }

    if (status == PRO_TK_NO_ERROR)

```

```

{
    status = ProParameterValueGet ( &cutdia_param, &cutdia_pvalue);
    PT_TEST_LOG_SUCC ("ProParameterValueGet");

    cut_dia = cutdia_pvalue.value.d_val;
}

/*-----*\
Look up the coefficient. If its previously been retrieved from the
database, use the cached value. Otherwise, look it up and cache it.
\*-----*/
if (status == PRO_TK_NO_ERROR)
{
    lookup = PTAFExCalcFeedCoeffFind (input);
    if (lookup == NULL)
    {
        coeff = PTAFExCalcFeedGetCoeffFromDB (input);
        PTAFExCalcFeedItemAdd (input, coeff);
    }
    else
    {
        coeff = lookup->output_param;
    }
}

/*-----*\
Set the result of the calculation, to be passed to the left hand side of
the relation.
\*-----*/
output->type = PRO_PARAM_DOUBLE;
output->value.d_val = coeff * cut_feed/cut_dia;
}
else
{
/*-----*\
Unable to calculate the relation, probably due to the inavailability of
the required parameters. Return an error so that the relations interface
flags a problem.
\*-----*/
    status = PRO_TK_GENERAL_ERROR;
}
return (status);
}

```

Creating Manufacturing Features

The creation of manufacturing features mirrors the creation of solid features in Pro/TOOLKIT. All features at the very least must define a feature type. Certain manufacturing features also have a requirement that some “non-redefinable” elements must be defined.

With the exception of fixtures, all features are created in the manufacturing solid. However, fixtures are owned by the manufacturing assembly.

Like solid feature creation, manufacturing feature creation consists of several distinct steps:

1. Create the feature element tree.
2. Add nodes or subtrees to the feature tree.
3. Create a selection that represents the model in which to construct the feature.
4. Create the feature.

The following sections document only the first two steps for manufacturing features, because the actual process of feature creation is common to all.

Creating Fixtures

A fixture setup feature is one of the simplest manufacturing features and contains a maximum of four elements.

You should name a fixture setup feature. Optionally, you can define the following:

- The time required to perform the setup
- The identifiers of the fixturing components
- Associated comments

To Create an Element Tree for a Fixture Setup Feature

1. Allocate space for the tree using the following call:

```
ProElementAlloc (PRO_E_FEATURE_TREE);
```
2. Define the feature type element (PRO_E_FEATURE_TYPE) to be an integer of value PRO_FEAT_FIXSETUP.
3. Define the name (PRO_E_FEAT_NAME) to be a wide string.
4. Optionally, add the setup time (PRO_E_SETUP_TIME) as a double.
5. Optionally, add the component identifiers of the fixturing models (PRO_E_FIXT_COMPONENTS).

Note: Because this is a multivalue element, you can add multiple (integer) values to the PRO_E_FIXT_COMPONENT element.

When the tree is complete, you can pass it (and a selection handle that represents the manufacturing assembly) to the function **ProFeatureCreate()**.

Creating Workcells

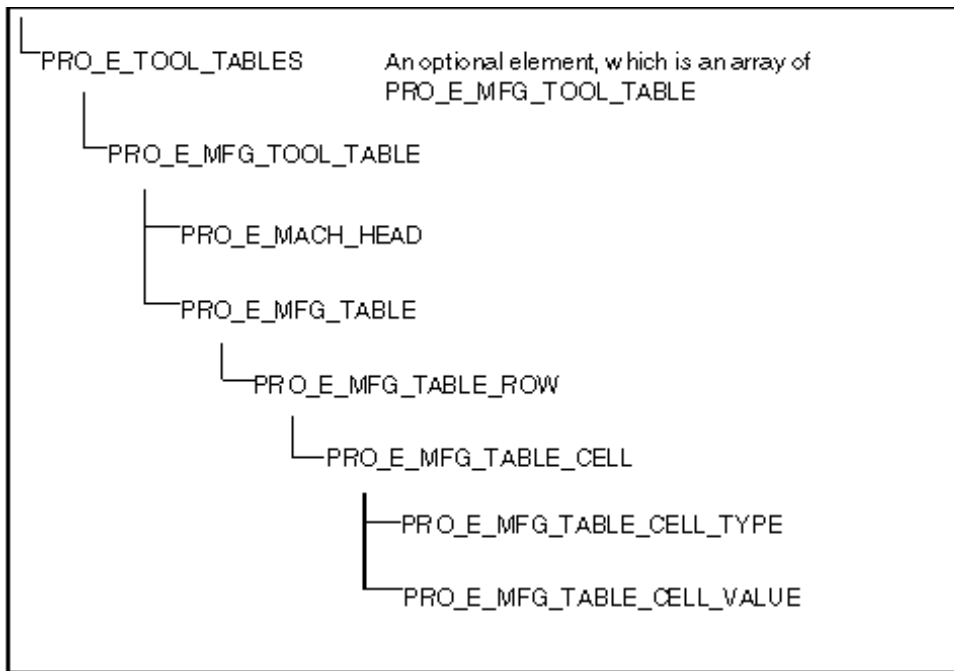
The element tree for workcells is described in the include file *ProWcell.h*. For this feature, the feature type element is **PRO_FEAT_WORKCELL**.

The following table shows the required and optional elements for workcell features. In this table, the “Value” column specifies whether the element is required (R) or optional (O).

Element	Description	Value
Cell type	Mill, mill/turn, and so on.	R
Number of axes	The number of axes.	R
Table direction	Horizontal or vertical (for turn or mill/turn)	R
Machine number heads	1 or 2 (for turn or mill/turn)	R
Name	The workcell name.	O
Tooling	Add tools to the workcell.	O
Tool table	Manipulate the tools in a tool table.	O
Parameters	The workcell parameters.	O

The feature element has two complex elements—**PRO_E_MFG_PARAMS**, described in the section ‘Manufacturing Parameters’, and **PRO_E_TOOL_TABLE**. The following figure shows how the tool table element is constructed.

Figure 53-3: Tool Table Element



The first thing to note is that a workcell can have multiple tool tables, if it has more than one machine head. In this case, you can create a tool table for each head.

A manufacturing table is made up of an array of `PRO_E_MFG_TABLE_ROW` elements, which is itself an array of `PRO_E_MFG_TABLE_CELL` elements. Each `PRO_E_MFG_TABLE_CELL` is a compound element that contains two more elements—the cell type and its value.

The following examples show how to create a tool table with five drill bits, ranging in size from M8 to M16. As described in the *Pro/NC User's Guide*, the tool table for most workcells (excluding CMM) is of the following format:

POSITION	TOOL_ID	REGISTER	COMMENTS
1	drill_M8		8MM Drill
2	drill_M10		10MM Drill

For example, to create the first row of the table, you would define the following `PRO_E_MFG_TABLE_CELLS`:

- To define the first cell so the position of the tool is index 1, set the value of the PRO_E_MFG_TABLE_CELL_TYPE to PRO_TOOL_TABLE_INDEX, and set the value of PRO_E_MFG_TABLE_CELL_VALUE to the integer value 1.
- Similarly, define the tool identifier to be a *drill_M8*. Set the cell type element to PRO_TOOL_TABLE_TOOL_ID, and set the value to a wide string of value *drill_M8*. Because the register column is empty, it can be ignored. To set the comments element, set the type to PRO_TOOL_TABLE_COMMENTS, and set the value to a wide string of value “8MM Drill.”

Example 6: Creating a 2-Axis Lathe Workcell

This example shows how to set up a simple 2-axis, horizontal lathe, using the previously created tool table. Note that both the PRO_E_LATHE_DIR and PRO_E_MACH_NUM_HEADS are required for this workcell type. All the other elements are optional.

```
#include <ProToolkit.h>
#include <ProMfg.h>
#include <ProSolid.h>
#include <ProSelection.h>
#include <ProAsmcomppath.h>
#include <ProElement.h>
#include <ProElemId.h>
#include <ProValue.h>
#include <ProFeature.h>
#include <ProUtil.h>
#include <TestError.h>
#include <TestMfgNew.h>
#include <UtilString.h>
#include <UtilTree.h>
#include <TestConsts.h>
#include <PTApplsUnicodeUtils.h>

/*=====*\
FUNCTION : UserWorkcellCreate
PURPOSE  : Create a workcell
\*=====*/
int UserWorkcellCreate ()
{
    ProMdl    p_mfg_model;
    ProElement wc_elem      = (ProElement)NULL;
                                /* Individual element */
    ProElement wc_elem_tree = (ProElement)NULL;
                                /* Entire tree          */
    ProValueData value_data;
    ProValue    value      = (ProValue)NULL;
    ProErrorlist errors;
```

```

ProSelection    selection;
ProFeature      wc_feature;
ProError        err  = PRO_TK_NO_ERROR;
int             i = 0;

ProError UserWpieceSelCreate ( ProMfg, int, ProType, ProSelection*);
ProError UserToolTableSetup ( ProElement*);

static ElemTable  workcell_elem_table[] = {
    { PRO_E_FEATURE_TYPE,    PRO_VALUE_TYPE_INT    },
    { PRO_E_WCELL_TYPE,     PRO_VALUE_TYPE_INT    },
    { PRO_E_NUM_AXES,       PRO_VALUE_TYPE_INT    },
    { PRO_E_LATHE_DIR,      PRO_VALUE_TYPE_INT    },
    { PRO_E_MACH_NUM_HEADS, PRO_VALUE_TYPE_INT    },
    { PRO_E_TOOL_TABLES,    ARRAY                  },
    { PRO_E_FEAT_NAME,      PRO_VALUE_TYPE_WSTRING } };
int             wc_elem_type_size = sizeof(workcell_elem_table)/
                                sizeof(ElemTable);
err = ProMdlCurrentGet(&p_mfg_model);
ERROR_CHECK("UserWorkcellCreate","ProMdlCurrentGet", err);

err = ProElementAlloc(PRO_E_FEATURE_TREE, &wc_elem_tree);
ERROR_CHECK("UserWorkcellCreate","ProElementAlloc()", err);

for (i=0 ; i < wc_elem_type_size; i++)
{
    err = ProElementAlloc(workcell_elem_table[i].elem_type, &wc_elem);
    ERROR_CHECK("UserWorkcellCreate","ProElementAlloc()", err);
    switch (workcell_elem_table[i].elem_type)
    {
        case PRO_E_FEATURE_TYPE:
            value_data.v.i = PRO_FEAT_WORKCELL;
            break;

        case PRO_E_WCELL_TYPE:
            value_data.v.i = PRO_WCELL_LATHE;
            break;

        case PRO_E_NUM_AXES:
            value_data.v.i = 2;
            break;

        case PRO_E_LATHE_DIR:
            value_data.v.i = PRO_WCELL_LATHE_HORIZONTAL;
            break;

        case PRO_E_MACH_NUM_HEADS:
            value_data.v.i = 1;
            break;

        case PRO_E_FEAT_NAME:

```



```

value_data.v.w = (wchar_t*) malloc (sizeof(ProName));
if (value_data.v.w)
{
    ProStringToWstring(value_data.v.w, "Lathe_2_Axis");
    ERROR_CHECK("UserWorkcellCreate", "ProStringToWstring()", err);
}
break;
case PRO_E_TOOL_TABLES:
err = UserToolTableSetup( &wc_elem );
break;
default:
ProTKFprintf(stderr, "Error setting element type\n");
return PRO_TK_GENERAL_ERROR;
break;
}

if ( workcell_elem_table[i].val_type != ARRAY)
{
    value_data.type = workcell_elem_table[i].val_type;

    err = ProValueAlloc ( &value );

    if (err == PRO_TK_NO_ERROR)
    {
        err = ProValueDataSet (value, &value_data);
        ERROR_CHECK("UserWorkcellCreate", "ProValueDataSet()", err);
    }

    if ( err == PRO_TK_NO_ERROR )
    {
        err = ProElementValueSet ( wc_elem, value );
        ERROR_CHECK("UserWorkcellCreate", "ProElementValueSet()",
                    err);
    }
}
if ( err == PRO_TK_NO_ERROR )
{
    err = ProElementtreeElementAdd (wc_elem_tree, NULL, wc_elem );
    ERROR_CHECK("UserWorkcellCreate", "ProElementtreeElementAdd()",
                err);
}

}
ProUtilElementtreePrint(wc_elem_tree, PRO_TEST_INFO_WINDOW, NULL );

if (err == PRO_TK_NO_ERROR )
err = UserWpieceSelCreate ((ProMfg)p_mfg_model, PRO_TK_NOT_USED,
                           PRO_TK_NOT_USED, &selection);

if (err == PRO_TK_NO_ERROR)
{

```

```

err = ProFeatureCreate(selection, wc_elem_tree, NULL, 0,
                        &wc_feature, &errors);
ERROR_CHECK("UserWorkcellCreate", "ProFeatureCreate()", err);
}

err = ProElementFree(&wc_elem_tree);
ERROR_CHECK("UserWorkcellCreate", "ProElementFree()", err);

return ((int)err);
}

/*=====*\
Function : UserWpieceSelCreate()
Purpose  : Create a selection handle for the mfg workpiece or items in it
\*=====*/
ProError UserWpieceSelCreate (
    ProMfg mfg_model,
    int item_id,
    ProType item_type,
    ProSelection *selection)
{
    ProSolid      mfg_solid;
    ProAsmcomppath comp_path;
    ProModelitem  wkpiece_item;
    ProError err;

    err = ProMfgSolidGet(mfg_model, &mfg_solid);
    ERROR_CHECK("UserWorkcellCreate", "ProMfgSolidGet()", err);

    if (err == PRO_TK_NO_ERROR)
    {
        err= ProMfgFeatureOwnerGet(mfg_model, &comp_path);
        ERROR_CHECK("UserWorkcellCreate", "ProMfgSolidGet()", err);
    }

    if (err == PRO_TK_NO_ERROR)
    {
        err = ProModelitemInit( mfg_solid, item_id, item_type,
                                &wkpiece_item );
        ERROR_CHECK("UserWorkcellCreate", "ProModelitemInit()", err);
    }

    if (err == PRO_TK_NO_ERROR)
    {
        err = ProSelectionAlloc(&comp_path, &wkpiece_item, selection);
        ERROR_CHECK("UserWorkcellCreate", "ProSelectionAlloc()", err);
    }

    return err;
}

```

Creating Operations

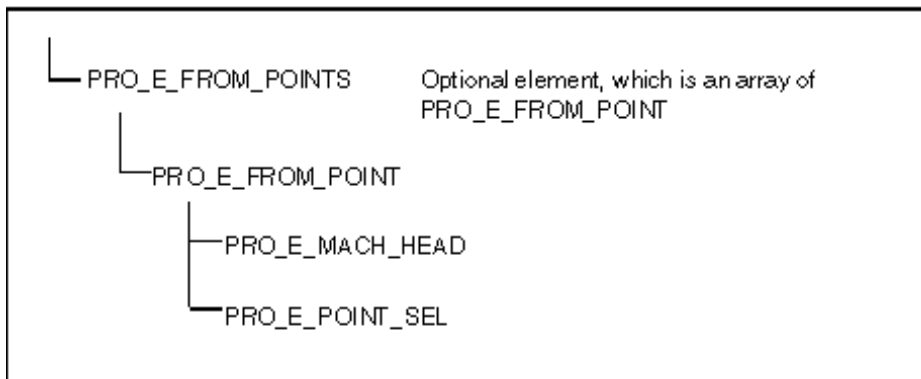
The element tree for manufacturing operations is described in the header file *ProMfgoper.h*. For this feature, the feature type element is PRO_FEAT_WORKCELL.

The following table shows the elements of a manufacturing operation. In this table, the “Value” column specifies whether the element is required (R) or optional (O).

Element	Description	Value
Workcell	The identifier of the workcell feature in which to perform the operation	R
Machine coordinate system	The identifier of the machining coordinate system	R
Name	The operation name	O
Comments	The operation comments	O
From point	The datum point from which to start the operation	O
Home point	The datum point on which to end the operation	O
Parameters	The operation parameters	O

The creation of the feature tree is simple, apart from the home and point elements. The following figure shows the element tree for the home points.

Figure 53-4: Home Point Element Tree



In this example, there is a from and home point defined for each machining head. If there is only one head, the value of the PRO_E_MACH_HEAD element should be 1. Note that the PRO_E_POINT_SEL selection should be initialized *not* to the datum point feature, but to the datum point geometry. To find this geometry, call the function **ProFeatureGeomitemVisit()**.

Example 7: Creating an Operation

This example shows how to create an operation.

```
#include <ProToolkit.h>
#include <ProMfg.h>
#include <ProSolid.h>
#include <ProSelection.h>
#include <ProAsmcomppath.h>
#include <ProElement.h>
#include <ProElemId.h>
#include <ProValue.h>
#include <ProFeature.h>
#include <ProUtil.h>

#include <TestError.h>
#include <TestMfgNew.h>
#include <UtilString.h>
#include <UtilTree.h>
#include <UtilVisit.h>
#include <TestConsts.h>
/*=====*\
Function : UserOperationCreate
Purpose  : Create operation using element tree
\*=====*/
ProError UserOperationCreate (
    ProMfg      mfg_model,
    ProName     feat_name,
    int         wcell_id,
    ProSelection csys_sel )
{
    ProError      err = PRO_TK_NO_ERROR;
    ProElement    oper_elem = (ProElement)NULL;
                                /* Individual element */
    ProElement    oper_elem_tree = (ProElement)NULL; /* Entire tree */
    ProValueData  value_data;
    ProValue      value = (ProValue)NULL;
    ProErrorlist  errors;
    ProSelection  selection;
    ProFeature    oper_feature;
    int           i = 0;

    ProError UserWpieceSelCreate ( ProMfg, int, ProType, ProSelection*);
```

```

static ElemTable oper_elem_table[] = {
    { PRO_E_FEATURE_TYPE, PRO_VALUE_TYPE_INT },
    { PRO_E_WCELL, PRO_VALUE_TYPE_INT },
    { PRO_E_MACH_CSYS, PRO_VALUE_TYPE_SELECTION },
    { PRO_E_FEAT_NAME, PRO_VALUE_TYPE_WSTRING } };

int oper_elem_type_size = sizeof(oper_elem_table)/
    sizeof(ElemTable);

err = ProElementAlloc(PRO_E_FEATURE_TREE, &oper_elem_tree);
ERROR_CHECK("UserOperationCreate", "ProElementAlloc()", err);

for (i=0 ; i < oper_elem_type_size; i++)
{
    err = ProElementAlloc(oper_elem_table[i].elem_type, &oper_elem);
    ERROR_CHECK("UserOperationCreate", "ProElementAlloc()", err);

    switch (oper_elem_table[i].elem_type)
    {
        case PRO_E_FEATURE_TYPE:
            value_data.v.i = PRO_FEAT_OPERATION;
            break;

        case PRO_E_WCELL:
            value_data.v.i = wcell_id;
            break;

        case PRO_E_MACH_CSYS:
            err = ProSelectionCopy(csys_sel, &(value_data.v.r));
            ERROR_CHECK("UserOperationCreate", "ProSelectionCopy()", err);
            break;

        case PRO_E_FEAT_NAME:
            value_data.v.w = (wchar_t*) malloc (sizeof(ProName));
            if (value_data.v.w)
                ProUtilWstrcpy(value_data.v.w, feat_name);
            else
                err = PRO_TK_BAD_INPUTS;
            break;

        default:
            ProTKFprintf(stderr, "Error setting element type\n");
            return PRO_TK_GENERAL_ERROR;
            break;
    }

    if ( oper_elem_table[i].val_type != ARRAY)
    {
        value_data.type = oper_elem_table[i].val_type;
        err = ProValueAlloc ( &value );
    }
}

```

```

    if (err == PRO_TK_NO_ERROR)
    {
        err = ProValueDataSet (value, &value_data);
        ERROR_CHECK("UserOperationCreate", "ProValueDataSet()", err);
    }

    if ( err == PRO_TK_NO_ERROR )
    {
        err = ProElementValueSet ( oper_elem, value );
        ERROR_CHECK("UserOperationCreate", "ProElementValueSet()",
                    err);
    }
}

if ( err == PRO_TK_NO_ERROR )
{
    err = ProElemtreeElementAdd (oper_elem_tree, NULL, oper_elem );
    ERROR_CHECK("UserOperationCreate", "ProElemtreeElementAdd()",
                err);
}

}

ProUtilElementtreePrint(oper_elem_tree, PRO_TEST_INFO_WINDOW, NULL );

if (err == PRO_TK_NO_ERROR )
err = UserWpieceSelCreate (mfg_model, PRO_TK_NOT_USED,
                          PRO_TK_NOT_USED, &selection);

if (err == PRO_TK_NO_ERROR)
{
    err = ProFeatureCreate(selection, oper_elem_tree, NULL, 0,
                          &oper_feature, &errors);
    ERROR_CHECK("UserOperationCreate", "ProFeatureCreate()", err);
}

err = ProElementFree(&oper_elem_tree);
ERROR_CHECK("UserOperationCreate", "ProElementFree()", err);

return ((int)err);
}

```

Creating NC Sequences

Header file *ProNcseq.h* describes the element tree for manufacturing NC sequences.

There are six supported NC sequence types:

- PRO_NCSEQ_PROF_SUR_MILL—Profile milling, feature type PRO_FEAT_MILL

- PRO_NCSEQ_VOL_MILL—Volume milling, feature type PRO_FEAT_MILL
- PRO_NCSEQ_CONV_SURF_MILL—Conventional surface, feature type PRO_FEAT_MILL
- PRO_NCSEQ_FACE_MILL—Face milling, feature type PRO_FEAT_MILL
- PRO_NCSEQ_PREV_TOOL_MILL—Local milling using previous tool, feature type PRO_FEAT_MILL
- PRO_NCSEQ_HOLEMAKING—Holemaking, type PRO_FEAT_DRILL

Like workcells, an NC sequence feature has a number of non-redefinable elements. For all NC sequences, the PRO_E_NCSEQ_TYPE and PRO_E_NUM_AXES elements are non-redefinable. For holemaking sequences, the PRO_E_HOLEMAKING_TYPE, PRO_E_PECK_TYPE, and PRO_E_HOLE_CYCLE_TYPE elements are non-redefinable. These elements are specific to holemaking sequences and need not be defined for milling sequences. The following table shows the elements of an NC sequence. In this table, the Value column specifies whether the element is required (R) or optional (O).

Element	Description	Value
Feature type	The feature type	R
Type	The sequence type	R
Operation	Operation to which to add the NC sequence	R
Retraction plane	The retraction plane	R
Tool	The tool	R
Csys	The manufacturing coordinate system	R
Parameters	The manufacturing parameters	R
Name	The name of the NC sequence	O
Number of axes	The number of axes	O
Machine head	The machine head	O
Fixture	The fixture	O
<i>Entities to be machined</i>		
Surface	The surface	R
Holes or volume	The holes or volume	R
Start path	The start path	R

Element	Description	Value
End path	The end path	R

The retraction plane that must be defined as part of the NC sequence requires the identifier of the underlying geometry of the datum plane. To obtain this identifier, visit the datum plane geometry items using **ProFeatureGeomitemVisit()**.

There are also a number of required parameters for each NC sequence that must be defined. For conventional milling, the following parameters are required:

- CUT_FEED
- TOLERANCE
- STEP_OVER
- SPINDLE_SPEED
- CLEAR_DIST

For face milling, the following parameters are required:

- CUT_FEED
- STEP_DEPTH
- TOLERANCE
- STEP_OVER
- SPINDLE_SPEED
- CLEAR_DIST

For holmaking, the following parameters are required:

- CUT_FEED
- TOLERANCE
- SPINDLE_SPEED
- CLEAR_DIST

Both milling and holmaking features elements require that the entities (and some associated properties) to be machined are set by API functions, rather than by element tree. Like the standard elements, these functions require a call to **ProElementAlloc()** to reserve space for the elements. Once the elements are complete, you can add them to the tree like the other standard elements.

Note: Currently, using **ProFeatureElemtreeExtract()** with NC sequences yields an element tree without holes or surface elements. In other words, there is no way to retrieve hole set or surface information.

Milling-Specific Functions

Functions Introduced:

- **ProNcseqElemSurfaceAdd()**
- **ProNcseqElemMillsurfSet()**
- **ProNcseqElemSurfaceflipSet()**

After you allocate the surface element with **ProElementAlloc()**, you can add the surface to be milled to the element using the function **ProNcseqElemSurfaceAdd()**.

If the model contains a milling surface, you can set the whole surface in the NC sequence using **ProNcseqElemMillsurfSet()**. To control its orientation, call **ProNcseqElemSurfaceflipSet()**.

Example 8: Adding Surfaces

This example shows how to use the function **ProNcseqElemSurfaceAdd()**.

```

/*=====*\
FUNCTION: UserMfgSurfacesAdd
PURPOSE: Add surfaces to element using ProNcseqElemSurfaceAdd().
\*=====*/
ProError UserMfgSurfacesAdd (
    ProElement    *surfaces)
{
    ProError      err;
    ProSelection  *selection;
    int           num, i;

/*-----*\
    Select some surfaces.
\*-----*/
    err = ProSelect ("surface", -1, NULL, NULL, NULL, NULL,
        &selection, &num);

/*-----*\
    Add the surfaces to the sequence element.
\*-----*/
    for (i = 0; i < num && err == PRO_TK_NO_ERROR; i++)
    {
        err = ProNcseqElemSurfaceAdd (*surfaces, selection[i]);
    }

    return err;

```

}

Holemaking-Specific Functions

To add hole sets to an element tree, first, obtain the hole set number by calling the function **ProNcseqElemHolesetAdd()**. This hole set is used to reference a set of holes with the same properties, including depth, direction, countersink diameter, and so on. Note that feature element PRO_E_HOLESETS has limited support for drill point sets, but allows the user to identify and to delete existing drill point sets in a feature, or to overwrite drill point sets with drill axes sets.

For example, you might want one set of holes to be countersunk to a diameter of 10mm, and another set to 14mm. Because these countersinking operations are done with the same tool, they should be in the same NC sequence. Because the countersink diameter is different for each, you should create two hole sets.

Note: The following functions (and the element tree PRO_E_HOLES) exist in Pro/TOOLKIT Revisions 20 and later only to provide backwards compatibility. Use the more complete and powerful element tree PRO_E_HOLESETS for holemaking functions.

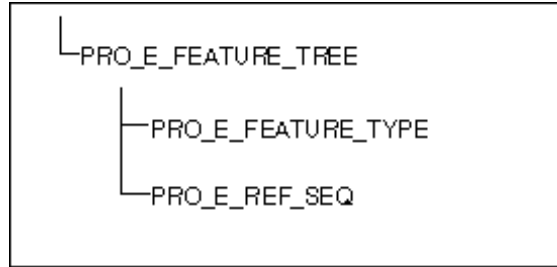
- ProNcseqElemHolesetAdd()
- ProNcseqElemHolesetDepthTypeSet()
- ProNcseqElemHolesetDepthBySet()
- ProNcseqElemHolesetDepthSet()
- ProNcseqElemHolesetStartSet()
- ProNcseqElemHolesetEndSet()
- ProNcseqElemHolesetDirectionSet()
- ProNcseqElemHolesetAxisAdd()
- ProNcseqElemHolesetDrillpartAdd()
- ProNcseqElemHolesetCsinkdiamSet()

Creating Material Removal Volumes

In Pro/ENGINEER, material removal features can be created by defining geometry to represent the volume removed, or they can be calculated automatically from the NC sequence. The current release of Pro/TOOLKIT supports automatic material removal feature creation only.

The feature tree is very simple, as shown in the following figure.

Figure 53-5: Feature Tree



Set the PRO_E_FEATURE_TYPE value to PRO_FEAT_MAT_REMOVAL, and set the PRO_E_REF_SEQ to the identifier of the NC sequence feature from which to create the material removal volume.

Example 9: Creating a Conventional Milling Sequence

This example shows how to create a milling feature using an element tree.

```
#include <ProToolkit.h>
#include <ProMfg.h>
#include <ProSolid.h>
#include <ProSelection.h>
#include <ProAsmcomppath.h>
#include <ProElement.h>
#include <ProElemId.h>
#include <ProValue.h>
#include <ProFeature.h>
#include <ProUtil.h>
#include <ProTool.h>

#include <TestError.h>
#include <TestMfgNew.h>
#include <UtilString.h>
#include <UtilTree.h>
#include <UtilVisit.h>
#include <TestConsts.h>
#include <PTApplsUnicodeUtils.h>
#include <ProMessage.h>
#include <ProModelitem.h>
#include <UtilMessage.h>

/*=====*\
Function : UserMillingCreate
Purpose  : Create a milling feature using element tree
```

```

\*=====*/
ProError UserMillingCreate (
    ProMfg          mfg_model,
    ProNcseqType    ncseq_type,
    ProName         feat_name,
    int             oper_id,
    int             retract_id,
    ProSelection    *mach_csys,
    wchar_t         *tool_name )
{
    ProError        err = PRO_TK_NO_ERROR;
    ProElement      ncseq_elem = (ProElement)NULL;
                                /* Individual element */
    ProElement      ncseq_elem_tree = (ProElement)NULL; /* Entire tree */
    ProValueData    value_data;
    ProValue        value = (ProValue)NULL;
    ProErrorlist    errors;
    ProSelection    selection;
    ProFeature      ncseq_feature;
    ProFeatureCreateOptions fd_opts[] = {PRO_FEAT_CR_DEFINE_MISS_ELEMS };
    int             i = 0;

    ProError UserWpieceSelCreate ( ProMfg, int, ProType, ProSelection*);
    ProError UserMfgParamsSet (ProElement*);
    ProError UserMfgSurfacesAdd (ProElement*);

    static ElemTable ncseq_elem_table[] = {
        { PRO_E_FEATURE_TYPE,    PRO_VALUE_TYPE_INT      },
        { PRO_E_NCSEQ_TYPE,      PRO_VALUE_TYPE_INT      },
        { PRO_E_OPERATION,       PRO_VALUE_TYPE_INT      },
        { PRO_E_RETRACT,         PRO_VALUE_TYPE_INT      },
        { PRO_E_CSYS,            PRO_VALUE_TYPE_SELECTION },
        { PRO_E_MFG_PARAMS,      ARRAY                    },
        { PRO_E_SURFACES,        COMPLEX                  },
        { PRO_E_TOOL,            PRO_VALUE_TYPE_WSTRING   },
        { PRO_E_FEAT_NAME,       PRO_VALUE_TYPE_WSTRING   } };

    int             ncseq_elem_type_size = sizeof(ncseq_elem_table)/
                                sizeof(ElemTable);

    /*-----*\
        Allocate feature tree element
    /*-----*/
    err = ProElementAlloc(PRO_E_FEATURE_TREE, &ncseq_elem_tree);
    ERROR_CHECK("UserNcsequenceCreate", "ProElementAlloc()", err);

    for (i=0 ; i < ncseq_elem_type_size; i++)
    {
        /*-----*\
            Allocate sequence element
        /*-----*/

```

```

err = ProElementAlloc(ncseq_elem_table[i].elem_type, &ncseq_elem);
ERROR_CHECK("UserNcsequenceCreate", "ProElementAlloc()", err);
switch (ncseq_elem_table[i].elem_type)
{
    case PRO_E_FEATURE_TYPE :
        value_data.v.i = PRO_FEAT_MILL;
        break;

    case PRO_E_NCSEQ_TYPE :
        value_data.v.i = ncseq_type ;
        break;

    case PRO_E_OPERATION :
        value_data.v.i = oper_id ;
        break;

    case PRO_E_RETRACT :
        value_data.v.i = retract_id ;
        break;

    case PRO_E_MFG_PARAMS:
        err = UserMfgParamsSet(&ncseq_elem);
        break;

    case PRO_E_CSYS:
        err = ProSelectionCopy(*mach_csys, &(value_data.v.r));
        break;

    case PRO_E_SURFACES:
        err = UserMfgSurfacesAdd(&ncseq_elem);
        break;

    case PRO_E_FEAT_NAME:
        value_data.v.w = (wchar_t*) malloc (sizeof(ProName));

        if (value_data.v.w)
            ProUtilWstrcpy(value_data.v.w, feat_name);
        else
            err = PRO_TK_BAD_INPUTS;
        break;

    case PRO_E_TOOL:
        value_data.v.w = (wchar_t*) malloc (sizeof(ProName));
        if (value_data.v.w)
            ProUtilWstrcpy(value_data.v.w, (wchar_t*)tool_name);
        else
            err = PRO_TK_BAD_INPUTS;
        break;

    default:

```

```

        ProTKFprintf(stderr, "Error setting element type\n");
        return PRO_TK_GENERAL_ERROR;
        break;
    }
/*-----*\
    If element is simple add it its value to the element
\*-----*/
    if ( ncseq_elem_table[i].val_type != ARRAY &&
        ncseq_elem_table[i].val_type != COMPLEX )
    {
        value_data.type = ncseq_elem_table[i].val_type;
        err = ProValueAlloc ( &value );

        if (err == PRO_TK_NO_ERROR)
        {
            err = ProValueDataSet (value, &value_data);
            ERROR_CHECK("UserNcsequenceCreate", "ProValueDataSet()", err);
        }

        if ( err == PRO_TK_NO_ERROR )
        {
            err = ProElementValueSet ( ncseq_elem, value );
            ERROR_CHECK("UserNcsequenceCreate", "ProElementValueSet()",
                        err);
        }
    }
/*-----*\
    Add the element to the feature tree
\*-----*/
    if ( err == PRO_TK_NO_ERROR )
    {
        err = ProElemtreeElementAdd (ncseq_elem_tree, NULL, ncseq_elem );
        ERROR_CHECK("UserNcsequenceCreate", "ProElemtreeElementAdd()",
                    err);
    }

    }

    ProUtilElementtreePrint(ncseq_elem_tree, PRO_TEST_INFO_WINDOW, NULL );

/*-----*\
    Create the nc-sequence feature in the workpiece
\*-----*/
    if (err == PRO_TK_NO_ERROR )
        err = UserWpierceSelCreate (mfg_model, PRO_TK_NOT_USED,
                                    PRO_TK_NOT_USED, &selection);

    if (err == PRO_TK_NO_ERROR)
    {
        err = ProFeatureCreate(selection, ncseq_elem_tree, fd_opts, 1,
                                &ncseq_feature, &errors);
    }

```

```

        ERROR_CHECK("UserNcsequenceCreate", "ProFeatureCreate()", err);

        if (err != PRO_TK_NO_ERROR)
        {
            ProTKFprintf(stderr, "Error in element %d\n",
                errors.error_list[0].err_item_id);
        }
    }

/*-----*\
    Free the feature tree
\*-----*/
    err = ProElementFree(&ncseq_elem_tree);
    ERROR_CHECK("UserNcsequenceCreate", "ProElementFree()", err);

    return ((int)err);
}

```

Analyzing Manufacturing Features

Functions Introduced:

- **ProMfgoperToolpathDisplay()**
- **ProNcseqToolpathDisplay()**
- **ProNcseqNumGet()**
- **ProNcseqCutTimeGet()**
- **ProNcseqRemovedVolGet()**

You can use the functions **ProMfgoperToolpathDisplay()** and **ProNcseqToolpathDisplay()** to invoke the corresponding toolpath for the specified object. The function **ProNcseqNumGet()** returns the number of the specified NC sequence.

The final two functions access the machining time and the volume of material removed during the machining sequence.

Production Applications: Customized Tool Database

Pro/TOOLKIT supports integration of the Pro/NC tool search command with external tool databases. These functions and callbacks allow users to create queries for third-party tool manager applications. These applications query external databases (you can specify more than one) and return logical tool data to Pro/ENGINEER.

Topic	Page
Overview	54 - 2
Setting up the Database and Custom Search Parameters	54 - 2
Registering the External Database	54 - 3
Querying the External Database	54 - 4
Returning the Search Results	54 - 8

Overview

Pro/TOOLKIT supplies functions that allow integration of the **Tool Selection** user interface with a third-party tool manager application. This integration consists of includes three areas of functionality:

- The ability to preregister one or more databases with the Pro/ENGINEER user interface, and to assign custom search parameters to each database. Custom search parameters are items which would not be included natively inside of Pro/ENGINEER, but which could be used to initiate or narrow the tool search.
- Callback functions, that are invoked when the user opts to search inside the tool database. Within the callback functions, the application can extract the native and non-native search parameters and use these parameters to execute the search.
- The ability to return a list of detailed results to Pro/ENGINEER to be displayed in the results dialog box.

Setting up the Database and Custom Search Parameters

The functions described in this section enable you to setup the external database and custom search parameters supported by the database.

Functions Introduced:

- **ProMfgdbDataAlloc()**
- **ProMfgdbDataDbnameAdd()**
- **ProMfgdbNameCreate()**
- **ProMfgdbSearchoptCreate()**
- **ProMfgdbDataSearchoptAdd()**
- **ProMfgdbSearchoptApplicDataAdd()**
- **ProMfgdbSearchoptAllowedValueAdd()**

The function **ProMfgdbDataAlloc()** allocates a ProMfgdbData handle containing database and search information.

The function **ProMfgddbDataDbnameAdd()** adds the name of an external tool database to the handle allocated by **ProMfgddbDataAlloc()**. To specify more than one database, make multiple calls to **ProMfgddbDataDbnameAdd()**.

The function **ProMfgddbNameCreate()** allocates memory for a handle containing the name of a group into which search options are organized.

The function **ProMfgddbSearchoptCreate()** allocates and initializes a structure for a custom search option. It requires that the option be assigned to a group, typically this would be allocated from **ProMfgddbNameCreate()**.

The function **ProMfgddbDataSearchoptAdd()** adds a custom search option to the ProMfgddbData handle.

The function **ProMfgddbSearchoptApplicDataAdd()** specifies the category and object type for which a custom search option is valid. Call this function at least once for each option. Assign an option to multiple categories or object types with multiple calls to this function.

The function **ProMfgddbSearchoptAllowedValueAdd()** adds valid values available for a search option. Assign multiple values with multiple calls to this function.

Registering the External Database

The function described in this section allows you to register the Pro/TOOLKIT application to search the external tool database.

Function Introduced:

- **ProMfgddbRegister()**

The function **ProMfgddbRegister()** registers an external database with Pro/ENGINEER. You should supply the ProMfgddbData handle you created for your external database(s).

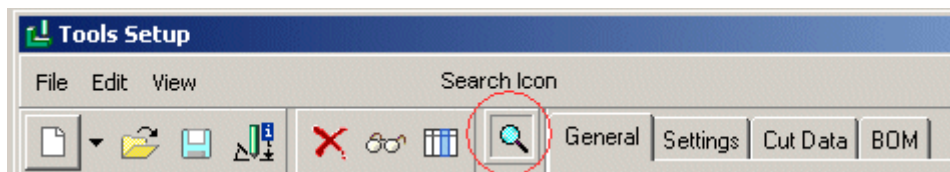
The function also requires that you supply three callbacks:

- A ProMfgddbLoginAction, which is called by Pro/ENGINEER when the user attempts to initiate access to the external database.
- A ProMfgddbLogoffAction, which is called when the user is ready to close the connection to the external database.

- A ProMfgdbSearchAction, which provides the custom implementation for the search, and provides the results back to Pro/ENGINEER.

Finally, the function requires that you supply the object category. Although there are several categories listed for this function, currently the function supports only the following category:

- PROMFGDBCAT_CUTTING_TOOL
 - When one or more external databases have been registered with Pro/ENGINEER, a search icon is displayed in the **Tool Setup** dialog box in Pro/ENGINEER as shown in the following figure. Click the search icon to invoke the **Cutting Tool Search** dialog box.



Querying the External Database

When users invoke the Search icon to construct a query, they will be presented with the Cutting Tool Search dialog, as shown in the following figure. Within the dialog, users can specify the query parameters, values and group the query constraints with logical operators. When the user completes the query and executes the Find command, Pro/ENGINEER will call the Pro/TOOLKIT application with the data from this dialog.

Cutting Tool Search

File

Tool Type:

Parameter Owner:

Group of Parameters:

Value:

TOOL_ID
 LENGTH_UNITS
TOOL_MATERIAL
 NUM_OF_TEETH
 CUTTER_DIAM
 LENGTH

=

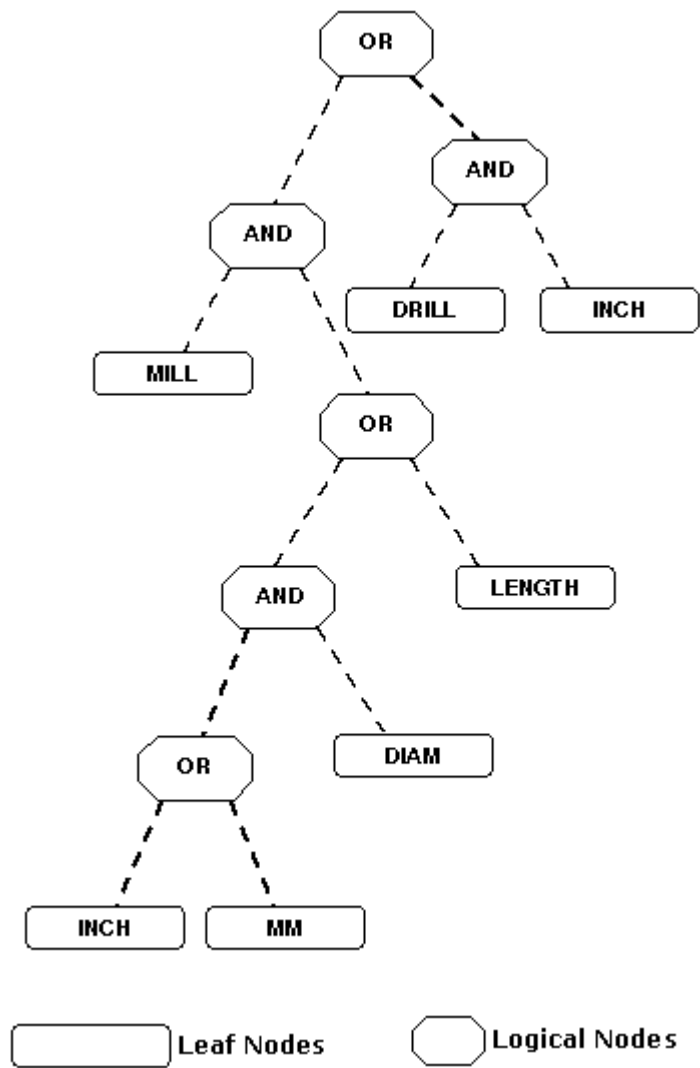
☒ OR ☐ AND

(TOOL_TYPE="MILLING"
 AND
 ((LENGTH_UNITS="INCH"
 AND
 TOOL_MATERIAL="CARBIDE")
 OR
TOOL_MATERIAL="HSS"))

Production Applications:
 Customized Tool
 Database

The functions described in this section enable you to convert the Pro/ENGINEER queries into a format that can be used with the external database.

Pro/ENGINEER tool database queries are arranged in a tree format, as shown in the following figure. Leaf nodes contain expressions, for example, units and type.



Functions Introduced:

- **ProMfgdbQuerynodeIsLeaf()**
- **ProMfgdbQuerynodeLeftChildGet()**
- **ProMfgdbQuerynodeRightChildGet()**
- **ProMfgdbQuerynodeLogicOperGet()**
- **ProMfgdbQuerynodeExprGet()**
- **ProMfgdbQueryTargetGet()**
- **ProMfgdbExprNameGet()**
- **ProMfgdbExprCategoryGet()**
- **ProMfgdbExprValueGet()**
- **ProMfgdbExprValueTypeGet()**
- **ProMfgdbExprCompopGet()**

The function **ProMfgdbQuerynodeIsLeaf()** returns TRUE if a specified node is a leaf node, that is, a node that contains an expression.

The functions **ProMfgdbQuerynodeLeftChildGet()** and **ProMfgdbQuerynodeRightChildGet()** return either the left or right branch of a query node, respectively.

The function **ProMfgdbQuerynodeLogicOperGet()** returns the logical operator (AND or OR) applied by this node to the specified query. Apply this function only to nodes for which **ProMfgdbQuerynodeIsLeaf()** returns FALSE.

The function **ProMfgdbQuerynodeExprGet()** returns the expression for the specified query. The node that is interrogated must be a leaf node.

The function **ProMfgdbQueryTargetGet()** returns the target object that is searched for by the query tree that contains the specified node. All nodes in the same query tree return the same target.

After the query functions return expressions from a leaf node, your Pro/TOOLKIT application can gather information contained in the expressions. The following functions return attributes and operators contained in the returned expressions of a leaf node.

The function **ProMfgdbExprNameGet()** returns the name of the attribute contained in the specified expression. The function **ProMfgdbExprCategoryGet()** returns the category of the attribute contained in the specified expression. The functions **ProMfgdbExprValueGet()** and **ProMfgdbExprValueTypeGet()** return the value and value type contained in the specified expression, respectively. The function **ProMfgdbExprCompopGet()** returns the comparison operator (=, <, >, and so on) contained in the specified expression.

Returning the Search Results

The functions in this section enable you to populate the results of the query to the external database and pass them back to Pro/ENGINEER so that Pro/ENGINEER will display them appropriately in the results window.

Functions Introduced:

- **ProMfgdbMatchAlloc()**
- **ProMfgdbMatchParamAdd()**

Use the function **ProMfgdbMatchAlloc()** to allocate memory for the structure used to store a match to the query.

The function **ProMfgdbMatchParamAdd()** adds a parameter to the match structure. Make multiple calls to this function to add multiple parameters to the match.

An example of the results passed back from a database query is shown in the following figure:

Search result #2 - number of rows: 4

TOOL_TYPE	TOOL_ID	LENGTH_UNITS	TOOL_MATERIAL	NUM_OF_TEETH
MILLING	T0001	INCH	T-15 COBALT	6.000000
MILLING	KMC950	Millimeter	T-18 COBALT	4.000000
DRILLING	DRILL_W_ERROR	INCH	COBALT	
DRILLING	JOBBER_2	INCH	COBALT	

Send Select All Unselect All Close

Production Applications:
Customized Tool
Database

Production Applications: NC Sequences and CL Commands

This chapter describes the Pro/TOOLKIT functions that enable you to create auxiliary and custom NC sequences and CL commands.

Topic	Page
Auxiliary and Custom NC Sequences	55 - 2
CL Commands	55 - 3

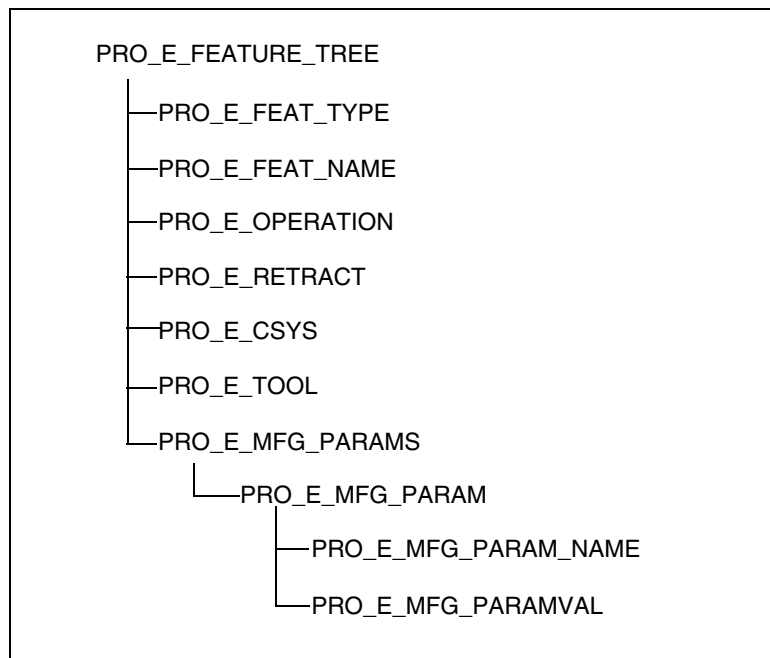
Auxiliary and Custom NC Sequences

In Pro/NC, you can create auxiliary NC sequences to obtain low-level control over cut and tool motion, and CL commands. In the context of a Pro/TOOLKIT application, an auxiliary NC sequence is a manufacturing feature created in the manufacturing solid that has the feature type `PRO_FEAT_AUXILIARY`. You create an auxiliary NC sequence in Pro/TOOLKIT using the feature element tree.

Although toolpath creation is not yet available in Pro/TOOLKIT, you can create a custom NC sequence in which CL commands are defined programmatically. The custom NC sequence is a feature of the manufacturing solid. Its feature type is `PRO_E_FEAT_BLD_PATH` and it must refer to the auxiliary NC sequence.

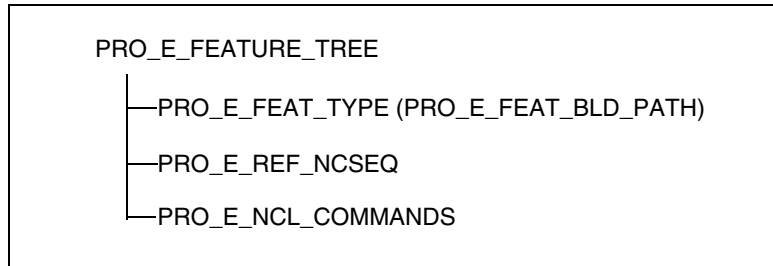
The following figure shows the feature element tree for auxiliary NC sequences.

Figure 55-1 Element Tree for Auxiliary NC Sequence



The following figure shows the feature element tree for custom NC sequences.

Figure 55-2: Element Tree for Custom NC Sequence



CL Commands

Functions Introduced:

- **ProClcmdElemCreate()**
- **ProClcmdElemSet()**
- **ProClcmdElemAdd()**
- **ProClcmdElemNumberGet()**
- **ProClcmdElemGetByIndex()**
- **ProClcmdElemRemoveByIndex()**
- **ProClcmdElemGet()**
- **ProClcmdElemRemove()**
- **ProNotificationSet()**
- **ProClCommandExpandAction()**
- **ProClCommandGetLocAction()**

You can create a CL command programmatically in Pro/TOOLKIT as an element of a custom NC sequence feature. You should provide the CL command string, location, and axis, as well as necessary command interpretation and location retrieval functions.

The following functions are used in CL command creation process:

- **ProClcmdElemCreate()**—Allocate and initialize a CL command element.
- **ProClcmdElemSet()**—Use on an existing element to fill the CL command with user data.
- **ProClcmdElemAdd()**—Add the CL command element to the feature element tree when you have allocated it and assigned a CL command.
- **ProClcmdElemNumberGet()**—Determine the number of CL commands in a custom NC sequence feature.
- **ProClcmdElemGetByIndex()**—Retrieve the handle to a CL command element, given its position (index) in a custom NC sequence.
- **ProClcmdElemRemoveByIndex()**—Remove a CL command element from the specified NC sequence.
- **ProClcmdElemGet()**—Retrieve a CL command element. This function requires as input the exact CL command string, cutter location, and axis.
- **ProClcmdElemRemove()**—Delete a CL command, given the CL command string, cutter location, and axis.

When user CL commands are created, there are two types of notifications that can be trapped. With PRO_NCL_COMMAND_EXPAND, you must provide a callback function to interpret (if necessary) the CL commands. PRO_NCL_COMMAND_GET_LOC invokes the callback function that retrieves the CL command location. You must set up any notifications that are necessary using the function **ProNotificationSet()**.

Customizing CL Commands

To Customize CL Commands

1. Create an auxiliary NC sequence:
 - Build a complete element tree that includes all the necessary feature elements (such as the feature type, operation, retraction plane, machine coordinate system, and tools).
 - Call **ProFeatureCreate()** to create the auxiliary NC feature.

2. Create a custom NC sequence that contains the customized CL commands:
 - Add all the feature elements to the element tree. Note that for PRO_E_NCL_COMMANDS, you must set up CL command data.
 - Set CL notification for interpretation and location retrieval.
 - Create the custom NC sequence by calling **ProFeatureCreate()**.

Production Applications: Process Planning

This chapter describes how to use the Pro/TOOLKIT functions for assembly process operations. It assumes that you are familiar with the functionality of Pro/PROCESS for ASSEMBLIES.

Topic	Page
Process Step Objects	56 - 2
Visiting Process Steps	56 - 2
Process Step Access	56 - 2
Creating Process Steps	56 - 3

Process Step Objects

Function Introduced:

- **ProProcstepInit()**

Process steps are represented by the object *ProProcstep*, which is an instance of *ProModelitem*. The object *ProProcstep* describes the contents and ownership of a assembly process step.

The declaration is as follows:

```
typedef struct pro_model_item
{
    ProType    type;
    int        id;
    ProMdl     owner;
} ProProcstep;
```

To create a new process step handle, use the function **ProProcstepInit()**.

Visiting Process Steps

Function Introduced:

- **ProProcstepVisit()**

The function **ProProcstepVisit()** enables you to visit all the process steps in the specified solid. For a detailed explanation of visiting functions, see the section 'Visit Functions' in the 'Fundamentals' chapter.

Process Step Access

Functions Introduced:

- **ProProcstepActiveGet()**
- **ProProcstepActiveSet()**
- **ProProcstepNumberGet()**

These functions access the process step objects.

The functions **ProProcstepActiveGet()** and **ProProcstepActiveSet()** enable you to get and set the current active process step.

The function **ProProcstepNumberGet()** retrieves the process step number for the specified solid and process step.

Creating Process Steps

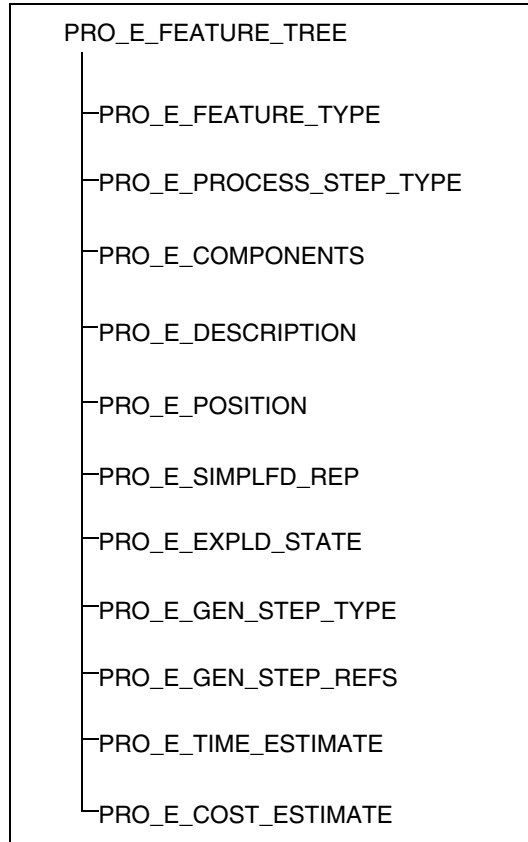
To create an assembly process step, use the function **ProFeatureCreate()**. As with any other type of feature, you use the element tree to create a process step feature.

This chapter describes the basic principles of programmatic process step creation. The chapter 'Element Trees: Principles of Feature Creation' is a necessary background for this topic; therefore, you should read that chapter first.

The element tree for a process step feature is documented in the header file *ProProcstep.h* and has a fairly simple structure, as shown in the following figure. You can also create a copy of the feature element tree of an existing process step feature by calling the function **ProFeatureElemtreeExtract()** with an input feature of type *ProProcstep*.

The following figure shows the element tree of a process step feature.

Figure 56-1: Element Tree of Process Step Feature



Feature Elements

The following table describes the tree elements in more detail.

Element ID	Data Type	Description
PRO_E_FEATURE_TYPE	PRO_VALUE_TYPE_INT	PRO_FEAT_ PROCESS_STEP
PRO_E_PROCESS_STEP_TYPE	PRO_VALUE_TYPE_INT	See 'Types of Process Step'
PRO_E_COMPONENTS	PRO_VALUE_TYPE_ SELECTION	Step components

Element ID	Data Type	Description
PRO_E_DESCRIPTION	PRO_VALUE_TYPE_WSTRING	Step description
PRO_E_POSITION	PRO_VALUE_TYPE_TRANSFORM	Position transformation
PRO_E_GEN_STEP_TYPE	PRO_VALUE_TYPE_WSTRING	General step type
PRO_E_GEN_STEP_REFS	PRO_VALUE_TYPE_SELECTION	General step references
PRO_E_SIMPLFD_REP	PRO_VALUE_TYPE_INT	Simplified representation identifier
PRO_E_EXPLODE_STATE	PRO_VALUE_TYPE_INT	Explode state identifier
PRO_E_TIME_ESTIMATE	PRO_VALUE_TYPE_DOUBLE	Time estimate (greater than 0.0)
PRO_E_COST_ESTIMATE	PRO_VALUE_TYPE_DOUBLE	Cost estimate (any value)

Types of Process Step

The types of process step are as follows:

- PRO_PROCSTEP_ASSEMBLE—Step to assemble components
- PRO_PROCSTEP_DISASSEMBLE—Step to disassemble components
- PRO_PROCSTEP_REASSEMBLE—Step to reassemble components
- PRO_PROCSTEP_REPOSITION—Step to reposition components
- PRO_PROCSTEP_GENERAL—General step (default or user-defined types)

Optional Elements

The following elements in the tree are optional for all types of process step:

- PRO_E_DESCRIPTION
- PRO_E_SIMPLFD_REP
- PRO_E_EXPLODE_STATE
- PRO_E_TIME_ESTIMATE
- PRO_E_COST_ESTIMATE

For a detailed explanation of explode states and related functions, see the section 'Exploded Assemblies' in the 'Assembly: Basic Assembly Access' chapter.

General Process Steps

The following two elements are used for general process steps (type PRO_PROCSTEP_GENERAL) only:

- PRO_E_GEN_STEP_TYPE—Default types, as well as user-defined types
- PRO_E_GEN_STEP_REFS—The following are valid reference types:
 - PRO_PART
 - PRO_FEATURE
 - PRO_SURFACE
 - PRO_EDGE
 - PRO_CURVE
 - PRO_AXIS
 - PRO_CSYS
 - PRO_POINT

Reposition Process Steps

The element PRO_E_POSITION is used for reposition process steps. It defines the transformation of the repositioned components.

Production Applications: NC Process Manager

Topic	Page
Overview	57 - 2
Accessing the Process Manager	57 - 4
Manufacturing Process Items	57 - 4
Manufacturing Features	57 - 12
Import and Export of Process Table Contents	57 - 13
Notification	57 - 13

Overview

The Process Manager functionality is based on the Process Table, which lists all the manufacturing process objects, such as workcells, operations, fixture setups, tooling, and NC sequences.

The Process Table lets you create new manufacturing objects and modify properties of existing ones. You can create new Holemaking and some milling steps directly in the Process Table. Steps of other types, that is, NC Sequences created outside the Process Manager, are also listed in the Process Table. You can manipulate the NC Sequences like any other steps, for example, reorder or duplicate them. However, customizations done to the toolpath outside the Process Table will be lost as a result of manipulation.

You can also define manufacturing templates based on existing Milling and Holemaking steps, and use these templates to create manufacturing steps in a different model.

You can add manufacturing templates as annotations to the reference part. This way, when you bring a reference part into a manufacturing model, the annotation can be automatically extracted into the manufacturing process to create the necessary steps, complete with tooling, parameters, and geometric references.

Accessing the Process Manager

You can access the Process Manager in Pro/ENGINEER in the Manufacturing Mode via Pro/TOOLKIT as follows:

- When the process manager dialog box is not active.
- When the process manager dialog box is active.

Accessing the Process Manager when the Dialog Box is not Active

You can access and modify the contents of the process manager without opening the Process Manager User Interface or while editing a manufacturing model in Pro/ENGINEER.

Functions Introduced:

- **ProMfgProctableEnable()**
- **ProMfgProctableDisable()**

The function **ProMfgProactableEnable()** ensures that the process manager database is prepared for access or modification by Pro/TOOLKIT functions. The application must call the function **ProMfgProactableEnable()** before using the process manager APIs discussed in the following sections.

The function **ProMfgProactableDisable()** enables Pro/ENGINEER to perform cleanup of data that was loaded while calling the function **ProMfgProactableEnable()**.

Accessing the Process Manager User Interface

You can add new items to the Process Manager User Interface using a Pro/TOOLKIT application. The Pro/TOOLKIT application should register the new user interface items in Pro/ENGINEER before you open the Process Manager dialog box. Once the dialog box is opened, Pro/ENGINEER automatically adds the registered menu buttons and menus to the dialog box.

Functions Introduced:

- **ProMfgproactableMenuAdd()**
- **ProMfgproactablePushbuttonAdd()**
- **ProMfgproactableItemAccessFunction**
- **ProMfgproactableItemActionFunction**
- **ProMfgproactableSelecteditemsGet()**
- **ProMfgproactableSelecteditemsSet()**
- **ProMfgproactableDisplayUpdate()**

The function **ProMfgproactableMenuAdd()** adds a new menu or submenu to an existing menu in the process manager user interface.

The function **ProMfgproactablePushbuttonAdd()** adds a new button at the end of the existing menu in the Process Manager user interface. When adding a new menu button, you must supply two callback functions whose signature matches **ProMfgproactableItemAccessFunction** and **ProMfgproactableItemActionFunction** respectively.

The function whose signature matches **ProMfgproactableItemAccessFunction** sets the access state for the button in the process manager dialog box. The function whose signature is similar to **ProMfgproactableItemActionFunction** provides the implementation for the new menu button.

Use the functions **ProMfgproctableSelecteditemsGet()** and **ProMfgproctableSelecteditemsSet()** in the context of menu buttons added to the Process Manager. These functions provide access to the steps, operations and workcells that are currently selected by the user in the Process Manager dialog box. This information can guide the application as to which items it should modify. Select the appropriate type of table for the selected item to be able to determine the subtype of the process item.

These functions can be used out of context of the menu addition APIs as well.

The function **ProMfgproctableDisplayUpdate()** refreshes the Process Manager dialog box. It allows the application to update the contents of the process table after some changes have been made during a custom menu button action.

Manufacturing Process Items

This section gives you more information on the handles that are provided to enable access to process item data.

Steps, Operations and Workcells

An NC sequence is an assembly feature that represents a single tool path. NC sequences listed in the Process Table are called steps.

An operation is a series of NC sequences performed in a particular workcell and using a particular coordinate system for output of Cutter Location data.

A workcell is a machine tool definition and is stored in the assembly.

The object `ProMfgprocItem` will be used for all functions that operate on steps, operations, and workcells. This item is a derivative of `ProModelitem` and is defined as

```
typedef struct pro_model_item
{
    ProType type;
    int id;
    ProMdl owner;
}ProMfgprocItem;
```

For all three item types, the `ProType` field should be `PRO_NC_PROCESS_ITEM`.

Step Models

The Step Model object represents the default properties of steps in the process manager and is a derivative of the `ProModelItem` and is defined as `typedef struct pro_model_item`.

```
{  
  
ProType type;  
  
int id;  
  
ProMdl owner;  
  
}ProMfgprocItem;
```

Step objects have a `step id`, the `Mfg owner`, and the `ProType` field as `PRO_NC_PROCESS_MODEL`.

Determining the Subtype of a Process Item

Function Introduced:

- **ProMfgprocitemSubtypeGet()**

The function **ProMfgprocitemSubtypeGet()** returns the subtype of a manufacturing process item. It could be one of the following:

- `PRO_MFGPROCITEM_STEP`—Specifies if the subtype is a step.
- `PRO_MFGPROCITEM_OPERATION`—Specifies if the subtype is an operation.
- `PRO_MFGPROCITEM_WORKCELL`—Specifies if the subtype is a workcell.

Accessing Details of Process Items

Although a `ProMfgprocItem` is not a feature, it shares many properties with `Pro/TOOLKIT` features. In fact, process items generally map one-to-one with a specified manufacturing feature. Because of this, many of the process item's properties can be accessed via the corresponding manufacturing feature element tree. Also, new items may be created by populating the element tree for the new step, operation or workcell.

Visiting Steps, Operations and Workcells

Functions Introduced:

- **ProMfgProctableVisit()**
- **ProMfgprocitemVisitAction()**
- **ProMfgprocitemFilterAction()**

The function **ProMfgProctableVisit()** enables you to visit the steps, workcells, and operations in a process table. As with other Pro/TOOLKIT visit functions, specify the visit action and visit filter functions. For more information on Visit Functions, refer to the chapter Fundamentals.

Creating Steps, Operations, and Workcells

Functions Introduced:

- **ProMfgprocitemCreate()**
- **ProMfgprocitemFromTemplateCreate()**

Use the function **ProMfgprocitemCreate()** to create a new step, operation, or workcell in the process table. The input arguments of this function are:

- *model*—Specifies the manufacturing model in which to create the new item.
- *table_type*—Specifies the type of table where the item will reside. The valid values are:
 - `PRO_MFGPROCTABLE_PROCESS`—Specifies the process table.
 - `PRO_MFGPROCTABLE_WORKCELLS`—Specifies the workcell table.
- *elem_tree*—Specifies the manufacturing feature element tree used to enumerate the properties of the new manufacturing item. To create a step you must fill one of the following element trees:
 - **Milling and Drilling**—The element tree documented in header file *ProNcseq.h*.
 - **Fixture**—The element tree documented in header file *ProFixture.h*.
 - **Assembly setup step**—The element tree documented in header file *ProFixture.h*.

To create an Operation you must fill out the element tree documented in header file *ProMfgoper.h*.

To create a Workcell you must fill the element tree documented in header file *ProWcell.h*.

- predecessor—Specifies the step or operation immediately before the new manufacturing item in the process table. When creating workcells, this can be NULL.

The function **ProMfgprocitemCreate()** returns the new manufacturing item.

Use the function **ProMfgprocitemFromTemplateCreate()** to create a new manufacturing step or workcell from a predefined template. A template is an XML file containing information necessary to create the manufacturing step or workcell, such as:

- Resources, that is, workcells, tools, and fixtures
- Ordered operations
- Ordered steps and their reference information such as manufacturing templates and manufacturing criteria.
- Retract plane for a step or an operation, as a distance calculated from the step coordinate system.

Accessing the Properties of Manufacturing Process Items

The functions described below enable you to access the properties of process items.

Functions Introduced:

- **ProMfgprocitemFeatureGet()**
- **ProMfgprocitemElemtreeGet()**
- **ProMfgprocitemElemtreeFree()**
- **ProMfgprocitemAnnotationGet()**
- **ProMfgprocitemReorderlimitsGet()**
- **ProMfgprocitemNextitemGet()**
- **ProMfgprocitemPreviousitemGet()**
- **ProMfgprocitemHolesetdepthGet()**
- **ProMfgprocitemHolesetdepthtypeGet()**
- **ProMfgprocitemHolesetdepthSet()**
- **ProMfgprocitemDefaultfixturesetupstepGet()**

The function **ProMfgprocitemFeatureGet()** returns the feature referenced by the process item. If the item is related to an actual manufacturing feature, some of the access functions that follow will not be supported. The properties of the item can be obtained by reading the feature's properties and element tree.

The function **ProMfgprocitemElemtreeGet()** obtains the element tree for a manufacturing item. This tree should be freed using **ProMfgprocitemElemtreeFree()**.

The function **ProMfgprocitemAnnotationGet()** returns the annotation element that created the process item.

The function **ProMfgprocitemReorderlimitsGet()** identifies the limits where the specified process item may be moved in the process table. The top limit represents the first process item after which you can insert the step. The lower limit, if any, represents the first process item that must come after the step in the process sequence.

The limits of reorder are based on step dependencies and priority and prerequisite rules. The input argument *options* specifies the checks to determine the position of reordering the process item elements. The valid values are:

- PRO_MFGPROCORDER_CHECK_PARENT_CHILD
- PRO_MFGPROCORDER_CHECK_PREREQUISITES
- PRO_MFGPROCORDER_CHECK_PRIORITIES

The function **ProMfgprocitemNextitemGet()** returns the process item following the specified item in the process table.

The function **ProMfgprocitemPreviousitemGet()** returns the process item preceding the specified process item in the process table.

The function **ProMfgprocitemHolesetdepthtypeGet()** returns the type of holeset and holeset end reference contained in the manufacturing process step.

The output argument *end_type* can have the following values:

- PRO_MFGSTEP_HOLESETEND_REFERENCE—Specifies the end reference surface for the hole depth.
- PRO_MFGSTEP_HOLESETEND_ALONG_AXIS—Specifies the end reference by entering a depth value along the hole axis.

The function **ProMfgprocitemHolesetdepthGet()** returns the process item holeset depth for a holeset of type BLIND and end reference type of ALONG_AXIS.

The function **ProMfgprocitemHolesetdepthSet()** sets the holeset depth for a step. This function supports steps with only one holeset. The holeset type must be BLIND and the end reference type will be changed to ALONG AXIS after using this function.

The function **ProMfgprocitemDefaultfixturesetupstepGet()** returns the fixture setup defined for the specified operation.

Modifying Process Items

Functions Introduced:

- **ProMfgprocitemRedefine()**
- **ProMfgprocitemDelete()**
- **ProMfgprocitemReorder()**

Use the function **ProMfgprocitemRedefine()** to redefine the manufacturing item created using an element tree.

The function **ProMfgprocitemDelete()** deletes the specified step, operation, or workcell from the manufacturing table.

The function **ProMfgprocitemReorder()** reorders a step or operation within the process table.

Parameters

Several categories of parameters are accessible for items in the process manager table:

- Manufacturing parameters
- AE parameters
- Step parameters
- Template parameters
- Special parameters
- Global Parameters

Manufacturing Parameters

You can access the manufacturing parameters by reading or writing the manufacturing feature or step element tree.

Annotation Element Parameters

You can access the annotation element parameters by getting the parent Annotation Element via **ProMfgprocitemAnnotationGet()** and reading parameters from that object. Refer to the chapter Parameters for details on how to read parameters from items such as annotation elements.

Step Parameters and Relations

Step Parameters are parameters assigned to the step object. You can define parameters and relations to a particular step in the process table.

You can access these parameters by reading them directly from the step object using the standard parameter functions.

Refer to the chapter Parameters for details on how to read parameters from items.

Template Parameters

Template parameters are inherited from the step's process template.

Function Introduced:

- **ProMfgprocitemTemplateparamGet()**

The function **ProMfgprocitemTemplateparamGet()** obtains a template parameter value for the process item.

Special Parameters

Other special parameters are properties of steps and can include the extract status, machining time, the template name, and so on. Some of these properties may be also accessible via the step element tree.

Functions Introduced:

- **ProMfgprocitemPropertyGet()**
- **ProMfgprocitemPropertySet()**

The function **ProMfgprocitemPropertyGet()** returns the value of the special property of the process manager step or operation. The special property can be as follows:

- **PRO_MFGPROP_EXTRACT_STATUS**—Specifies the extraction priority of the step.
- **PRO_MFGPROP_ACTUAL_MACHINING_TIME**—Specifies the machining time of a step.
- **PRO_MFGPROP_ACTUAL_MACHINING_LENGTH**—Specifies the machining distance of a step.
- **PRO_MFGPROP_TEMPLATE_NAME**—Specifies the name of the manufacturing template used.
- **PRO_MFGPROP_GROUP_LEVEL_1**—Specifies a merge group created by merging several Holmaking steps together, to optimize the tool path.
- **PRO_MFGPROP_GROUP_LEVEL_2**—Specifies a merge group created by merging **PRO_MFGPROP_GROUP_LEVEL_1** type merge groups together.

ProMfgprocitemPropertyGet() returns string values for **PRO_MFGPROP_GROUP_LEVEL_1** and **PRO_MFGPROP_GROUP_LEVEL_2** depending on the location of the process manager step in the merged group. The following table lists all the possible values taken by the special properties.

Step Location	Value returned for PRO_MFGPROP_ GROUP_LEVEL_1	Value returned for PRO_MFGPROP_ GROUP_LEVEL_2
If the step is a merge leader, but not a merge member.	“*leader”	Empty string
If the step is both a merge leader and merge member.	Name of the parent merge leader	“*leader”
If the step is merge member, but not a merge leader.		

Step Location	Value returned for PRO_MFGPROP_ GROUP_LEVEL_1	Value returned for PRO_MFGPROP_ GROUP_LEVEL_2
Case 1: It is a two level merge. For example, the parent itself is a merge member.	Name of the top merge leader	Name of the parent merge leader
Case 2: It is a one level merge	Name of the parent merge leader	Empty string
If the step is neither merge leader nor a merge member.	Empty string	Empty string

The function **ProMfgprocitemPropertySet()** sets the value of the special property of the process manager step or operation.

Global Parameters and Relations

Function Introduced:

- **ProMfgProctablemodelGet()**

The function **ProMfgProctablemodelGet()** obtains the process model that provides access to the manufacturing global parameters.

You can use this function to specify parameters and relations to all or selected steps in the process table. The `modelitem` handle returned by this function can be used to access the Global parameters of the process table.

Manufacturing Features

Functions Introduced:

- **ProMfgproctableFeaturesCreate()**
- **ProMfgproctableFeaturesDelete()**

The function **ProMfgproctableFeaturesCreate()** creates features for items in the process manufacturing table.

The function **ProMfgproctableFeaturesDelete()** deletes all existing manufacturing features from the Pro/ENGINEER model while keeping everything in the process manager unchanged.

Import and Export of Process Table Contents

Functions Introduced:

- **ProMfgProactableWrite()**
- **ProMfgProactableSynchronize()**

The function **ProMfgProactableWrite()** exports the contents of the manufacturing process table to CSV format. The input arguments of the function are:

- *mfg*—Specifies the manufacturing model.
- *table_type*—Specifies the type of table to be generated as output. The valid values are:
 - PRO_MFGPROCTABLE_PROCESS
 - PRO_MFGPROCTABLE_WORKCELLS
- *view_name*—Specifies the view name in the setup file; pass NULL to use the current view.
- *output_file*—Specifies the full path and name of the output file.

The function **ProMfgProactableSynchronize()** synchronizes the manufacturing table with the input CSV file.

Notification

Function Introduced:

- **ProMfgproactableExtractionPostAction**

The notification **ProMfgproactableExtractionPostAction** identifies when the user has extracted the process table in order to form manufacturing features. It provides the path to the extraction log file, which can be examined by the application for further information.

Example 1: To Add Menu Button to The Manufacturing Process Table

This example enables you to add the **Highlight Prerequisite** and the **Update Speed and Feed** options respectively, to the **Tools** menu in the Manufacturing Process Table.

```
/*=====*\n  FUNCTION : user_initialize()\n  PURPOSE  : \n/*=====*/
```

```

int user_initialize(
    int argc,
    char *argv[],
    char *version,
    char *build,
    wchar_t errbuf[80])
{
    uiCmdCmdId    cmd_id;

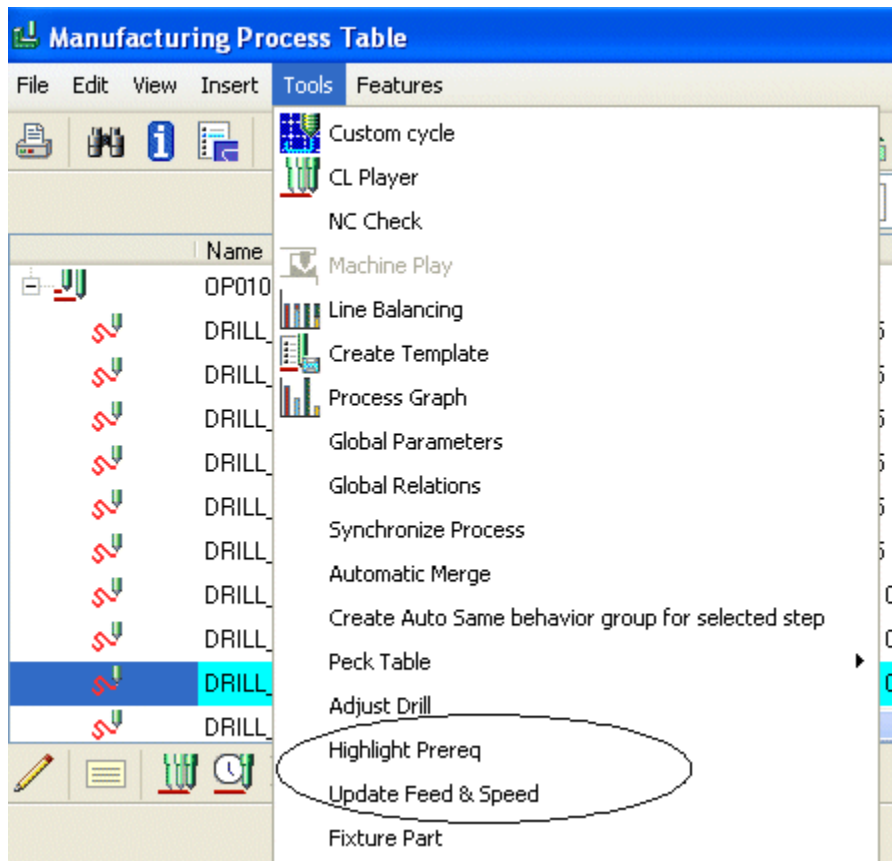
    PTUtilInitCounters();

    status = ProMfgproctablePushbuttonAdd ("Tools","Highlight Prereq",
    L"Highlight Prereq",
                                L"button_helptext",NULL,
                                PTMfgProctableItemActionPreReq,
                                "Callback on the new submenu added in
existing menu");
    PT_TEST_LOG_SUCC("ProMfgproctablePushbuttonAdd");

    status = ProMfgproctablePushbuttonAdd ("Tools","Update Feed & Speed",
    L"Update Feed & Speed",
                                L"button_helptext",NULL,
                                PTMfgProctableItemFeedSpeed,
                                "Callback on the new submenu added in
existing menu");
    PT_TEST_LOG_SUCC("ProMfgproctablePushbuttonAdd");

    return (PRO_TK_NO_ERROR);
}

```



The image above displays the **Highlight Prereq** and **Update Feed & Speed** menu buttons added to the **Tools** menu in the Manufacturing Process Table. When you click the **Highlight Prereq** option the function **PTMfgProactableItemActionPreReq** is invoked. The following example deals with this function in detail.

Example 2: To Highlight the Pre-requisites for the Selected Step in The Manufacturing Process Table

This example enables you to highlight the pre-requisites for the selected step in the Manufacturing Process Table.

```

/*=====*\
    FUNCTION :   PTMfgProactableItemActionPreReq
    PURPOSE  :   Execute Highlighting of Pre-Requisite steps
\*=====*/

ProError PTMfgProactableItemActionPreReq (ProMfg mfg, char* button_name,
ProAppData appdata)

```

```

{
    ProElement elem_tree;

    status = PTMfgProcitemElemtreeGet(&elem_tree);
    PT_TEST_LOG_SUCC("PTMfgProcitemElemtreeGet");

    if ( status == PRO_TK_NO_ERROR)
    {
        status = ProElemtreeWrite (elem_tree, PRO_ELEMTREE_XML,
L"selecteditem_prereq.inf");
        PT_TEST_LOG_SUCC (" ProElemtreeWrite ");

        if ( status == PRO_TK_NO_ERROR )
        {
            status = ProElemtreeElementVisit( elem_tree, ( ProElempath )NULL ,
                ( ProElemtreeVisitFilter) NULL,
                ( ProElemtreeVisitAction)
PTMfgElemtreePreReqAction,
                NULL );
            PT_TEST_LOG_SUCC (" ProElemtreeElementVisit ");
        }

    }

    return PRO_TK_NO_ERROR;
}

```

Manufacturing Process Table					
File Edit View Insert Tools Features					
	Name	Type	Workcell	Tool	Axes
	OP010	OPERATION	TOYODA		
	DRILL_DEPTH	STANDARD DRILLING		DRILL5	3 Axes
	DRILL_DEPTH_000	STANDARD DRILLING		DRILL5	3 Axes
	DRILL_TEST	STANDARD DRILLING		DRILL5	3 Axes
	DRILL_DEPTH_001	STANDARD DRILLING		DRILL5	3 Axes
	DRILL_DEPTH_002	STANDARD DRILLING		DRILL5	3 Axes
	DRILL_DEPTH_003	STANDARD DRILLING		DRILL5	3 Axes
	DRILL_REF	STANDARD DRILLING		DRILL10	3 Axes
	DRILL_REF_000	STANDARD DRILLING		DRILL10	3 Axes
	DRILL_REF_001	STANDARD DRILLING		DRILL10	3 Axes
	DRILL_REF_002				

Production
Applications: NC

Example 3: To Update Feed and Spindle Speed

When you click the **Update Speed and Feed** options in the **Tools** menu in the Manufacturing Process Table, the function **PTMfgProctableItemFeedSpeed** is invoked. This sample code enables you to change the cut feed and the spindle speed for the specified tool.

```

/*=====*\
  FUNCTION : PTMfgProctableItemFeedSpeed()
  PURPOSE  : Execute Feed/Speed
\*=====*/

ProError PTMfgProctableItemFeedSpeed (ProMfg mfg, char* button_name,
ProAppData appdata)
{
  ProAssembly r_solid_obj;
  ProMdl mfg_mdl_under_test;
  ProElempath epath;
  ProElement params_element;
  ProElempathItem p_items[3];
  ProElemId elem_id;
  ProValueDataType value_type = -1;

```

```

ProValue      value = (ProValue)NULL;
ProValueData  value_data;

status = ProMdlCurrentGet(&mfg_mdl_under_test);
PT_TEST_LOG_SUCC("ProMdlCurrentGet");

status = PTMfgProcitemElemtreeGet(&complete_tree);
PT_TEST_LOG_SUCC("PTMfgProcitemElemtreeGet");

if (status == PRO_TK_NO_ERROR)
    PTMfgProcitemToolNameGet(complete_tree);

if (status == PRO_TK_NO_ERROR)
{
    status = ProElempathAlloc(&epath);
    PT_TEST_LOG_SUCC("ProElempathAlloc()");

    p_items[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
    p_items[0].path_item.elem_id = PRO_E_MFG_PARAMS ;

    status = ProElempathDataSet(epath, p_items, 1);
    PT_TEST_LOG_SUCC("ProElempathDataSet()");

    status= ProElemtreeElementGet(complete_tree, epath,
&params_element);
    PT_TEST_LOG_SUCC("ProElemtreeElementGet()");

    if ( status == PRO_TK_NO_ERROR )
    {
        status = ProElemtreeElementVisit( params_element,NULL,
                                           ( ProElemtreeVisitFilter) NULL,
                                           ( ProElemtreeVisitAction)
PTMfgElemtreeWalkthroughAction,
                                           NULL );
        PT_TEST_LOG_SUCC (" ProElemtreeElementVisit ");
    }

    ProElempathFree(&epath);
}

status = ProMfgAssemGet(mfg_mdl_under_test,&r_solid_obj);
PT_TEST_LOG_SUCC("ProMfgAssemGet");

status = ProSolidFeatVisit ((ProSolid)r_solid_obj,
                             (ProFeatureVisitAction)PTMfgMaterialGet,
                             NULL,
                             NULL);
PT_TEST_LOG_SUCC("ProSolidFeatureVisit()");

```



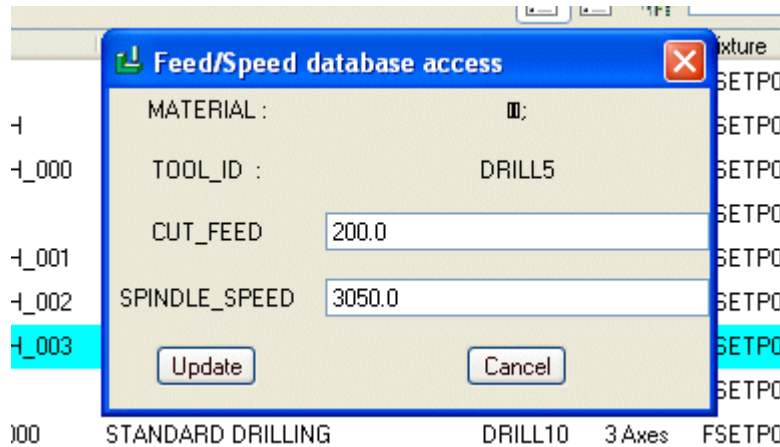
```

status = PTMfgDialogCreate(params_element);

PTUtilInitCounters();

return PRO_TK_NO_ERROR;
}

```



Production
Applications: NC

The section below describes the action functions for the each of the preceding sample code.

```

/*****ACTION FUNCTIONS*****/

/*-----*\
FUNCTION : PTMfgElemtreeWalkthroughAction()
PURPOSE  : Action function for PTUtilElemtreeWalkthroug
\*-----*/

ProError PTMfgElemtreeWalkthroughAction ( ProElement params_element,
                                           ProElement element, ProElemPath elem_path,
                                           ProAppData app_data )
{
    ProElemPathItem p_items[3];
    ProValueDataType value_type = -1;
    ProElemId elem_id, p_elem_id;
    char * actual_value;
    ProValue value = (ProValue) NULL;
    ProValueData value_data;
    Data_Input *my_data;
    ProElemPath epath;

    my_data = (Data_Input*) app_data;

```

```

status = ProElementIdGet( element, &elem_id);
PT_TEST_LOG_SUCC("ProElementIdGet");

if ( elem_id == PRO_E_MFG_PARAM_NAME )
{
    status = ProElementValuetypeGet ( element, &value_type );
    PT_TEST_LOG(" ProElementValuetypeGet ", status,
        status != PRO_TK_NO_ERROR && status !=
PRO_TK_INVALID_TYPE &&
        status != PRO_TK_EMPTY );

    if(value_type ==PRO_VALUE_TYPE_STRING)
    {
        status = ProElementStringGet( element,( ProElementStringOptions)
NULL, &actual_value);
        PT_TEST_LOG_SUCC ( " ProElementStringGet ");

        if ( (strcmp ( "CUT_FEED",actual_value) == 0)
|| ( strcmp ( "SPINDLE_SPEED",actual_value) == 0))
        {

            status = ProElempathAlloc(&epath);
            PT_TEST_LOG_SUCC ("ProElempathAlloc (");

            p_items[0].type = PRO_ELEM_PATH_ITEM_TYPE_INDEX;
            p_items[0].path_item.elem_index = count;
            p_items[1].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
            p_items[1].path_item.elem_id = PRO_E_MFG_PARAMVAL;

            status = ProElempathDataSet(epath, p_items, 2);
            PT_TEST_LOG_SUCC ("ProElempathDataSet()");

            status= ProElemtreeElementGet(params_element, epath,
&p_element);
            PT_TEST_LOG_SUCC ("ProElemtreeElementGet()");

            status = ProElementIdGet( p_element, &p_elem_id);
            PT_TEST_LOG_SUCC ( " ProElementIdGet ");

            if (p_elem_id ==PRO_E_MFG_PARAMVAL)
            {

                status = ProElementValueGet ( p_element, &value);
                PT_TEST_LOG_SUCC("ProElementValueGet");

                status = ProValueDataGet (value, &value_data);
                PT_TEST_LOG("ProValueDataGet()", status, (status !=
PRO_TK_NO_ERROR));

                if (strcmp ( "CUT_FEED",actual_value) == 0)
                {

```

```

        cut_feed = value_data.v.d;

        if (FLAG == 1)
        {
            value_data.v.d = my_data->cut_feed;
            status = ProValueDataSet (value, &value_data);
            PT_TEST_LOG("ProValueDataSet()", status, (status !=
PRO_TK_NO_ERROR));

            status = ProElementValueSet ( p_element, value);
            PT_TEST_LOG_SUCC("ProElementValueGet");

        }
    }

    if (strcmp ( "SPINDLE_SPEED",actual_value) == 0)
    {
        spindle_speed = value_data.v.d;

        if (FLAG == 1)
        {
            value_data.v.d = my_data->spindle_speed;
            status = ProValueDataSet (value, &value_data);
            PT_TEST_LOG("ProValueDataSet()", status, (status !=
PRO_TK_NO_ERROR));

            status = ProElementValueSet (p_element, value);
            PT_TEST_LOG_SUCC("ProElementValueGet");

        }
    }
    count++;
}

ProElempathFree(&epath);

}

}

status = ProMfgproctableDisplayUpdate ();
PT_TEST_LOG_SUCC ("ProMfgproctableDisplayUpdate");

}

return PRO_TK_NO_ERROR;
}

```

```

/*-----*\
FUNCTION : PTMfgUsrUpdateAction
PURPOSE  : Action function for the File Selection
\*-----*/

void PTMfgUsrUpdateAction(char *dialog, char *component, ProAppData data)
{
    ProElement params_element;
    ProCharName str;
    ProCharName str2;
    double cut_feed_value;
    double spindle_speed_value;
    Data_Input data1;
    wchar_t* wstr;
    wchar_t* wstr2;
    ProErrorlist p_errors;

    params_element = (ProElement) data;

    ProUIInputpanelValueGet(DIALOG_NAME, "cut_feed", &wstr);
    ProWstringToString(str, wstr);
    cut_feed_value= atof(str);

    ProUIInputpanelValueGet(DIALOG_NAME, "spindle_speed", &wstr2);
    ProWstringToString(str2, wstr2);
    spindle_speed_value= atof(str2);

    data1.cut_feed = cut_feed_value;
    data1.spindle_speed = spindle_speed_value;

    count =0;
    FLAG =1;

    status = ProElemtreeWrite (params_element, PRO_ELEMTREE_XML,
L"before_redef.inf");
    PT_TEST_LOG_SUCC (" ProElemtreeWrite ");

    status = ProElemtreeElementVisit( params_element, ( ProElempath )NULL ,
                                     ( ProElemtreeVisitFilter) NULL,
                                     ( ProElemtreeVisitAction)
PTMfgElemtreeWalkthroughAction,
                                     (ProAppData)&data1 );
    PT_TEST_LOG_SUCC (" ProElemtreeElementVisit ");

    status = ProMfgprocitemRedefine (sel_items,complete_tree,&p_errors);
    PT_TEST_LOG_SUCC ("ProMfgprocitemRedefine");

    status = ProMfgproctableDisplayUpdate ();
    PT_TEST_LOG_SUCC ("ProMfgproctableDisplayUpdate");
}

```

```

PTUtilInitCounters();

status = ProUIDialogExit(DIALOG_NAME, 0);
PT_TEST_LOG_SUCC("ProUIDialogExit");
}

/*-----*\
FUNCTION : PTMfgUsrCancelAction
PURPOSE  : Action function for the Directory Selection
\*-----*/

void PTMfgUsrCancelAction(char *dialog, char *component, ProAppData data)
{
    ProError status;

    PTUtilInitCounters();

    status = ProUIDialogExit(DIALOG_NAME, 0);
    PT_TEST_LOG_SUCC("ProUIDialogExit");
}

/*-----*\
FUNCTION : PTMfgElemtreePreReqAction()
PURPOSE  : Get the material
\*-----*/

ProError PTMfgMaterialGet (ProFeature* feat, ProAppData data)
{
    ProAsmcompType f_type;
    ProMaterial p_material;
    ProFeattype feat_type;
    ProSolid part;
    ProMdl p_mdl_handle;
    ProCharName c_name;
    ProMdl mfg_mdl_under_test;

    status = ProMdlCurrentGet(&mfg_mdl_under_test);
    PT_TEST_LOG_SUCC("ProMdlCurrentGet");

    status = ProFeatureTypeGet(feat, &feat_type);
    PT_TEST_LOG_SUCC("ProFeatureTypeGet");

    if(feat_type==PRO_FEAT_COMPONENT)
    {
        status = ProAsmcompTypeGet ((ProAsmcomp*)feat, feat->owner, &f_type);
        PT_TEST_LOG_SUCC("ProAsmcompTypeGet");

        if ( f_type == PRO_ASM_COMP_TYPE_REF_MODEL)
        {
            status = ProAsmcompMdlGet ((ProAsmcomp*)feat, &p_mdl_handle);

```

```

        PT_TEST_LOG_SUCC("ProAsmcompMdlGet");

        status = ProMaterialCurrentGet
((ProSolid)p_mdl_handle,&p_material);
        PT_TEST_LOG_SUCC("ProMaterialCurrentGet");

        ProWstringToString (c_name, p_material.matl_name);

ProWstringCopy(p_material.matl_name,material_name,PRO_VALUE_UNUSED);

    }

    }

    return PRO_TK_NO_ERROR;
}

/*-----*\
    FUNCTION :   PTMfgElemtreePreReqAction()
    PURPOSE  :   Action function for PTUtilElemtreeWalkthroug
\*-----*/

ProError PTMfgElemtreePreReqAction ( ProElement elem_tree,
                                     ProElement element, ProElempath elem_path,
                                     ProAppData app_data )
{
    ProElemId elem_id;
    ProValueData value_data;
    int ii, counter = 0;
    ProValue *values = (ProValue *)NULL;
    ProMfgprocItem *item_array;
    ProMfgprocItem item;
    ProAssembly r_solid_obj;
    ProElement elem_tree_p;

    ProMdl mfg_mdl_under_test;

    status =ProMdlCurrentGet(&mfg_mdl_under_test);
    PT_TEST_LOG_SUCC("ProMdlCurrentGet");

    status = ProMfgAssemGet(mfg_mdl_under_test,&r_solid_obj);
    PT_TEST_LOG_SUCC ("ProMfgAssemGet");

    status = ProElementIdGet( element, &elem_id);
    PT_TEST_LOG_SUCC("ProElementIdGet");

    if (elem_id == PRO_E_NCSEQ_PREREQUISITE_ARR)
    {

```

```

        status = ProArrayAlloc( 0, sizeof(ProValue), 1, (ProArray *)&values
);
        PT_TEST_LOG_SUCC("ProArrayAlloc");

        status = ProArrayAlloc( 0, sizeof(ProMfgprocItem), 1, (ProArray
*)&item_array );
        PT_TEST_LOG_SUCC("ProArrayAlloc(");

        status = ProElementValuesGet(element, &values );
        PT_TEST_LOG_SUCC("ProArrayAlloc");

        if( status == PRO_TK_NO_ERROR )
            status = ProArraySizeGet( (ProArray)values, &counter );

        for( ii = 0; ii < counter; ii++ )
        {

            if( ( ProValueDataGet( values[ii], &value_data ) ==
PRO_TK_NO_ERROR )
                && ( value_data.type == PRO_VALUE_TYPE_INT ) )
            {
                item.type = PRO_NC_STEP_OBJECT;
                item.id = value_data.v.i;
                item.owner = r_solid_obj;
            }

            status = ProArrayObjectAdd((ProArray*)&item_array,-1,1,&item);
            PT_TEST_LOG_SUCC("ProArrayObjectAdd");
        }

        status = ProMfgproctableSelecteditemsSet
(PRO_MFGPROCTABLE_PROCESS,item_array);
        PT_TEST_LOG_SUCC ("ProMfgproctableSelecteditemsSet");

        status = ProMfgproctableDisplayUpdate ();
        PT_TEST_LOG_SUCC ("ProMfgproctableDisplayUpdate");

        ProArrayFree ( (ProArray *)&values );
        ProArrayFree ( (ProArray *)&item_array );
    }

    return PRO_TK_NO_ERROR;
}
/*-----*\
    FUNCTION : PTMfgDialogCreate
    PURPOSE  : Creates the dialog
\*-----*/

ProError PTMfgDialogCreate(ProElement params_element)
{

```

```

int choice;
ProCharName str;
ProName wstr;

status = ProUIDialogCreate(DIALOG_NAME,DIALOG_NAME);
PT_TEST_LOG_SUCC("ProUIPTMfgDialogCreate");

status = ProUIDialogWidthSet(DIALOG_NAME,300);
PT_TEST_LOG_SUCC("ProUIPTMfgDialogCreate");

status = ProUIDialogHeightSet(DIALOG_NAME,160);
PT_TEST_LOG_SUCC("ProUIPTMfgDialogCreate");

status = ProUIDialogTitleSet (DIALOG_NAME,L"Feed/Speed database
access");
PT_TEST_LOG_SUCC ("ProUIDialogTitleSet()");

status = ProUILabelTextSet (DIALOG_NAME, "tool_id_val", tool_id);
PT_TEST_LOG_SUCC ("ProUILabelTextSet()");

status = ProUILabelTextSet (DIALOG_NAME, "material_val", material_name);
PT_TEST_LOG_SUCC ("ProUILabelTextSet()");

sprintf(str,"%f",cut_feed);
ProStringToWstring(wstr, str);

ProUIInputpanelValueSet(DIALOG_NAME,"cut_feed",wstr);
PT_TEST_LOG_SUCC("ProUIInputpanelValueSet");

sprintf(str,"%f",spindle_speed);
ProStringToWstring(wstr, str);

ProUIInputpanelValueSet(DIALOG_NAME,"spindle_speed",wstr);
PT_TEST_LOG_SUCC("ProUIInputpanelValueSet");

status = ProUIPushbuttonActivateActionSet(DIALOG_NAME, UPDATE,
PTMfgUsrUpdateAction, params_element);
PT_TEST_LOG_SUCC("ProUIPushbuttonActivateActionSet");

status = ProUIPushbuttonActivateActionSet(DIALOG_NAME, CANCEL,
PTMfgUsrCancelAction, NULL);
PT_TEST_LOG_SUCC("ProUIPushbuttonActivateActionSet");

status = ProUIDialogActivate(DIALOG_NAME, &choice);
PT_TEST_LOG_SUCC("ProUIDialogActivate");

status = ProUIDialogDestroy(DIALOG_NAME);
PT_TEST_LOG_SUCC("ProUIDialogDestroy");
}
/*-----*\
FUNCTION : PTMfgProcitemElemtreeGet

```



```

    PURPOSE : Gets the elem tree of selected item
\*-----*/

ProError PTMfgProcitemElemtreeGet(ProElement *elem_tree)
{

    status = ProMfgproctableSelecteditemsGet
(PRO_MFGPROCTABLE_PROCESS,&sel_items);
    PT_TEST_LOG_SUCC ("ProMfgproctableSelecteditemsGet");

    /*TBD : Currently one one ( fitst item is checked here...
    To be updated with multiple selections of items in ui ..*/

    status = ProMfgprocitemElemtreeGet (&sel_items[0], elem_tree);
    PT_TEST_LOG_SUCC("ProMfgprocitsmElemtreeGet");

    return status;

}

/*-----*\
    FUNCTION : PTTestMfgProcItemVisit
    PURPOSE : Action function for ProMfgProctableVisit
\*-----*/

ProError PTTestMfgProcItemVisit ( ProMfgprocItem* item, ProError error,
ProAppData app_data)
{

    ProAnnotation annotation;
    ProMfgstepHolesetEndType end_type;
    ProDrillDepthType depth_type;

    status = ProMfgprocitemAnnotationGet (item, &annotation);
    PT_TEST_LOG_SUCC ("ProMfgprocitemAnnotationGet");

    if ( status == PRO_TK_NO_ERROR)
    {
        status = ProParameterVisit
(&annotation,NULL,PTMfgParameterVisit,"HOLE_ADJUST");
        PT_TEST_LOG_SUCC("ProParameterVisit");
    }

    if ( found == PRO_B_TRUE)
    {
        status = ProParameterVisit
(&annotation,NULL,PTMfgParameterVisit,"HOLE_DEP_1");
        PT_TEST_LOG_SUCC("ProParameterVisit");
    }
}

```

```

        status = ProParameterVisit
(&annotation,NULL,PTMfgParameterVisit,"HOLE_DEP_2");
        PT_TEST_LOG_SUCC("ProParameterVisit");

    }

    status = ProMfgprocitemHolesetdepthtypeGet
(item,&depth_type,&end_type);
    PT_TEST_LOG_SUCC("ProMfgprocitemHolesetdepthtypeGet");

    if ((end_type == PRO_MFGSTEP_HOLESETEND_ALONG_AXIS) && (depth_type
==PRO_DRILL_BLIND))

    {

        status = ProMfgprocitemHolesetdepthSet (item ,hole_dep_1);
        PT_TEST_LOG_SUCC ("ProMfgprocitemHolesetdepthSet");
    }

    if (end_type ==PRO_MFGSTEP_HOLESETEND_REFERENCE)
    {
        status = ProMfgprocitemHolesetdepthSet (item ,hole_dep_2);
        PT_TEST_LOG_SUCC ("ProMfgprocitemHolesetdepthSet");
    }

    return PRO_TK_NO_ERROR;

}

/*-----*\
    FUNCTION :   PTMfgParameterVisit
    PURPOSE   :   Visit the param.
\*-----*/

ProError PTMfgParameterVisit(ProParameter* param, ProError status,
ProAppData data)
{

    ProCharName str;
    ProParamvalue value;
    ProParamvalueType type;
    short l_val;

    ProWstringToString(str,param->id);

    if ( strcmp (str, data) == 0)

    {
        status = ProParameterValueGet (param, &value);
        PT_TEST_LOG_SUCC ("ProParameterValueGet()");
    }

```

```

status = ProParamvalueTypeGet (&value, &type);
PT_TEST_LOG_SUCC ("ProParamvalueTypeGet()");

if (type == PRO_PARAM_BOOLEAN)
{
    status = ProParamvalueValueGet (&value, type, (void *) &l_val);
    PT_TEST_LOG_SUCC ("ProParamvalueValueGet()");

    if (l_val == 1)
        found = PRO_B_TRUE;

}

if (type == PRO_PARAM_DOUBLE)
{
    if ( strcmp (str,"HOLE_DEP_1") == 0)
    {
        status = ProParamvalueValueGet (&value, type, (void *)
&hole_dep_1);
        PT_TEST_LOG_SUCC ("ProParamvalueValueGet()");
    }

    if ( strcmp (str,"HOLE_DEP_2") == 0)
    {
        status = ProParamvalueValueGet (&value, type, (void *)
&hole_dep_2);
        PT_TEST_LOG_SUCC ("ProParamvalueValueGet()");
    }

}

}

return PRO_TK_NO_ERROR;
}
/*-----*\
    FUNCTION : PTMfgProcitemToolNameGet
    PURPOSE  : Get the tool name from elem tree.
\*-----*/

ProError PTMfgProcitemToolNameGet(ProElement elem)
{
    ProElempath epath;
    ProElement tool_element;
    ProElempathItem p_items[3];
    ProElemId elem_id;
    ProValueDataType value_type = -1;
    ProValue value = (ProValue)NULL;
    ProValueData value_data;
    char toolname[100];

```

```

    status = ProElemtreeWrite (elem, PRO_ELEMTREE_XML,
L"selecteditem_feed.inf");
    PT_TEST_LOG_SUCC (" ProElemtreeWrite ");

    status = ProElempathAlloc(&epath);
    PT_TEST_LOG_SUCC ("ProElempathAlloc ()");

    p_items[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
    p_items[0].path_item.elem_id = PRO_E_TOOL ;

    status = ProElempathDataSet(epath, p_items, 1);
    PT_TEST_LOG_SUCC ("ProElempathDataSet()");

    status= ProElemtreeElementGet(complete_tree, epath, &tool_element);
    PT_TEST_LOG_SUCC ("ProElemtreeElementGet()");

    status = ProElementIdGet( tool_element, &elem_id);
    PT_TEST_LOG_SUCC("ProElementIdGet");

    if ( elem_id == PRO_E_TOOL )
    {
        status = ProElementValueGet (tool_element, &value);
        PT_TEST_LOG_SUCC("ProElementValueGet");

        status = ProValueDataGet (value, &value_data);
        PT_TEST_LOG_SUCC("ProValueDataGet()");

        if (value_data.v.w == NULL)
            ProStringToWstring(tool_id, "TOOL_NOT_SET");

        else
            ProWstringCopy(value_data.v.w, tool_id, PRO_VALUE_UNUSED);
    }
    ProElempathFree(&epath);

    return PRO_TK_NO_ERROR;

}
/*=====*\
    FUNCTION : user_terminate()
    PURPOSE  : To handle any termination actions
/*=====*/

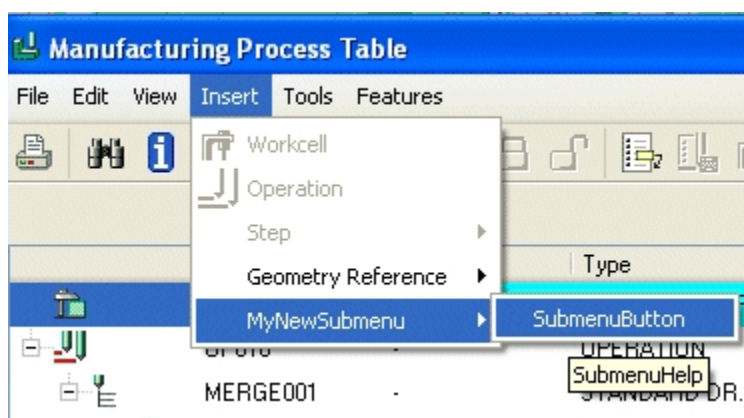
void user_terminate()
{
}

```

Example 4: To Add a Submenu to the Manufacturing Process Table

The following example enables you to add a menu and submenu to an existing menu option in the Manufacturing Process Table.

```
#include <ProMenuBar.h>
#include <ProMfgproctable.h>
#include <ProMfg.h>
#include <ProMdl.h>
#include <ProMessage.h>
#include <ProUtil.h>
ProName submenulabel;
ProName help;
ProName submenubutton;
ProError APITestCommand (ProMfg mfg, char* button_name,
                        ProAppData appdata);
ProStringToWstring (submenulabel, "MyNewSubmenu");
ProStringToWstring (submenubutton, "SubmenuButton");
ProStringToWstring (help, "SubmenuHelp");
status = ProMfgproctableMenuAdd ( "APITestMenu",
submenulabel, "Insert" );
PT_TEST_LOG_SUCC("ProMfgproctableMenuAdd")
status = ProMfgproctablePushbuttonAdd ("APITestMenu", "APITestMain",
submenubutton,
help, NULL, APITestCommand, NULL );
PT_TEST_LOG_SUCC("ProMfgproctablePushbuttonAdd");
ProError APITestCommand (ProMfg mfg, char* button_name, ProAppData
appdata) {
    printf("Just some output\n");
    return PRO_TK_NO_ERROR;
}
```



Production Applications: Diagrams and Cabling

This chapter describes the Pro/TOOLKIT diagram and cabling functions.

Topic	Page
Diagrams	58 - 2
Pro/ENGINEER Cabling Design	58 - 6

Diagrams

This section describes the functions used to access objects in Diagram mode. The section contains the following subsections:

- Listing the Contents of a Diagram on page 58 - 2
- Diagram Parameters on page 58 - 4
- Diagram Wire Spools on page 58 - 5
- Diagram Connectivity on page 58 - 5

Listing the Contents of a Diagram

Functions Introduced:

- **prodgm_get_ids()**
- **prodgm_get_nodes()**
- **prodgm_get_component_outl()**

The following four types of items on a diagram can be accessed using Pro/TOOLKIT:

- Components
- Connectors
- Spools
- Wires

Items of all four types are identified from within Pro/TOOLKIT by integer identifiers that are preserved across sessions of Pro/ENGINEER. The function **prodgm_get_ids()** provides an array that contains the identifiers of the items of a specified type on all sheets of the diagram. Specify the type using one of the following enumerated values:

- PRODGM_COMPONENT
- PRODGM_CONNECTOR
- PRODGM_SPOOL
- PRODGM_WIRE

The *REF_DES* and *MODEL_NAME* of components and connectors are stored as parameters and can be accessed using the function **prodgm_get_parameters()**. You can read and modify the names of wires using the functions **proddb_get_element_name()** and **proddb_set_element_name()**.

Components and connectors also contain nodes—the points at which wires can connect. These are identified by names that are unique within the owning component or connector. The function **prodgm_get_nodes()** returns an array of node names for a specified component or connector. The component or connector is specified using the structure *Prodgm_item*, which is defined as follows:

```
typedef struct prodgm_item
{
    Prodgm_type  type;
    int          id;
} Prodgm_item;
```

In this structure, the *type* field can be `PRODGM_COMPONENT` or `PRODGM_CONNECTOR`, and *id* is one of the identifiers in the array returned by.

Function **prodgm_get_component_outl()** returns the page number and four corners of the specified item, corrected for rotation. This function supports only items of type `PRODGM_CONNECTOR` and `PRODGM_COMPONENT`.

The following section describes how to find out which wires connect to which components and connectors.

Diagram Item Color and Line Style

Functions Introduced:

- **prodgm_get_color()**
- **prodgm_set_color()**
- **prodgm_get_linestyle()**
- **prodgm_set_linestyle()**
- **prodgm_get_rgbcolor()**
- **prodgm_set_rgbcolor()**

Functions **prodgm_get/set_color()** set or return the color of the specified diagram item. Note that these functions do not support items of type `PRODGM_SPOOL`.

Use functions **prodgm_get/set_linestyle()** set or return the linestyle of the specified diagram item. Note that these functions support only the following item types:

- `PRODGM_WIRE`
- `PRODGM_RAIL`

- PRODGM_HIGHWAY

Functions **prodgm_get/set_color()** set or return the color of the specified diagram item. Note that these functions do not support items of type PRODGM_SPOOL.

Functions **prodgm_get/set_rgbcolor()** set or return the color of the specified diagram item. Note that these functions do not support items of type PRODGM_SPOOL.

Diagram Parameters

Functions Introduced:

- **prodgm_get_parameters()**
- **prodgm_set_parameters()**
- **prodgm_delete_parameters()**

The diagram parameter functions operate on components, connectors, nodes, and wires, and are equivalent to the option **Modify Parameters** in the DIAGRAM menu in Pro/ENGINEER. In Pro/ENGINEER, the node parameters (SIGNAL_NAME and SIGNAL_VALUE) for all the nodes are available when you select the owning component or connector, but in Pro/TOOLKIT you access the node parameters by specifying the individual node.

The input arguments are a structure of type *Prodgm_item* that identifies the item, and an optional node name. Pass NULL for the node name if you want to access the component or connector parameters.

Function **prodgm_get_parameters()** returns an array of parameters for the specified component, connector, spool, or node.

Function **prodgm_set_parameters()** sets a value for the specified component or connector parameter. Specify the argument node in argument *PRODEV_name* if you change the node parameters.

Note that **prodgm_set_parameters()** is used to set the value of an existing, and therefore predefined, parameter. You cannot add new user-defined parameters of your own.

Also note the following special cases:

- The GENDER attribute on components and connectors is an integer as seen from Pro/TOOLKIT (1 is MALE, 0 is FEMALE) and *cannot* be modified using the function **prodgm_set_parameters()**.

- The OBJ_TYPE attribute on components and connectors is an integer as seen from Pro/TOOLKIT (41 is a COMPONENT, and 42 is a CONNECTOR) and *cannot* be modified using the function **prodgm_set_parameters()**.

Use function **prodgm_delete_parameters()** to delete parameters from a specified diagram item.

Diagram Wire Spools

Functions Introduced:

- **prodgm_get_wire_spool()**
- **prodgm_set_wire_spool()**
- **prodgm_get_run_spool()**
- **prodgm_set_run_spool()**
- **prodgm_get_conds()**

Spools are identified by integer identifiers in Pro/TOOLKIT. The **prodgm_get/set_wire_spool()** functions enable you to get and set the spool used by a specified wire.

Function **prodgm_get_conds()** returns conductor names for the specified diagrams.

The functions **prodgm_set_run_spool()** and **prodgm_get_run_spool()** set or return the identifier of the spool associated with the specified wire or cable.

Diagram Connectivity

Functions Introduced:

- **prodgm_get_wire_nodes()**
- **prodgm_get_node_connections()**

The function **prodgm_get_wire_nodes()** provides a list of the component and connector nodes to which a specified wire connects. The nodes are described by an array of type *Prodgm_node_ref*, defined as follows:

```
typedef struct prodgm_node_ref
{
    Prodgm_type  owner_type; /* Either PRODGM_COMPONENT or
                             PRODGM_CONNECTOR */
    int          owner_id;   /* The identifier of the component or
                             connector */
    PRODEV_Name  node_name; /* The name of the node */
}
```

```
} Prodgm_node_ref;
```

The function **prodgm_get_node_connections()** provides a list of the other components and connectors that are connected to a specified component or connector node by wires. The input node is specified using the structure *Prodgm_node_ref*, and the output is an array of type *Prodgm_item*.

Pro/ENGINEER Cabling Design

This section describes the functions in Pro/TOOLKIT that access the contents of a cabling harness created by the Pro/ENGINEER module Pro/ENGINEER Cabling Design. The explanations in this section assume a knowledge of Pro/ENGINEER Cabling Design and the fundamental concepts of Pro/TOOLKIT, especially assemblies.

Be careful with the terminology of Pro/ENGINEER Cabling design. You can route three kinds of objects to make electrical connections within a harness: wires, cables, and bundles. The generic word for all three is cable. Unless otherwise specified, the explanations in the following sections use the word cable, in its generic sense, to include wires, cables, and bundles.

Pro/TOOLKIT uses the object type ProCable to refer to any type of cable object. The data object has the same structure as ProModelitem.

```
Typedef struct pro_model_item {  
    ProMdl owner;  
    int id;  
    prototype type;  
} ProCable;
```

Creating a Harness

Function Introduced:

- **ProHarnessCreate()**

ProCable operates on a Pro/ENGINEER assembly. Cables can belong to one or more harness. A harness is a special Pro/ENGINEER part designed to contain cables.

A harness cannot be retrieved into Pro/ENGINEER Part mode. It appears in the cabling assembly as an assembly component. A harness is identified by the Pro/ENGINEER object handle ProHarness, which is an alternate name for Pro/Part.

The function **ProHarnessCreate()** creates a new harness in the specified assembly.

Finding a Harness

Function Introduced:

- **ProAssemblyHarnessesCollect()**

The function **ProAssemblyHarnessesCollect()** returns an array of handles to any harness that is part of a specified assembly. To find harnesses at lower levels in the assembly hierarchy, traverse the subassemblies to find each one that contains a harness directly.

Finding the Cables in a Harness

Functions Introduced:

- **ProHarnessCablesCollect()**
- **ProCableHarnessesGet()**
- **ProInputFileRead()**

The function **ProHarnessCablesCollect()** provides an array of names of cables that exist in the specified harness. The output of the function includes cables that have not yet been routed.

Each cable can be created in, and routed through, several harnesses. The function **ProCableHarnessesGet()** provides an array of the handles to the harnesses below the current assembly that contain a cable with the specified name.

Use function **ProInputFileRead()** with argument **PRO_WIRELIST_TYPE** to read files in Mentor Graphics LCABLE format. This function does not create wires, but provides parameters from a wire list for use when creating in a harness assembly a wire with the same name as that in the LCABLE file.

Managing Spools

Functions Introduced:

- **ProAssemblySpoolsCollect()**
- **ProSpoolCreate()**
- **ProInputFileRead()**
- **ProOutputFileWrite()**

Function **ProAssemblySpoolsCollect()** returns a list of all spools defined in the specified assembly.

Use function **ProSpoolCreate()** to create a new spool of a given cable and sheath type in the specified assembly.

Use function **ProInputFileRead()** with argument PRO_SPOOL_FILE to create new spools or update existing ones. Function **ProOutputFileWrite()** with the same argument to export a spool file.

Spool Parameters

Functions Introduced:

- **ProSpoolParameterGet()**
- **ProSpoolParametersCollect()**
- **ProSpoolParameterDelete()**
- **ProSpoolParametersSet()**
- **ProSpoolsFromLogicalGet()**
- **ProSpoolsFromLogicalCreate()**

Function **ProSpoolParameterGet()** retrieves a single parameter for the specified spool. This function supports only single-valued parameters. If you specify a multivalued parameter, the function returns PRO_TK_E_NOT_FOUND.

Function **ProSpoolParameterCollect()** retrieves all parameters of the specified spool, both single- and multi-valued parameters.

Use **ProSpoolParameterDelete()** to remove a single parameter from the specified spool. This function deletes both single- and multi-valued parameters.

Function **ProSpoolParametersSet()** sets all parameters of the specified spool, both single- and multi-valued parameters. This function overwrites existing parameter values with values in the input parameter array.

Function **ProSpoolsFromLogicalGet()** returns a list of spool names in the specified assembly for which data has been imported from a logical reference but which have not yet been created. Use function **ProSpoolsFromLogicalCreate()** to create instances of spools for which logical data exists. Refer to the ProCable and Pro/HARNESS-MFG documentation for more on logical references.

Finding Harness Connectors

Functions Introduced:

- **ProAssemblyConnectorsGet()**
- **ProConnectorsFromLogicalGet()**

Each connector in a cabling assembly is a Pro/ENGINEER part that is a component of that assembly. A connector can be at any level in the assembly hierarchy. Each connector is identified by its assembly component path (*ProAsmcomppath*).

The function **ProAssemblyConnectorGet()** provides an array of member identifier tables identifying the connectors in the specified assembly. The function allocates the memory for these tables. Function **ProConnectorsFromLogicalGet()** returns a list of connector names in the specified assembly for which data has been imported from a logical reference but which have not yet been created. Refer to the ProCable and Pro/HARNESS-MFG documentation for more on logical references.

Connectors Parameters

Functions Introduced:

- **ProConnectorEntryPortsGet()**
- **ProConnectorParamsCollect()**
- **ProConnectorDesignate()**
- **ProConnectorUndesignate()**
- **ProConnectorParamGet()**
- **ProConnectorParamDelete()**
- **ProConnectorParamsSet()**
- **ProConnectorsFromLogicalGet()**
- **ProOutputFileWrite()**
- **ProInputFileRead()**

Each connector contains a set of entry ports to which cables can be connected. Each entry port is modeled by a coordinate system datum that belongs to the part that models the connector. The function **ProConnectorEntryPortsGet()** provides an array of the names of the coordinate system datums representing the entry ports in the specified connector. The connector is identified by its component path (its *memb_id_tab*).

The function **ProConnectorParamsCollect()** provides an array of the user parameters for the connector. However, this array contains only single-valued parameters that refer to the connector itself, not the parameters that describe the entry ports.

To access parameters on the connector entry ports, you must call the function **ProOutputFileWrite()** with the option **PRO_CONNECTOR_PARAMS_FILE**. This writes a text file to disk, which is the same format as the file you edit when using the ProCable command **Connector, Modify Parameters, Mod Param**.

The following example shows a sample connector parameters file. For an explanation of the parameters, see the *Pro/CABLING User's Guide*.

Connector Parameters File

```
! Enter or modify parameters for the connector. You may use the help
! functionality of Pro/TABLE to enter pre-defined parameters.
! Ref Descr
REF_DES MOTOR
! Conn Model
MODEL_NAME      MOTOR
! Num Of Pins
NUM_OF_PINS     2
! Type
TYPE            CONNECTOR
! Entry Port
!
!              TYPE      INT_LENGTH
ENTRY_PORT      ENTRY1   ROUND    0.2
ENTRY_PORT      ENTRY2   ROUND    0.2
! Signal
!              PIN_ID    SIGNAL_NAME      SIGNAL_VALUE      ENTRY_PORT
SIGNAL          1                ENTRY1
SIGNAL          2
! Pin
!              PIN_ID    CABLE_NAME      COND_ID
PIN_ASSIGN      2        WIRE_1
PIN_ASSIGN      1        WIRE_2
! Enter or modify parameters for the connector. You may use the help
! functionality of Pro/TABLE to enter pre-defined parameters.
! Ref Descr
REF_DES MOTOR
! Conn Model
MODEL_NAME      MOTOR
! Num Of Pins
NUM_OF_PINS     2
! Type
TYPE            CONNECTOR
! Entry Port
!
!              TYPE      INT_LENGTH
```



```

ENTRY_PORT      ENTRY1  ROUND  0.2
ENTRY_PORT      ENTRY2  ROUND  0.2
! Signal
!              PIN_ID  SIGNAL_NAME      SIGNAL_VALUE      ENTRY_PORT
SIGNAL           1
SIGNAL           2
! Pin
!              PIN_ID  CABLE_NAME      COND_ID
PIN_ASSIGN       2      WIRE_1
PIN_ASSIGN       1      WIRE_2

```

Note that this file is *not* free-format. Each parameter name and value is followed by a tab character, and each empty value is represented by a tab character. Therefore, the line in the example that assigns the first parameter, SIGNAL, contains three tab characters between the value of the PIN_ID and the value of ENTRY_PORT: the first tab belongs to the PIN_ID value, and next two tabs provide null values for SIGNAL_NAME and SIGNAL_VALUE.

The function **ProInputFileRead()** imports a file in this format, so you can use it in conjunction with **ProOutputFileWrite()** to edit the parameters on connectors and their entry ports. To identify the connector, both functions use the following arguments:

- *arg1*—Represents the *memb_id_tab*
- *arg2*—Represents the *memb_num*

The function **ProConnectorDesignate()** makes a part in the assembly take on the role of a cabling connector. It takes as input the component path that identifies the part in the cabling assembly, and an optional name that will be the reference descriptor (REF_DES) of the connector.

When a new connector has been designated, it has only the two parameters REF_DES and MODEL_NAME. The MODEL_NAME is set to be the name of the part designated, and the REF_DES is set to the value provided as input to the function **ProConnectorDesignate()**, if any, or to the MODEL_NAME otherwise. After you designate a connector, you must call **ProOutputFileWrite()** and **ProInputFileRead()** to set up the necessary parameters.

To undesignate a connector, call the function **ProConnectorUndesignate()**.

Function **ProConnectorParamGet()** retrieves a single parameter for the specified connector. This function supports only single-valued parameters. If you specify a multivalued parameter, the function returns PRO_TK_E_NOT_FOUND.

Use function **ProConnectorParamDelete()** to remove a single parameter from the specified connector. This function deletes both single- and multi-valued parameters.

Function **ProConnectorParamsCollect()** retrieves all parameters of the specified connector. This function supports both single- and multi-valued parameters.

Function **ProConnectorParamsSet()** sets all parameters of the specified connector. This function overwrites all existing parameter values with the values in the input parameter array. This function supports both single- and multi-valued parameters.

Function **ProConnectorsFromLogicalGet()** returns a list of connector names in the specified assembly for which data has been imported from a logical reference but which have not yet been created. Refer to the ProCable and Pro/HARNESS-MFG documentation for more information on logical references.

Managing Cables and Bundles

Functions Introduced:

- **ProCableCreate()**
- **ProBundleCreate()**
- **ProBundleCablesCollect()**

Use function **ProCableCreate()** to create a new cable or wire in a specified harness. The type of cable created corresponds to the spool type. This function creates all required parameters with default values. If the cable or wire name has already been imported from a wire list, then parameters from that reference are used instead of default values.

Use function **ProBundleCreate()** to create a new bundle in a specified harness. The type of bundle corresponds with the spool type. This function creates all required parameters with default values.

Cable Parameters

Functions Introduced:

- **ProCableParameterGet()**
- **ProCableParameterDelete()**
- **ProCableParametersCollect()**
- **ProCableParametersSet()**
- **ProCableparammemberFree()**
- **ProCableparamproarrayFree()**
- **ProCablesFromLogicalGet()**
- **ProCablesFromLogicalCreate()**
- **ProOutputFileWrite()**
- **ProInputFileRead()**

Function **ProCableParameterGet()** retrieves a single parameter for the specified cable. This function supports only single-valued parameters. If you specify a multivalued parameter, the function returns PRO_TK_E_NOT_FOUND.

Use **ProCableParameterDelete()** to remove a single parameter from the specified cable. This function deletes both single- and multi-valued parameters.

Function **ProCableParametersCollect()** retrieves all parameters of the specified cable. This function supports both single- and multi-valued parameters.

Function **ProCableParametersSet()** sets all parameters of the specified cable. This function overwrites all existing parameter values with the values in the input parameter array. This function supports both single- and multi-valued parameters.

Function **ProCableparammemberFree()** releases the memory assigned to the *ProCableparam* object. Function **ProCableparamproarrayFree()** releases the memory assigned to the array of *ProCableparam* objects. The *ProCableparam* object stores the names and values of cable parameters.

Function **ProCablesFromLogicalGet()** returns a list of cable names in the specified assembly for which data has been imported from a logical reference but which have not yet been created.

Use function **ProCablesFromLogicalCreate()** to create instances of cables for which logical data exists. Refer to the ProCable and Pro/HARNESS-MFG documentation for more information on logical references.

The functions **ProOutputFileWrite()** and **ProInputFileRead()** can be used with the option PRO_CABLE_PARAMS_FILE to export and import (and therefore edit) parameters on the specified cable.

The following example shows a sample cable parameters file. Like the file for connector parameters, the parameter names and values are separated by tab characters.

Cable Parameters File

```
! Enter or modify parameters for the cable.
! You can use the help functionality of Pro/TABLE
! to enter pre-defined parameters.
! Cable Name
NAME      WIRE_2
! Spool Name
SPOOL     24Y
! Modify the "DIRECTION" parameter for the end type of this cable.
!
      REF_DES  ENTRY_PORT  DIRECTION
END_TYPE   MOTOR   ENTRY1   FROM
END_TYPE   XCONN2  ENTRY1   TO
```

Cable Identifiers and Types

Functions Introduced:

- **ProCableByNameGet()**
- **ProCableNameGet()**
- **ProCableTypeGet()**

The functions **ProCableByNameGet()** and **ProCableNameGet()** provide the handle of a cable given its name. The functions also return the cable name when supplied with the handle.

The function **ProCableTypeGet()** provides the type of a named cable. The possible types are as follows:

- A wire—A single conductor
- A cable—With several conductors
- A bundle—A collection of other wires, cables, and bundles

Cable Cosmetic Features Create

Functions Introduced:

- **ProCableCosmeticFeatureCreate()**
- **ProCableTapeWindsGet()**
- **ProCableTapeWindsSet()**

The function **ProCableCosmeticFeatureCreate()** creates a cabling cosmetic feature like a marker, a tape or tie wrap. The selected cable location or cable segment point to use for the feature creation. If creating a tape feature, this must contain a cable location. If creating a marker, this must contain a point on the cable segment. If creating a tie wrap, this could be a cable location or a point on a cable segment.

The functions **ProCableTapeWindsGet()** and **ProCableTapeWindsSet()** provide access to the number of winds in a tape cosmetic feature.

Cable Connectivity

Function Introduced:

- **ProCableLogicalEndsGet()**

The function **ProCableLogicalEndsGet()** identifies the entry ports and their owning connectors to which the specified cable should be connected. This function depends on the connector parameters SIGNAL and PIN_ASSIGN. A cable connects logically to an entry port if the connector has a PIN_ASSIGN parameter that relates a PIN_ID value to that CABLE_NAME, and a SIGNAL parameter that relates the same PIN_ID value to that ENTRY_PORT name.

In the sample connector parameters file, the cable WIRE_2 connects to connector MOTOR, entry port ENTRY1.

A cable can have logical ends, even if it has not yet been routed.

The output of the function **ProCableLogicalEndsGet()** is in the form of two *Pro/Selection* structures for the coordinate system datums that represent the entry ports.

Cable Routing Locations

Functions Introduced:

- **ProCableLocationsCollect()**
- **ProHarnessLocationsCollect()**
- **ProCablelocationTypeGet()**
- **ProCablelocationPointGet()**
- **ProCablelocationCablesGet()**

The locations through which a cable is routed are identified by *ProCableLocation* structures, which are of the same structure as *ProModelItem*. The **ProCableLocationsCollect()** function provides an array of the structures for the locations through which the specified cable is routed. The function **ProHarnessLocationsCollect()** provides an array of the structures for the locations in a specified harness.

The function **ProCablelocationTypeGet()** gives the type of a specified location. The following table lists the valid location types.

Location Type	Equivalent Command in the CBL ROUTE Menu
PRO_LOC_CONNECTOR	Connector
PRO_LOC_POINT	Pnt/Vertex
PRO_LOC_FREE	Free
PRO_LOC_DEPENDENT	Dependent
PRO_LOC_AXIS	Along Axis
PRO_LOC_USE_DIR	Use Dir
PRO_LOC_OFFSET	Offset
PRO_LOC_SPLICE	Splice
PRO_LOC_LOC	Location
PRO_LOC_OFFSET_CSYS	Coordinate Offset
PRO_LOC_OFFSET_AXIS	Axis Offset

The function **ProCablelocationPointGet()** provides the XYZ coordinates of the location in the coordinate system of the harness.

The function **ProCablelocationCablesGet()** provides an array of the names of cables routed through the specified location.

Cable Geometry

Functions Introduced:

- **ProCableLengthGet()**
- **ProCableSegmentsGet()**
- **ProCablesegmentPointsGet()**
- **ProCablesegmentIsInBundle()**
- **ProCablesegmentIsNew()**

The function **ProCableLengthGet()** provides the length of the specified wire within a specified harness.

The functions **ProCableSegmentsGet()** and **ProCablesegmentPointsGet()** provide the geometry of a named wire, bundle, or cable within a specified harness part. The geometry of a wire or cable is divided into a number of segments, each of which represents a region where the wire or cable is bundled with other wires and cables. The geometry of each such segment is described by a series of three-dimensional locations and tangent vectors.

The function **ProCablesegmentIsInBundle()** determines whether a cable segment runs into a bundle.

The function **ProCablesegmentIsNew()** determines whether the cable segment is connected to a previous cable segment.

Measuring Harness Clearance

Function Introduced:

- **ProCableClearanceCompute()**

The function **ProCableClearanceCompute()** determines the minimum distance between two items in a harness. The items can be of any of the following types, in any combination:

- Part
- Surface
- Cable
- Cable location

The inputs identify the two items in terms of *ProSelection* structures. The function outputs a flag to show whether the two items interfere, and, if they do not interfere, the function also returns the three-dimensional locations of the two nearest points.

Cable Routing

Functions Introduced:

- **ProCableRoutingStart()**
- **ProCableThruLocationRoute()**
- **ProCableRoutingEnd()**
- **ProCablelocationrefAlloc()**
- **ProCablelocationrefFree()**

To Route a Group of Cables Through a Sequence of Locations:

1. Call **ProCableRoutingStart()** to identify the cables to be routed.
2. Call **ProCablelocationrefAlloc()** to create a routing reference location structure.
3. Call **ProCableThruLocationRoute()** for each location through which to route the cables.
4. Call **ProCablelocationrefFree()** to free the location reference.
5. Call **ProCableRoutingEnd()** to complete the routing.
6. Call **ProSolidRegenerate()** to make Pro/ENGINEER calculate the resulting cable geometry and create the necessary cable features.

Note: You must also call the function **ProWindowRepaint()** to see the new cables.

After the call to **ProCableRoutingStart()**, the information about the routing in progress is contained in an opaque data structure *ProRouting* that **ProCableRoutingStart()** provides. This pointer is then given as an input to the functions **ProCableThruLocationRoute()** and **ProCableRoutingEnd()**.

The inputs to **ProCableRoutingStart()** are the cabling assembly and harness handles, and an array of cables.

The input to **ProCableThruLocationRoute()** is a structure of type *ProCablelocationref*, which contains all the necessary information about the location through which to route the cables.

The following table shows the possible values of the *type* field, and the values that other fields need for each type.

type	refs	axis_flip	offsets
PROLOC_CONNECTOR	The coordinate system datum for the entry port	—	—
PROLOC_POINT	The datum point	—	—
PROLOC_AXIS	The axis	0 or 1 to show the routing direction	—
PROLOC_OFFSET	The coordinate system datum to define the offset directions	—	Offset distances from the previous location
PROLOC_LOC	An existing routing location	—	—

The function **ProCableThruLocationRoute()** also outputs an array of the structures for the locations created as a result of the call. (The function usually creates a single location, but creates two in the case of routing through an axis.)

As input, the **ProCableRoutingEnd()** function takes only the *void** for the routing data.

Deleting Cable Sections

Function Introduced:

- **ProCableSectionsDelete()**

The function **ProCableSectionsDelete()** deletes the section of cables that lies between designated locations.

ProCableSectionsDelete() does not delete loom bundle cable sections.

Production Applications: Piping

This chapter contains information about the Pro/PIPING API functions. The functions in this section allow a Pro/TOOLKIT application to create, read, and write pipe linestock information. These APIs also support analysis of pipeline connectivity as built with the Pro/ENGINEER module Pro/PIPING.

Topic	Page
Piping Terminology	59 - 2
Linestock Management Functions	59 - 3
Pipeline Connectivity Analysis	59 - 5

Piping Terminology

Pro/TOOLKIT supports Pro/Piping. Pro/Piping uses specific terminology. This section defines this terminology.

A **pipeline** is set of interconnecting pipes and fitments. A pipeline consists of an extension which terminates at open ends (that is, ends with no further pipeline items are attached), non-open ends (that is, ends with equipment such as nozzles or other pipelines), or junctions. A pipeline also contains extensions that branch from extensions which branch from it, then others which branch from those, and so on.

A pipeline extension is a non-branching sequence of pipeline items.

A **pipeline feature** is a feature which names the pipeline to show its grouping. All other features in the pipeline refer to this feature. A pipeline feature does not contain any geometry of its own.

At a **branch** (or **junction**), pipes are grouped into extensions such that the extension which continues across the branch has a continuous direction of flow, and, if that criterion leaves a choice, has the smallest change of direction possible for that branch. Other pipes which join that branch then form the end points of other extensions.

A **member** of an extension is a terminator, a series, or a junction.

A **terminator** is the open or non-open ends of the pipeline.

A **series** is a non-branching sequence of pipeline objects.

A **junction** is an assembly component or a datum point which represents a part which joins three or more pipe segments.

A **stubin** is a datum point which joints three or more series.

A **segment** is a section of pipe, either straight or arced. If arced, the segment is manufactured by taking a straight section of tube and bending it.

A **fitting** is a component that connects two pipe segments, for example, to form a corner where space does not allow a bent pipe segment, or to represent an item such as a valve.

A **pipeline object** is a segment, a fitting, or a stubin.

A **pipeline network** is a data structure which contains references to pipeline objects. The objects are structured to show their connectivity and sequence in relation to the flow.

Linestock Management Functions

This section presents functions for management of linestock.

Linestocks

Functions introduced:

- **ProAssemblyLnstksCollect()**
- **ProPipelineLnstkGet()**
- **ProPipelineLnstkSet()**
- **ProLnstkCreate()**

A linestock is represented by the object ProLnstk, which is a D Handle, with this declaration:

```
typedef struct pro_lnstk
{
    ProName          name;
    ProAssembly      owner;
} ProLnstk;
```

The function **ProAssemblyLnstksCollect()** finds all the linestocks defined for a specified assembly.

The functions **ProPipelineLnstkGet()** and **ProPipelineLnstkSet()** get and set the default linestock for a specified pipeline feature.

The function **ProLnstkCreate()** creates a new linestock in the specified assembly.

Linestock Parameters

Functions introduced:

- **ProLnstkParametersCollect()**
- **ProLnstkParametersSet()**
- **ProLnstkParameterAdd()**
- **ProLnstkParameterDelete()**

The parameters of a linestock differ from regular Pro/ENGINEER parameters in that they may be organized hierarchically. The data structure ProLnstkParam contains the description of a linestock parameter and its member parameters, if any. Its declaration follows, along with those of its member types.

```

typedef enum
{
    PROLNSTKPRM_SINGLE,
    PROLNSTKPRM_MULTIPLE
} ProLnstkParamType;

typedef struct _pro_lnstk_param_memb_
{
    ProName          name;
    ProParamvalue    value;
} ProLnstkParamMemb;

typedef struct _pro_lnstk_param_
{
    ProName          name;
    ProLnstkParamType param_type;
    union {
        ProParamvalue    value;
        ProLnstkParamMemb *members;
    } lnstk_param_value;
} ProLnstkParam;

```

The function **ProLnstkParametersCollect()** finds all the parameters for a specified linestock.

The function **ProLnstkParametersSet()** sets the parameters on a specified linestock to a specific list.

The function **ProLnstkParameterAdd()** adds a new parameter to the list of parameters on a specified linestock.

The function **ProLnstkParameterDelete()** deletes a named parameter from a linestock.

The enumerated types used to represent values of some of the parameters are as follows:

- ProLnstkPipeSection
- ProLnstkPipeXSection
- ProLnstkPipeShape
- ProLnstkPipeCrnrType

Pipeline Connectivity Analysis

The functions in the section support analysis of pipeline connectivity.

Networks

Functions introduced:

- **ProPipelineNetworkEval()**
- **ProPnetworkFree()**
- **ProPnetworkLabelGet()**
- **ProPnetworkSizeGet()**
- **ProPnetworkSpecGet()**

A pipeline is a collection of Pro/ENGINEER piping features and components that are connected together. A pipeline feature is a single feature that unites all the features and components in a pipeline. All the features and components that belong to one pipeline reference the pipeline feature.

A network is a temporary data structure which is the result of analyzing the connectivity and topology of the features and components in a pipeline. The functions in this section allow a Pro/TOOLKIT function to create and analyze the network for a pipeline, which would be the first step in, for example, an analysis of the fluid flow down the pipeline.

The network is a hierarchical data structure whose branches describe the various logical subdivisions into which the features and components of a pipeline divide themselves according to their connectivity.

A network is described by the opaque pointer ProPnetwork. The function **ProPipelineNetworkEval()** analyzes the features and components that belong to a pipeline (specified by its pipeline feature) and builds a network data structure.

After the structure has been analyzed it should be freed using **ProPnetworkFree()**.

The functions **ProPnetworkLabelGet()**, **ProPnetworkSizeGet()**, and **ProPnetworkSpecGet()** get information about the pipeline described by a specified network.

Extensions

Functions introduced

- **ProPnetworkExtensionVisit()**
- **ProPextensionFlowGet()**

A network contains a list of extensions. An extension is a non branching sequence of connected pipeline items. At a branch in a pipeline one extension is continuous across the branch and other extensions terminate at the branch. To decide which extension is continuous across the branch, the analysis performed by **ProPipelineNetworkEval()** uses the following rules:

- The extension must have a continuous direction of flow across the branch.
- Of all such possible extensions, the one chosen is the one that gives the smallest change of direction across the branch.

An extension is represented by the opaque pointer **ProPextension**. The function **ProPnetworkExtensionVisit()** visits all the extensions in a network.

The function **ProPextensionFlowGet()** tells you the flow direction in relation to the sequence of members in the extension.

Members

Functions introduced:

- **ProPextensionMemberVisit()**
- **ProPmemberTypeGet()**

An extension is conceptually divided into objects called members, described by the opaque object **ProPmember**. The members in an extension divide it at the pipeline branches which the extension crosses.

There are three types of member:

- Terminator—The end of a pipeline, where it either opens or connects to an item outside the pipeline, described by the opaque object **ProPterminator**.
- Junction—The item that describes how the pipeline branches, described by the opaque object **ProPjunction**.
- Series—A non branching sequence of pipeline objects, described by the opaque object **ProPseries**.

The function **ProPextensionMemberVisit()** visits all the members in an extension, and the function **ProPmemberTypeGet()** reports which of the three types the member represents. Each of three types of member is in turn composed of one or more objects.

The following sections describe the analysis of the three types of members.

Terminators

Functions introduced:

- **ProPmemberTerminatorGet()**
- **ProPterminatorTypeGet()**

The function **ProPmemberTerminatorGet()** outputs the Pterminator object, which represents the terminator in the specified member.

The function **ProPterminatorTypeGet()** tells you whether a terminator is an input or an output.

Junctions

Functions introduced:

- **ProPmemberJunctionGet()**

The function **ProPmemberJunctionGet()** outputs the Pjunction object which represents the junction in the specified member.

Series

Functions introduced:

- **ProPmemberSeriesGet()**
- **ProPseriesIdGet()**

The function **ProPmemberSeriesGet()** outputs the Pseries object which represents the series in the specified member.

The function **ProPseriesIdGet()** yields the integer id of the specified series.

Objects

Functions introduced:

- **ProPterminatorObjectGet()**
- **ProPjunctionObjectGet()**
- **ProPseriesObjectVisit()**
- **ProPobjectTypeGet()**
- **ProSelectionPipelineGet()**
- **ProPobjectSegmentGet()**
- **ProPobjectFittingGet()**
- **ProPfittingAsmcompGet()**
- **ProPobjectStubinGet()**
- **ProPstubinPointGet()**
- **ProPobjectSelectionGet()**
- **ProPselectionSelectionGet()**

A Piping Object describes a single item in a pipeline and is represented by the opaque pointer Pobject.

The functions **ProPterminatorObjectGet()** and **ProPjunctionObjectGet()** output the single object used to represent a terminator or a junction. The function **ProPseriesObjectVisit()** visits all the objects that represent the contents of a series.

The function **ProPobjectTypeGet()** yields one of the following types:

- **Segment**—A single pipe segment, either bent or straight. Can only belong to a series.
- **Fitting**—An assembly component that connects two or more pipeline segments. Can belong to a Series (if it connects two segments) or a Junction (if it connects more than two segments).
- **Stubin**—A datum point that defines the location where two or more pipeline segments connect directly without a fitting. Can only belong to a Junction.
- **Selection**—An object that contains a ProSelection describing the item a pipeline terminator connects to. Can only belong to a Terminator.

The function **ProSelectionPipelineGet()** outputs the pipeline feature to which the specified pipeline selection belongs.

The function **ProPobjectSegmentGet()** outputs the Segment contained by an Object of the appropriate type. The Segment is described in the next section.

The function **ProPobjectFittingGet()** outputs the fitting contained by an object of the appropriate type. The fitting is represented by the opaque object ProPf fitting. The assembly component that represents the fitting can be found using the function **ProPf fittingAsmcompGet()**.

The function **ProPobjectStubinGet()** outputs the stubin contained by an object of the appropriate type. The stubin is represented by the opaque pointer ProPstubin. The function **ProPstubinPointGet()** yields the 3-D location of the stubin.

The function **ProPobjectSelectionGet()** outputs the selection contained by an object of type terminator. The function **ProPselectionSelectionGet()** provides the ProSelection object that the selection contains and identifies the item outside the pipeline to which the terminator connects.

Segments

Functions introduced:

- **ProPsegmentTypeGet()**
- **ProPsegmentLinestockGet()**
- **ProPsegmentLengthGet()**
- **ProPsegmentCurvesGet()**

A segment is represented by the opaque pointer ProPsegment.

The function **ProPsegmentTypeGet()** tells you whether the segment is straight or bent.

The function **ProPsegmentLinestockGet()** outputs which linestock was used for this segment. Note that because the pipeline may contain fittings which cause a change in diameter, some segments in the pipeline may yield a different linestock from that provided by **ProPipelineLnstkgGet()** for the pipeline itself.

The function **ProPsegmentLengthGet()** outputs the physical length of the segment.

The function **ProPsegmentCurvesGet()** outputs an array of ProCurve objects that describes the geometry of the centerline of the segment. The curves are always listed in the direction of flow.

Production Applications: Welding

Welding is an optional Pro/ENGINEER module that allows you to model welds in assemblies. In addition, you can generate report tables about weld parameters and show welding symbols in assembly drawings. This chapter provides a brief overview of weld features. For more information, refer to the “Welding Design” module in the Pro/ENGINEER Online Help.

Topic	Page
Read Access to Weld Features	60 - 2
Customizing Weld Drawing Symbols	60 - 3

Read Access to Weld Features

The functions listed in this section provide access to basic information about existing weld features.

Functions introduced:

- **ProWeldTypeGet()**
- **ProWeldInfoGet()**
- **ProWeldIntermittenceGet()**
- **ProWeldSequenceIdGet()**
- **ProWeldRodGet()**
- **ProWeldRodNameGet()**
- **ProWeldCompoundGet()**
- **ProWeldFilletdataGet()**
- **ProWeldGroovedataGet()**
- **ProWeldPlugdataGet()**
- **ProWeldSlotdataGet()**
- **ProWeldSpotdataGet()**

Use function **ProWeldTypeGet()** to output the type and subtype of the specified weld. Function **ProWeldInfoGet()** outputs the information you get by using the Pro/ENGINEER command **Info Weld**.

Function **ProWeldIntermittenceGet()** outputs information about an intermittent weld, describing the size, number, and location of the welds that form it.

Use function **ProWeldSequenceIdGet()** to obtain the sequence ID of a weld feature.

Use function **ProWeldRodGet()** to provide the feature handle of the rod for the specified weld feature. Function **ProWeldRodNameGet()** gets the name of the specified weld rod feature.

ProWeldCompoundGet() outputs the list of welds in a compound weld.

Use functions **ProWeldFilletdataGet()**, **ProWeldGroovedataGet()**, **ProWeldPlugdataGet()**, **ProWeldSlotdataGet()**, and **ProWeldSpotdataGet()** to output data on a specific fillet, groove, plug, slot, or spot weld respectively.

Customizing Weld Drawing Symbols

Functions introduced:

- **ProDrawingWeldSympathGetAction()**
- **ProDrawingWeldGroupsGetAction()**
- **ProDrawingWeldSymtextGetAction()**

This section describes three notification functions invoked by Pro/ENGINEER when the user instantiates a weld symbol that documents a weld in drawing mode. Your callback functions can output information which is used to modify the weld symbol that appears on the drawing. The effect is to allow much greater customization of the appearance of the weld symbol than is possible without Pro/TOOLKIT.

Note: From Pro/ENGINEER Wildfire 5.0 onward, you can also create weld symbols in weld features as 3D Symbol Annotation Elements. Pro/TOOLKIT allows you to access the Weld Symbol Annotation Elements using existing **ProAnnotation***() functions. For more information on the functions, refer the “Annotations: Annotation Features and Annotations” chapter.

Each callback has input arguments which identify the drawing, the weld assembly, the weld feature being annotated, and the path to the drawing symbol being used. The functions for read-access to welds, described in the previous section, would be used inside the callbacks to find out about the weld being annotated.

Refer to the ‘Event-driven Programming: Notifications’ chapter for more data on how to set a notification.

Weld symbol notification types are:

- **PRO_DRAWING_WELD_SYMPATH_GET** -- allows the callback function to override the entire weld symbol by specifying the path and file name of a substitute symbol.
- **PRO_DRAWING_WELD_GROUPIDS_GET** -- allows the callback to selectively include or exclude symbol groups contained in the symbol. Additional inputs to the callback are a flag to show which way the symbol will point (left or right) and an array of the names of the groups in the symbol; the output is an array of booleans which select the groups to be included.
- **PRO_DRAWING_WELD_SYMTEXT_GET** -- allows the callback to substitute for variable text in the symbol.

All three notifications can be set at the same time, allowing you to use your own set of generic symbols which are designed to be customized according to the weld type and properties.

Example 1: Weld Callback Notification

The following example code shows how to use weld callback notification functions.

```
#include "ProToolkit.h"
#include "ProParameter.h"
#include "ProUtil.h"
#include "ProNotify.h"
#include "ProParamval.h"
#include <PTAppIsUnicodeUtils.h>
#include <ProWeld.h>

#include "TestError.h"
#include "UtilString.h"

#include <prodevelop.h>

int UsrWeldNotificationSet()
{
    int status;
    ProError ptk_get_text_func();
    ProError ptk_get_path_func();

    status = ProNotificationSet( PRO_DRAWING_WELD_SYMPATH_GET,
                                ptk_get_path_func );

    status = ProNotificationSet( PRO_DRAWING_WELD_SYMTEXT_GET,
                                ptk_get_text_func );

    return(0);
}
/*=====*\
FUNCTION: ptk_get_path_func
PURPOSE: Set path of a user defined symbol for features containing
parameter named "SYMBOL"
\*=====*/
ProError ptk_get_path_func( ProDrawing p_draw, ProMdl p_model,
                           int feat_id, ProPath sym_def_file_path,
                           ProPath ptk_sym_def_path)
{
    ProError status;
    ProParameter param;
    ProName name;
    ProParamvalue value;
    char filename[PRO_FILE_NAME_SIZE];
```



```

    char path[PRO_PATH_SIZE];
    ProFeature afeat;
/*-----*\
Check if feature being passed in contains a parameter named "SYMBOL"
\*-----*/
    ProStringToWstring(name, "SYMBOL");
    status=ProFeatureInit((ProSolid)p_model, feat_id, &afeat);
    status = ProParameterInit((ProModelitem*)&afeat, name, &param);
/*-----*\
Build new path using value of parameter string and return PRO_TK_NO_ERROR
to flag Pro/E to use this path instead of input path (sym_def_file_path)
\*-----*/
    if(status == PRO_TK_NO_ERROR)
    {
        status = ProParameterValueGet(&param, &value);
        ProWstringToString(filename, value.value.s_val);
        strcpy(path, "/whole_path/user_symlib/");
        strcat(path, filename);
        ProStringToWstring( ptk_sym_def_path, path);
        return PRO_TK_NO_ERROR;
    }

    return PRO_TK_E_NOT_FOUND;
}
/*=====*\
FUNCTION: ptk_get_text_func
PURPOSE: Modify symbol instance text based on sym_prompt_name values
\*=====*/
ProError ptk_get_text_func(ProDrawing p_draw, ProMdl p_model, int feat_id,
int sym_def_id, wchar_t sym_prompt_name[],
int sym_prompt_nm_idx, ProParamvalue* p_text,
ProParamvalue* p_ptk_text )
{
    char tmp[PRO_LINE_SIZE], s_id[PRO_LINE_SIZE];
    char name[PRO_LINE_SIZE], text_val[PRO_LINE_SIZE];
    wchar_t ws_id[PRO_LINE_SIZE], text[PRO_LINE_SIZE];
    ProFeature weld_fea;
    int sequence_id;
    ProError status;
    ProParameter param;
    ProParamvalue value;

    ProWstringToString(name, sym_prompt_name);
/*-----*\
Set sequence number
\*-----*/
    if(strcmp(name, "seq_var")==0)
    {
        ProFeatureInit((ProSolid)p_model, feat_id, &weld_fea);
        ProWeldSequenceIdGet(&weld_fea, &sequence_id);
        ProTKSprintf(s_id, "%d", sequence_id);

```

```

        ProStringToWstring(ws_id, s_id);
        status = ProParamvalueSet(p_ptk_text,ws_id,PRO_PARAM_STRING);
        return PRO_TK_NO_ERROR;
    }
/*-----*\
Set sample text string from parameter value
\*-----*/
    if(strcmp(name,"text")==0)
    {
        ProFeatureInit((ProSolid)p_model,feat_id,&weld_fea);
        ProStringToWstring(text,"TESTING");
        status = ProParameterInit((ProModelitem*)&weld_fea,text,&param);
        if(status == PRO_TK_NO_ERROR)
        {
            status = ProParameterValueGet(&param,&value);
            ProWstringToString(text_val,value.value.s_val);
            status = ProParamvalueSet(p_ptk_text,value.value.s_val,
                                     PRO_PARAM_STRING);
            return PRO_TK_NO_ERROR;
        }
    }

    return PRO_TK_E_NOT_FOUND;
}

```

Mechanica: Mechanica Items

This chapter describes how to access the properties of Mechanica items. The functions described in this chapter evaluate the model's structural characteristics and thermal profile and provide powerful tools for examining mechanism performance.

Topic	Page
Entering the Pro/ENGINEER Mechanica Environment	61 - 3
Selection of Mechanica Items	61 - 4
Accessing Mechanica Items	61 - 6
Mechanica Object References	61 - 7
Geometric References	61 - 8
Y-directions	61 - 11
Functions	61 - 12
Mechanica Expressions	61 - 15
Accessing the Properties used for Loads and Constraints	61 - 15
Mechanica Loads	61 - 20
Mechanica Load Sets	61 - 39
Mechanica Constraints	61 - 40
Mechanica Constraint Sets	61 - 56
Mechanica Beams	61 - 56
Mechanica Beam Sections	61 - 60
Sketched Beam Section	61 - 65
General Beam Section	61 - 66

Beam Orientations	61 - 68
Beam Releases	61 - 70
Mechanica Spring Items	61 - 71
Mechanica Spring Property Items	61 - 73
Mechanica Mass Items	61 - 74
Mechanica Mass Properties	61 - 77
Material Orientations	61 - 78
Mechanica Shells	61 - 82
Shell Properties	61 - 88
Shell Pairs	61 - 96
Interfaces	61 - 99
Gaps	61 - 106
Mesh Control	61 - 108
Welds	61 - 121

Entering the Pro/ENGINEER Mechanics Environment

You can access the Pro/TOOLKIT Mechanics functions in one of the following situations:

- When the Pro/ENGINEER session is in the Pro/ENGINEER Mechanics user interface for a given model.
- When the application initializes the Pro/ENGINEER Mechanics environment for a given model.

Functions Introduced:

- **ProMechanicaEnter()**
- **ProMechanicaLeave()**
- **ProMechanicsIsActive()**

The function **ProMechanicaEnter()** allows you to enter the Pro/ENGINEER Mechanics environment to access information about the Mechanics items in a specified model. The model must be displayed in the window.

Note: Models created in Mechanics Lite mode, in both Structure and Thermal, are not supported by the Pro/TOOLKIT functions. If you access a Mechanics Lite model in the Pro/ENGINEER Mechanics environment, the function **ProMechanicaEnter()** returns an error PRO_TK_CANT_ACCESS.

The function **ProMechanicaLeave()** exits the Pro/ENGINEER Mechanics environment entered using the previous function.

The function **ProMechanicsIsActive()** identifies whether the Pro/ENGINEER Mechanics environment is currently active. The environment might be active if the user has entered the Pro/ENGINEER Mechanics environment interactively, or if the function **ProMechanicaEnter()** has been called.

Note: Only functions related to the Mechanics database must be called between the calls to the functions **ProMechanicaEnter()** and **ProMechanicaLeave()**. **ProMechanicaEnter()** must not be used to initialize the Pro/ENGINEER Mechanics User Interface.

Selection of Mechanics Items

Pro/TOOLKIT supports selection of certain Mechanics items. Refer to the chapter "User Interface: Selection" for details about selection in Pro/ENGINEER. These items are selected using **ProSelect()** or are obtained from the selection buffer while Pro/ENGINEER is in the Pro/ENGINEER Mechanics Environment. The following table lists the selectable items and their **ProSelect()** filter strings:

Item Type	ProSelect() Filter String	Model Item Type
Load	sim_load*	PRO_SIMULATION_LOAD
Constraint	sim_load*	PRO_SIMULATION_CONSTRAINT
Beam	sim_beam	PRO_SIMULATION_BEAM
Spring	sim_spring	PRO_SIMULATION_SPRING
Gap	sim_gap	PRO_SIMULATION_GAP
Mass	sim_mass	PRO_SIMULATION_MASS
Shell	sim_shell	PRO_SIMULATION_SHELL
Shell pair	sim_shlpair	PRO_SIMULATION_SHELL_PAIR
Weld	sim_weld	PRO_SIMULATION_WELD
Interface	sim_connect	PRO_SIMULATION_INTERFACE

*Interactive selection using this filter will by default allow the user to select both loads and constraints. Use the **ProSelect()** filters if you desire to only allow selection of one of these types.

Example 1: Interactively Selecting and Deleting a Mechanics Item

This example shows how to select a Mechanics item interactively and delete it.

```
#include <ProMechItemSet.h>
#include <ProUtil.h>
#include <ProMessage.h>
#include <PTMechExamples.h>

/*=====*\
  FUNCTION: PTMechExDelete
  PURPOSE:  Deletes the mech item
\*=====*/

ProError PTMechExDelete ()
{
```

```

ProSelection* sels;
ProMechItem mech_item;
int n_sels, i;
ProMdl constraint_part;

status = ProMdlCurrentGet(&constraint_part);
PT_TEST_LOG_SUCC ("ProMdlCurrentGet");

/*-----*\
  Select the item to delete
\*-----*/

status = ProMessageDisplay(msgfile, "PTMechEx Select the item to
                                  delete");
PT_TEST_LOG_SUCC ("ProMessageDisplay()");

status =ProSelect("sim_beam,sim_spring,sim_gap,sim_mass,
                  sim_shell,sim_mc,sim_shlpair,sim_weld,
                  sim_connect",
                  PRO_VALUE_UNUSED,
                  NULL,
                  NULL,
                  NULL,
                  NULL,
                  &sels,
                  &n_sels);
PT_TEST_LOG_SUCC ("ProSelect()");

for (i = 0; i < n_sels; i++)
{
    status = ProSelectionModelitemGet (sels[i],
(ProModelitem*)&mech_item);
    PT_TEST_LOG_SUCC ("ProSelectionModelitemGet()");

/*-----*\
  Remove the mech item
\*-----*/

    status = ProMechitemRemove (&mech_item);
    PT_TEST_LOG_SUCC ("ProMechitemRemove");

}

/*-----*\
  Print Message from the result
\*-----*/

if (status == PRO_TK_NO_ERROR)
{
    status = ProMessageDisplay(msgfile, "PTMechEx Item deleted
successfully");
    PT_TEST_LOG_SUCC ("ProMessageDisplay()");
}

```

```

    }

else
{
    status = ProMessageDisplay(msgfile,
                              "PTMechEx Item deletion failed");
    PT_TEST_LOG_SUCC("ProMessageDisplay()");
}

return PRO_TK_NO_ERROR;
}

```

Accessing Mechanics Items

The functions described in this section allow access to Mechanics items in Pro/ENGINEER.

Functions Introduced:

- **ProSolidMechitemVisit()**
- **ProMechitemNameGet()**
- **ProMechitemStatusGet()**
- **ProMechitemFilterAction()**
- **ProMechitemVisitAction()**

The Mechanics item is a derivative of the structure `ProModelitem` and is defined as

```

typedef struct pro_model_item
{
    ProType type;

    int id;

    ProMdl owner;
}ProMechItem;

```

The function **ProSolidMechitemVisit()** traverses the Mechanics items in a specified model. This function allows you to specify the type of item to be located, or optionally all item types can be visited.

The function **ProMechitemNameGet()** returns the name of the Mechanics item.

The function **ProMechitemStatusGet()** returns the visibility status of the item in the current Pro/ENGINEER Mechanica environment.

The function **ProMechitemFilterAction()** is used to filter Mechanica items for visiting them.

The function **ProMechitemVisitAction()** is used to visit a Mechanica item.

Mechanica Object References

References to specific Mechanica items are contained within the Mechanica object references structure. This structure is represented in Pro/TOOLKIT by the opaque handle, *ProMechObjectref*.

Functions Introduced:

- **ProMechobjectrefAlloc()**
- **ProMechobjectrefTypeGet()**
- **ProMechobjectrefTypeSet()**
- **ProMechobjectrefIdGet()**
- **ProMechobjectrefIdSet()**
- **ProMechobjectrefPathGet()**
- **ProMechobjectrefPathSet()**
- **ProMechobjectrefFree()**
- **ProMechobjectrefProarrayFree()**

The function **ProMechobjectrefAlloc()** allocates memory for the Mechanica object references handle.

The function **ProMechobjectrefTypeGet()** returns the type of the specified Mechanica object. The output argument *type* is one of the PRO_SIMULATION* types.

The function **ProMechobjectrefTypeSet()** sets the type of the Mechanica object.

The function **ProMechobjectrefIdGet()** obtains the ID of the specified Mechanica object.

The function **ProMechobjectrefIdSet()** sets the ID of the specified Mechanica object.

The function **ProMechobjectrefPathGet()** returns the complete path of the Mechanics object reference from the root assembly to the part or assembly that owns the specified Mechanics object.

The function **ProMechobjectrefPathSet()** sets the complete path for the Mechanics object reference.

Use the function **ProMechobjectrefFree()** to release the memory assigned to the Mechanics object reference handle.

Use the function **ProMechobjectrefProarrayFree()** to release the memory assigned to a *ProArray* of Mechanics object reference handles.

Geometric References

Mechanics items use a geometric reference structure to contain references to Pro/ENGINEER geometry items. This structure is represented in Pro/TOOLKIT by the opaque handle *ProMechGeomref*.

Functions Introduced:

- **ProMechgeomrefAlloc()**
- **ProMechgeomrefTypeGet()**
- **ProMechgeomrefSubtypeGet()**
- **ProMechgeomrefIdGet()**
- **ProMechgeomrefPathGet()**
- **ProMechgeomrefFree()**
- **ProMechgeomrefProarrayFree()**
- **ProMechgeomrefTypeSet()**
- **ProMechgeomrefSubtypeSet()**
- **ProMechgeomrefIdSet()**
- **ProMechgeomrefPathSet()**

The function **ProMechgeomrefAlloc()** allocates memory for the geometric entity. The function returns a handle to the geometric entity.

The function **ProMechgeomrefTypeGet()** returns the *type* for the specified geometric entity. The output argument type can have one of the following values:

- PRO_MECH_POINT—Specifies a point.
- PRO_MECH_EDGE—Specifies an edge.
- PRO_MECH_SURFACE—Specifies a surface.
- PRO_MECH_VERTEX—Specifies a vertex.
- PRO_MECH_QUILT—Specifies a quilt.
- PRO_MECH_BOUNDARY—Specifies a boundary. This type is valid for all surfaces.
- PRO_MECH_CURVE—Specifies a curve.
- PRO_MECH_MODEL—Specifies a model. This type is valid for all parts and assemblies.
- PRO_MECH_AXIS—Specifies the axis.
- PRO_MECH_COORD_SYSTEM—Specifies the coordinate system.
- PRO_MECH_LAYER—Specifies a layer.
- PRO_MECH_VOLUME—Specifies a set of associated surfaces that visually represents an entity with volume.
- PRO_MECH_INT*—Specifies the datum reference features that store the design intent objects. Intent objects are families of associated points, curves, edges, or surfaces that logically define boundaries of geometry created or modified by a feature. The types of datum reference features available are:
 - PRO_MECH_INT_PNT—Specifies intent datum point references.
 - PRO_MECH_INT_CURVE—Specifies intent curve references.
 - PRO_MECH_INT_EDGE—Specifies intent edge references.
 - PRO_MECH_INT_SURFACE—Specifies intent surface references.
- PRO_MECH_FEAT—Specifies the references to a Weld Feature. The weld feature should be of type:
 - Groove or Fillet
 - A surface weld

The function **ProMechgeomrefTypeSet()** sets the type of the geometric entity.

The function **ProMechgeomrefSubtypeGet()** returns the subtypes of the specified geometric entity. Only certain types of geometric entities require subtypes. The geometric entity types and their respective subtypes are as follows:

- PRO_MECH_POINT
 - PRO_MECH_POINT_SINGLE—Specifies the placement of a point at any location.
 - PRO_MECH_POINT_FEATURE—Specifies the placement of a point along a surface.
 - PRO_MECH_POINT_PATTERN—Specifies the placement of a point along a curve.
- PRO_MECH_VERTEX
 - PRO_MECH_VERTEX_EDGE_START—Specifies the start point of the referenced edge.
 - PRO_MECH_VERTEX_EDGE_END—Specifies the end point of the referenced edge.
- PRO_MECH_COORD_SYSTEM
 - PRO_MECH_CSYS_CARTESIAN—Specifies a Cartesian coordinate system.
 - PRO_MECH_CSYS_CYLINDRICAL—Specifies a cylindrical coordinate system.
 - PRO_MECH_CSYS_SPHERICAL—Specifies a spherical coordinate system.
- PRO_MECH_SURFACE
 - PRO_MECH_SURFACE_NORMAL—Specifies that the surface reference uses the standard normal direction.
 - PRO_MECH_SURFACE_REVERSED—Specifies that the surface reference uses the standard normal direction.
- PRO_MECH_CURVE
 - PRO_MECH_CURVE_NORMAL—Specifies that the curve proceeds in the default direction (from t=0 to t=1).
 - PRO_MECH_CURVE_REVERSED—Specifies that the curve reference uses the reverse direction of the curve.
- PRO_MECH_EDGE
 - PRO_MECH_EDGE_SURF_0
 - PRO_MECH_EDGE_SURF_1

The function **ProMechgeomrefSubtypeSet()** sets the subtypes for the specified geometric entity.

The function **ProMechgeomrefIdGet()** returns the ID of the specified entity.

The function **ProMechgeomrefIdSet()** sets the ID of the specified entity.

The function **ProMechgeomrefPathGet()** returns the complete path of the assembly-component references from the root assembly to the part or assembly that owns the specified geometric reference entity.

The function **ProMechgeomrefPathSet()** sets the complete path of the assembly-component references.

Use the function **ProMechgeomrefFree()** to free the geometric reference entity from the memory.

Use the function **ProMechgeomrefProarrayFree()** to free the array of geometric entities from the memory.

Y-directions

Several types of Mechanica items require a Y-direction (indicating a direction governing the properties of the item). In Pro/TOOLKIT, Y-directions are represented using the opaque handle ProMechYDirection. The functions described in this section provide access to the y-direction handle.

Functions Introduced:

- **ProMechydirectionAlloc()**
- **ProMechydirectionTypeGet()**
- **ProMechydirectionCsysGet()**
- **ProMechydirectionCsysSet()**
- **ProMechydirectionReferenceGet()**
- **ProMechydirectionReferenceSet()**
- **ProMechydirectionVectorGet()**
- **ProMechydirectionVectorSet()**
- **ProMechydirectionFree()**

The function **ProMechydirectionAlloc()** allocates memory for the y-direction handle.

The function **ProMechydirectionTypeGet()** returns the type of the y-direction. Pass the y-direction handle as the input to this function. The output value *type* can have the following values:

- `PRO_MECH_YDIR_VECTOR`—Specifies a direction vector.
- `PRO_MECH_YDIR_REF`—Specifies a referenced coordinate system.
- `PRO_MECH_YDIR_CSYS`—Specifies a world coordinate system.

The function **ProMechydirectionCsysGet()** returns the coordinate system if the specified y-direction handle is of type `PRO_MECH_YDIR_CSYS`.

The function **ProMechydirectionCsysSet()** sets the coordinate system for the specified y-direction handle. Calling this function changes the Y-direction type to the appropriate type and discards any data related to its previous type.

The function **ProMechydirectionReferenceGet()** returns the reference entity if the specified y-direction handle is of type `PRO_MECH_YDIR_REF`.

The function **ProMechydirectionReferenceSet()** sets the reference entity for the specified y-direction handle. Calling this function changes the Y-direction type to the appropriate type and discards any data related to its previous type.

The function **ProMechydirectionVectorGet()** returns the vector direction if the specified y-direction handle is of type `PRO_MECH_YDIR_VECTOR`.

The function **ProMechydirectionVectorSet()** sets the vector for the specified y-direction handle. Calling this function changes the Y-direction type to the appropriate type and discards any data related to its previous type.

Use the function **ProMechydirectionFree()** to free the y-direction handle.

Functions

The functions described in this section provide access to the data and contents of Mechanica function items.

Function items use the *ProType* field in the *ProMechitem* structure as `PRO_SIMULATION_FUNCTION`.

Functions Introduced:

- **ProMechfuncCsysGet()**
- **ProMechfuncDataGet()**

- **ProMechfuncDescriptionGet()**
- **ProMechfuncdataTypeGet()**
- **ProMechfuncdataExpressionGet()**
- **ProMechfuncdataFuncvalueinterpGet()**
- **ProMechfuncdataIndependentvarGet()**
- **ProMechfuncdataIndepvarinterpGet()**
- **ProMechfuncdataFuncTableGet()**
- **ProMechtableentryFunctionvalueGet()**
- **ProMechtableentryFree()**
- **ProMechtableentryProarrayFree()**
- **ProMechfuncdataFree()**

Use the function **ProMechfuncCsysGet()** to obtain the reference co-ordinate system for the specified Mechanica function item.

The function **ProMechfuncDataGet()** returns the handle to the function data of the Mechanica item.

Use the function **ProMechfuncDescriptionGet()** returns the description of the Mechanica function item.

Use the function **ProMechfuncdataTypeGet()** to determine the type of the function used to create the Mechanica item. Specify the handle to the function data as the input for this function.

The output argument *value* specifies the type of the function and can have the following values:

- **PRO_MECH_FUNCTION_SYMBOLIC**—Specifies a symbolic expression for a function.
- **PRO_MECH_FUNCTION_TABLE**—Specifies a function created using data from an interpolation table.

The function **ProMechfuncdataExpressionGet()** returns the symbolic expression for the specified symbolic function.

The function **ProMechfuncdataFuncvalueinterpGet()** specifies the interpolation method used for the function value of the tabular function. The output argument *value* can have the following values:

- **PRO_MECH_TABLE_LINEAR**—This method linearly interpolates the variable between the values.
- **PRO_MECH_TABLE_LOG**—This method linearly interpolates the log of the variable between values.

The function **ProMehfuncdataIndependentvarGet()** returns the type of the independent variable for the specified tabular function. The independent variable corresponds to the coordinate system axes and has the following values:

- **PRO_MECH_INDEP_VAR_X**—Specifies the value of the X-axis in the Cartesian coordinate system.
- **PRO_MECH_INDEP_VAR_Y**—Specifies the value of the Y-axis in Cartesian coordinate system
- **PRO_MECH_INDEP_VAR_Z**—Specifies the value of the Z-axis in the Cartesian or cylindrical coordinate system.
- **PRO_MECH_INDEP_VAR_R**—Specifies the value of the radius in a cylindrical or spherical coordinate system.
- **PRO_MECH_INDEP_VAR_THETA**—Specifies the value of the angle in a cylindrical or spherical coordinate system.
- **PRO_MECH_INDEP_VAR_PHI**—Specifies the value of the second angle in a spherical coordinate system.
- **PRO_MECH_INDEP_VAR_TIME**—Specifies the value of the time variable (for a time-dependent function).

The function **ProMehfuncdataIndepvarinterpGet()** specifies the interpolation method used for the independent variable of the tabular function.

The function **ProMehfuncdataFuncTableGet()** returns an array of table entries for the specified tabular function. Free the array of table entries using the function **ProMechtableentryProarrayFree()**.

The function **ProMechtableentryFunctionvalueGet()** returns the value of the specified function in the table entry.

The function **ProMechtableentryIndependentvarGet()** returns the value for the specified independent variable in the table entry.

Use the function **ProMehfuncdataFree()** to free the array containing the function data.

Mechanica Expressions

Most of the **ProMech*Get()** functions described in the following sections return mathematical expressions of the type *ProMechExpression*. This mathematical expression specifies a numeric or relational value, which is expressed and stored as a string. Use the function **ProMathExpressionEvaluate()** to evaluate such a mathematical expression. Refer to the “Evaluating Mathematical Expressions for a Solid” section in the “Core: Solids, Parts, and Materials” chapter for more information on this function.

Accessing the Properties used for Loads and Constraints

Loads and constraints use complicated structures to mirror the number and type of properties available in the user interface. Several structures recur in different types of loads and constraints, and are described by the following table:

Property	Pro/TOOLKIT Opaque Handle
Vectored value—Specifies a value applied in a specified direction.	<i>ProMechVectoredValue</i>
Direction vector—Represents a defined direction.	<i>ProMechDirectionVector</i>
Value—Specifies a scalar value potentially affected by a defined variation.	<i>ProMechValue</i>
Variation—Specifies the components of a defined variation (either by function or by interpolation).	<i>ProMechVariation</i>
Interpolation point—Specifies an individual interpolation point used to define a variation.	<i>ProMechInterpolationPnt</i>

The types specified in the preceding table and the functions required to access these types are defined in the header file `ProMechValue.h` and `ProMechValueSet.h`.

Functions Introduced:

- **ProMechvectoredvalueAlloc()**

- **ProMechvectoredvalueDirectiontypeGet()**
- **ProMechvectoredvalueDirectiontypeSet()**
- **ProMechvectoredvalueDirectionvectorGet()**
- **ProMechvectoredvalueDirectionvectorSet()**
- **ProMechvectoredvalueMagnitudeGet()**
- **ProMechvectoredvalueMagnitudeSet()**
- **ProMechvectoredvaluePointsGet()**
- **ProMechvectoredvaluePointsSet()**
- **ProMechvectoredvalueFree()**
- **ProMechdirvectorAlloc()**
- **ProMechdirvectorComponentsGet()**
- **ProMechdirvectorComponentsSet()**
- **ProMechdirvectorCsysGet()**
- **ProMechdirvectorCsysSet()**
- **ProMechdirvectorVariationGet()**
- **ProMechdirvectorVariationSet()**
- **ProMechdirvectorFree()**
- **ProMechvalueAlloc()**
- **ProMechvalueValueGet()**
- **ProMechvalueValueSet()**
- **ProMechvalueVariationGet()**
- **ProMechvalueVariationSet()**
- **ProMechvalueFree()**
- **ProMechvalueProarrayFree()**
- **ProMechvariationAlloc()**
- **ProMechvariationTypeGet()**
- **ProMechvariationFunctionidGet()**
- **ProMechvariationFunctionidSet()**
- **ProMechvariationInterpolationGet()**
- **ProMechvariationInterpolationSet()**
- **ProMechvariationFree()**

- **ProMechinterpolationpntAlloc()**
- **ProMechinterpolationpntPointGet()**
- **ProMechinterpolationpntPointSet()**
- **ProMechinterpolationpntMagnitudeGet()**
- **ProMechinterpolationpntMagnitudeSet()**
- **ProMechinterpolationpntProarrayFree()**
- **ProMechinterpolationpntFree()**
- **ProMechexternalfielddataAlloc()**
- **ProMechvariationExternalfielddataGet()**
- **ProMechvariationExternalfielddataSet()**
- **ProMechexternalfielddataCsysGet()**
- **ProMechexternalfielddataCsysSet()**
- **ProMechexternalfielddataFileGet()**
- **ProMechexternalfielddataFileSet()**
- **ProMechexternalfielddataFree()**

The function **ProMechvectedvalueAlloc()** allocates memory for the directed value.

The function **ProMechvectedvalueDirectiontypeGet()** returns the method used to specify the direction for the vectored value. The output argument *type* can have one of the following values:

- **PRO_MECH_DIRECTION_BY_VECTOR**—The vector is defined by specifying a direction.
- **PRO_MECH_DIRECTION_BY_2_POINTS**—The direction of the vector is specified using two points.

The function **ProMechvectedvalueDirectiontypeSet()** sets the direction for the vectored value.

The function **ProMechvectedvalueDirectionvectorGet()** returns the direction of vector for the vectored load of type **PRO_MECH_DIRECTION_BY_VECTOR**.

The function **ProMechvectedvalueDirectionvectorSet()** sets the direction vector.

The function **ProMechvectoredvalueMagnitudeGet()** specifies the magnitude of the vectored load. If the value of the magnitude is positive, the load acts in the same direction as the vector and if the value of the magnitude is negative, the load acts in the direction opposite to that of the vector.

The function **ProMechvectoredvalueMagnitudeSet()** sets the magnitude of the vectored load.

The function **ProMechvectoredvaluePointsGet()** returns an array of points that define the direction of the vector if type of the vector is `PRO_MECH_DIRECTION_BY_2_POINTS`. Use the function **ProMechvectoredvaluePointsSet()** to set the array of points.

Use the function **ProMechvectoredvalueFree()** to free the memory containing the directed value.

The function **ProMechdirvectorComponentsGet()** returns the components of the vector for each coordinate direction. Use the function **ProMechdirvectorComponentsSet()** to set the component values of the vector.

The function **ProMechdirvectorCsysGet()** returns the coordinate system used to calculate the vector. Use the function **ProMechdirvectorCsysSet()** to set the coordinate system.

The function **ProMechdirvectorVariationGet()** specifies the spatial or time-based variation applied to the direction vector. Use the function **ProMechdirvectorVariationSet()** to set the spatial variation to the direction vector.

The function **ProMechvalueValueGet()** returns the expression used to specify the load or constraint value. Use the function **ProMechvalueValueSet()** to set the load or constraint value.

The function **ProMechvalueVariationGet()** returns the spatial or time-based variation assigned to the specified value. Use the function **ProMechvalueVariationSet()** to set the spatial variation.

The function **ProMechvariationTypeGet()** returns the variation type for the specified variation handle. The output argument *type* can have one of the following values:

- `PRO_MECH_VARIATION_UNIFORM`—Specifies that the variation of the load is uniform over the entity.
- `PRO_MECH_VARIATION_INTERPOLATION`—Specifies that the load varies along the entity as defined by the interpolation points.

- **PRO_MECH_VARIATION_FUNCTION**—Indicates that the variation is defined by specifying a function that is used to vary the property.

The function **ProMechvariationFunctionidGet()** returns the function id for the specified variation handle if the variation is of type **PRO_MECH_VARIATION_FUNCTION**. The function id is always extracted from the same model from where the varied value was obtained. Use the function **ProMechvariationFunctionidSet()** to set the function id for the variation.

The function **ProMechvariationInterpolationGet()** returns an array of interpolation points for the specified variation handle if the variation is of type **PRO_MECH_VARIATION_INTERPOLATION**. Use the function **ProMechvariationInterpolationSet()** to set the interpolation points for the specified variation.

The function **ProMechinterpolationpntAlloc()** allocates memory for the interpolation points.

The function **ProMechinterpolationpntPointGet()** specifies the geometric entity associated with the specified interpolation point. Use the function **ProMechinterpolationpntPointSet()** to set the geometric entity.

The function **ProMechinterpolationpntMagnitudeGet()** returns the magnitude of the load at the specified interpolation point. Use the function **ProMechinterpolationpntMagnitudeSet()** to set the magnitude of the load.

Use the function **ProMechinterpolationpntFree()** to free the memory containing the interpolation point. Use **ProMechinterpolationpntProarrayFree()** as a shortcut to free an entire array of interpolation points.

The function **ProMechexternalfielddataAlloc()** allocates memory for the external data structure.

The function **ProMechvariationExternalfielddataGet()** returns the external field assigned for variation of a value.

Use the function **ProMechvariationExternalfielddataSet** to set the external field assigned for variation of a value.

The function **ProMechexternalfielddataCsysGet()** returns the reference coordinate system for the external field data. Use the function **ProMechexternalfielddataCsysSet()** to set the reference coordinate system for the external field data.

The function **ProMechexternalfielddataFileGet()** returns the information related to the external (FEM Neutral Format) FNF file to be imported for the external field. Use the function **ProMechexternalfielddataFileSet()** to set the path to the fnf file.

Mechanica Loads

Loads are used to simulate forces that act on the model in the real world. Once you create the loads, you can examine the response of the mechanism to the loads.

Accessing Mechanica Loads

The functions described in this section provide a handle to the to the different types of Mechanica loads. Loads use the ProType field in the ProMechitem structure as PRO_SIMULATION_LOAD.

Functions Introduced:

- **ProMechloadTypeGet()**
- **ProMechloadForcedataGet()**
- **ProMechloadPressuredataGet()**
- **ProMechloadBearingdataGet()**
- **ProMechloadCentrifugaldataGet()**
- **ProMechloadGravitydataGet()**
- **ProMechloadStructtempdataGet()**
- **ProMechloadGlobaltempdataGet()**
- **ProMechloadMecttempdataGet()**
- **ProMechloadExttempdataGet()**
- **ProMechloadHeatdataGet()**
- **ProMechloadReferencesGet()**
- **ProMechloadLoadsetsGet()**

The function **ProMechLoadTypeGet()** returns the type of load contained in the structure ProModelItem.

The output argument *value* can have one of the following values:

- **PRO_MECH_LOAD_FORCE**—Specifies a force load. Use the function **ProMechloadForcedataGet()** to access the data and contents of the force load structure.
- **PRO_MECH_LOAD_PRESSURE**—Specifies a pressure load. Use the function **ProMechloadPressuredataGet()** to access the data and contents of the pressure load structure.
- **PRO_MECH_LOAD_BEARING**—Specifies a bearing load. Use the function **ProMechloadBearingdataGet()** to access the data and contents of the bearing load structure.
- **PRO_MECH_LOAD_CENTRIFUGAL**—Specifies a centrifugal load. Use the function **ProMechloadCentrifugaldataGet()** to access the data and contents of the centrifugal load structure.
- **PRO_MECH_LOAD_GRAVITY**—Specifies a gravity load. Use the function **ProMechloadGravitydataGet()** to access the data and contents of the gravity load structure.
- **PRO_MECH_LOAD_STRUCTURAL_TEMPERATURE**—Specifies a temperature load. Use the function **ProMechloadStructtempdataGet()** to access the data and contents of the temperature load.
- **PRO_MECH_LOAD_GLOBAL_TEMPERATURE**—Specifies a global temperature load. Use the function **ProMechloadGlobaltempdataGet()** to access the data and contents of the global temperature load structure.
- **PRO_MECH_LOAD_MECH_TEMPERATURE**—Specifies a Mechanical temperature load. Use the function **ProMechloadMecttempdataGet()** to access the data and contents of the mechanical temperature load.
- **PRO_MECH_LOAD_EXTERNAL_TEMPERATURE**—Specifies an external temperature load. Use the function **ProMechloadExttempdataGet()** to access the data and contents of the external temperature load.
- **PRO_MECH_LOAD_HEAT**—Specifies a heat load. Use the function **ProMechloadHeatdataGet()** to access the data and contents of the heat load.

The function **ProMechloadReferencesGet()** returns the geometric references. It specifies the geometric references used to define the load.

The function **ProMechloadLoadsetsGet()** returns the load set(s) that contain the given load. (Currently, Pro/ENGINEER Mechanica allows a load to be assigned to only one set).

Modifying Mechanica Loads

The functions in this section assign load data to a load in the model. The load should have been created already using **ProMechitemCreate()**. Changes made via these functions will not be reflected in the user interface until you call the function **ProMechitemUpdateComplete()**.

Note: Once a load has been created and updated successfully, its load type cannot be changed.

The following is the procedure to create a new user-visible load:

1. Create the load using **ProMechitemCreate()**.
2. Set the load references using **ProMechloadReferencesSet()**.
3. Set the load type-specific data using one of the **ProMechload*dataSet()** functions.
4. Assign the load to a load set using **ProMechloadLoadsetAssign()**.
5. Check the status of the load using **ProMechitemStatusGet()**.
6. Complete the load set using **ProMechitemUpdateComplete()**.

Functions Introduced:

- **ProMechloadForcedataSet()**
- **ProMechloadPressuredataSet()**
- **ProMechloadBearingdataSet()**
- **ProMechloadCentrifugaldataSet()**
- **ProMechloadGravitydataSet()**
- **ProMechloadStructtempdataSet()**
- **ProMechloadGlobaltempdataSet()**
- **ProMechloadMecttempdataSet()**
- **ProMechloadExttempdataSet()**
- **ProMechloadHeatdataSet()**
- **ProMechloadReferencesSet()**
- **ProMechloadLoadsetAssign()**

Use the function **ProMechloadForcedataSet()** to set the data and the contents of the force load structure.

Use the function **ProMechloadPressuredataset()** to set the data and contents of the pressure load structure.

Use the function **ProMechloadBearingdataset()** to set the data and contents of the pressure load structure.

Use the function **ProMechloadCentrifugaldataset()** to set the data and contents of the centrifugal load structure.

Use the function **ProMechloadGravitydataset()** to set the data for the gravity load structure.

Use the function **ProMechloadStructtempdataset()** to set the data of the temperature load.

Use the function **ProMechloadGlobaltempdataset()** to set the data for the global temperature load structure.

Use the function **ProMechloadMecttempdataset()** to set the data of the mechanical temperature load.

Use the function **ProMechloadExttempdataset()** to set the data and contents of the external temperature load.

Use the function **ProMechloadHeatdataset()** to set the data and contents of the heat load.

Use the function **ProMechloadReferencesSet()** to set the geometric references for the load. This must be a valid set of references for the load type.

Use the function **ProMechloadLoadsetAssign()** to assign a load to a particular loadset.

Example 2: Modifying Magnitude of Force or Pressure Load

This example shows how to modify the magnitude of a force or pressure load.

```
#include <ProMechLoadsSet.h>
#include <ProUtil.h>
#include <ProMessage.h>
#include <ProMechValueSet.h>
#include <PTMechExamples.h>

/*=====*\
  FUNCTION: PTMechExLoadPostfilter
  PURPOSE: Selection filter to ensure that only pressure and force loads
           are selected
\*=====*/
```

```

static ProError PTMechExLoadPostfilter (ProSelection selection,
                                         ProAppData app_data)
{
    ProModelitem mech_item;
    ProMechLoadType load_type;

    status = ProSelectionModelitemGet (selection,
                                         (ProModelitem*)&mech_item);
    PT_TEST_LOG_SUCC ("ProSelectionModelitemGet()");

    if (mech_item.type != PRO_SIMULATION_LOAD)
        return (PRO_TK_CONTINUE);

    status = ProMechloadTypeGet (&mech_item,&load_type);
    PT_TEST_LOG_SUCC ("=ProMechloadTypeGet");

    if (((load_type != PRO_MECH_LOAD_FORCE) &&
         (load_type != PRO_MECH_LOAD_PRESSURE)) ||
        (status != PRO_TK_NO_ERROR))
        return (PRO_TK_CONTINUE);

    return (PRO_TK_NO_ERROR);
}

/*=====*\
FUNCTION: PTMechExMagChange
PURPOSE: Increases the magnitude of force/pressure
\*=====*/

ProError PTMechExMagChange ()
{
    ProSelection* sels;
    ProSelFunctions sel_funcs;
    ProMechItem mech_item;
    int n_sels, i=0;
    ProMdl constraint_part;
    ProMechLoadType load_type;
    ProMechVectorValue force_vect_value;
    ProMechValue force_value_mag;
    ProMechForceData force_data;
    ProMechStatus mech_status;
    ProMechPressureData press_data;
    ProMechValue press_mech_val;
    double mag;
    ProLine val,val_5times;
    char curr_mag1[PRO_LINE_SIZE];
    char curr_mag2[PRO_LINE_SIZE];

    status = ProMdlCurrentGet(&constraint_part);
    PT_TEST_LOG_SUCC ("ProMdlCurrentGet");

```

```

sel_funcs.pre_filter = NULL;
sel_funcs.post_filter = PTMechExLoadPostfilter;
sel_funcs.post_select = NULL;
sel_funcs.app_data = NULL;

while (1)
{
    status = ProMessageDisplay(msgfile,
                                "PTMechEx Select the load", msgbuf);
    PT_TEST_LOG_SUCC("ProMessageDisplay()");

    status =ProSelect ("sim_load",
                        1,
                        NULL,
                        &sel_funcs,
                        NULL,
                        NULL,
                        &sels,
                        &n_sels);
    PT_TEST_LOG_SUCC ("ProSelect()");

    if( status != PRO_TK_NO_ERROR || n_sels <= 0 )
        break;

    status = ProSelectionModelitemGet (sels[i],
                                       (ProModelitem*)&mech_item);
    PT_TEST_LOG_SUCC ("ProSelectionModelitemGet()");

    status =ProMechloadTypeGet (&mech_item,&load_type);
    PT_TEST_LOG_SUCC ("=ProMechloadTypeGet");

/*-----*\
    Get the force data
/*-----*/

    if (load_type == PRO_MECH_LOAD_FORCE)
    {

        status = ProMechloadForcedataGet (&mech_item, &force_data);
        PT_TEST_LOG_SUCC ("ProMechloadForcedataGet");

        status = ProMechforcedataForceGet (force_data,&force_vect_value);
        PT_TEST_LOG_SUCC("ProMechforcedataForceGet");

        status = ProMechvectoredvalueMagnitudeGet
            (force_vect_value,&force_value_mag);
        PT_TEST_LOG_SUCC("ProMechvectoredvalueMagnitudeGet");
/*-----*\
        Check if the magnitude is required field
/*-----*/

```

```

    if (status == PRO_TK_E_NOT_FOUND)
    {
        status = ProMessageDisplay(msgfile,
                                   "PTMechEx Magnitude is not used for this load.");
        PT_TEST_LOG_SUCC("ProMessageDisplay()");
        continue;
    }

/*-----*\
    Get the magnitude value
\*-----*/

    status = ProMechvalueValueGet (force_value_mag,val);
    PT_TEST_LOG_SUCC("ProMechvalueValueGet");

/*-----*\
    Increase the magnitude
\*-----*/

    if ( status == PRO_TK_NO_ERROR)
    {
        ProWstringToString(curr_mag1,val);
        PTMechExMagValueIncrease(val,val_5times);

/*-----*\
    Set the magnitude value
\*-----*/

    status = ProMechvalueValueSet (force_value_mag,val_5times);
    PT_TEST_LOG_SUCC ("ProMechvalueValueSet");

    status = ProMechvectoredvalueMagnitudeSet (force_vect_value,
                                                force_value_mag);
    PT_TEST_LOG_SUCC ("ProMechvectoredvalueMagnitudeSet");

    status = ProMechforcedataForceSet (force_data, force_vect_value);
    PT_TEST_LOG_SUCC ("ProMechforcedataForceSet");

    }

/*-----*\
    Set the force data
\*-----*/

    status = ProMechloadForcedataSet (&mech_item,force_data);
    PT_TEST_LOG_SUCC("ProMechloadForcedataSet");

/*-----*\
    Confirm the changed value
\*-----*/

    status = ProMechloadForcedataGet (&mech_item, &force_data);

```

```

PT_TEST_LOG_SUCC ("ProMechloadForcedataGet");

status = ProMechforcedataForceGet (force_data,&force_vect_value);
PT_TEST_LOG_SUCC("ProMechforcedataForceGet");

status = ProMechvectoredvalueMagnitudeGet
        (force_vect_value,&force_value_mag);
PT_TEST_LOG_SUCC("ProMechvectoredvalueMagnitudeGet");

status = ProMechvalueValueGet (force_value_mag,val);
PT_TEST_LOG_SUCC("ProMechvalueValueGet");

ProWstringToString(curr_mag2,val);
}

if (load_type == PRO_MECH_LOAD_PRESSURE)

/*-----*\
  Get the pressure data
  \*-----*/
{
    status = ProMechloadPressuredataGet (&mech_item,&press_data);
    PT_TEST_LOG_SUCC("ProMechloadPressuredataGet");

    status = ProMechpressuredataValueGet (press_data,&press_mech_val);
    PT_TEST_LOG_SUCC("ProMechpressuredataValueGet");

/*-----*\
  Get the value
  \*-----*/

    status = ProMechvalueValueGet (press_mech_val,val);
    PT_TEST_LOG_SUCC("ProMechvalueValueGet");

/*-----*\
  Increase the magnitude
  \*-----*/

    if ( status == PRO_TK_NO_ERROR)

    {

        ProWstringToString(curr_mag1,val);
        PTMechExMagValueIncrease(val,val_5times);

        status = ProMechvalueValueSet (press_mech_val,val_5times);
        PT_TEST_LOG_SUCC ("ProMechvalueValueSet");

        status = ProMechpressuredataValueSet
                (press_data,press_mech_val);
        PT_TEST_LOG_SUCC("ProMechpressuredataValueSet");
    }
}

```

```

    }

/*-----*\
Set the pressure data
\*-----*/

    status = ProMechloadPressuredataset (&mech_item,press_data);
    PT_TEST_LOG_SUCC("ProMechloadPressuredataset");

/*-----*\
Confirm the changed value
\*-----*/

    status = ProMechloadPressuredataset (&mech_item,&press_data);
    PT_TEST_LOG_SUCC("ProMechloadPressuredataset");

    status = ProMechpressuredataValueGet (press_data,&press_mech_val);
    PT_TEST_LOG_SUCC("ProMechpressuredataValueGet");

    status = ProMechvalueValueGet (press_mech_val,val);
    PT_TEST_LOG_SUCC("ProMechvalueValueGet");

    ProWstringToString(curr_mag2,val);
}

status = ProMechitemUpdateComplete (&mech_item);
PT_TEST_LOG_SUCC ("ProMechitemUpdateComplete");

status = ProMechitemStatusGet (&mech_item, &mech_status);
PT_TEST_LOG_SUCC ("ProMechitemStatusGet");

/*-----*\
Print Message from the result
\*-----*/
    if (mech_status == PRO_MECH_ACTIVE)
    {
        status = ProMessageDisplay(msgfile, "PTMechEx Magnitude changed
                                           successfully",curr_mag1,curr_mag2);
        PT_TEST_LOG_SUCC("ProMessageDisplay()");
    }
    else
    {
        status = ProMessageDisplay(msgfile, "PTMechEx Magnitude modification
                                           failed", msgbuf);
        PT_TEST_LOG_SUCC("ProMessageDisplay()");
    }

}
return PRO_TK_NO_ERROR;
}

```

Force and Moment Loads

The functions described in this section provide read access to the data and contents of Mechanica force loads.

Functions Introduced:

- **ProMechforcedataAlloc()**
- **ProMechforcedataTypeGet()**
- **ProMechforcedataTypeSet()**
- **ProMechforcedataForceGet()**
- **ProMechforcedataAxialForceIgnoreflagGet()**
- **ProMechforcedataAxialForceIgnoreflagSet()**
- **ProMechforcedataForceSet()**
- **ProMechforcedataMomentGet()**
- **ProMechforcedataAxialMomentIgnoreflagGet()**
- **ProMechforcedataAxialMomentIgnoreflagSet()**
- **ProMechforcedataMomentSet()**
- **ProMechforcedataRefpntGet()**
- **ProMechforcedataRefpntSet()**
- **ProMechforcedataFree()**

The function **ProMechforcedataAlloc()** allocates memory for the Mechanica force load data.

The function **ProMechforcedataTypeGet()** specifies the distribution of the load over the selected entity. The output argument *value* can have one of the following values:

- **PRO_MECH_FORCE_AT_POINT**—Specifies a total load applied to a single point.
- **PRO_MECH_FORCE_TOTAL**—Specifies a load distributed along the length or area of the entity such that the integral of the load over the selected entity equals the total prescribed value.
- **PRO_MECH_FORCE_TOTAL_AT_POINT**—Specifies a Total Load At Point (TLAP) load.
- **PRO_MECH_FORCE_PER_UNIT**—Specifies a load applied to each unit that makes up the selected entity.

- **PRO_MECH_FORCE_TLAP_UNASSOCIATED**—Specifies that the Total Load At Point (TLAP) loads are un-associated from the selected entities in the model.
- **PRO_MECH_FORCE_TBLAP**—Specifies a Total Bearing Load At Point (TBLAP) for cylindrical surfaces or curves.

Use function **ProMechforcedataTypeSet()** to set the type of the force load.

The function **ProMechforcedataForceGet()** returns the magnitude and direction of the force applied to a geometric entity.

The function **ProMechforcedataAxialForceIgnoreflagGet()** returns a boolean flag indicating whether the axial force is ignored in case of a Total Bearing Load At Point (TBLAP).

ProMechforcedataAxialForceIgnoreflagSet() sets a boolean flag indicating whether the axial force is to be applied in case of a Total Bearing Load At Point (TBLAP).

The function **ProMechforcedataForceSet()** sets the force component for the load data.

The function **ProMechforcedataMomentGet()** returns the magnitude and direction of the moment applied to a geometric entity.

ProMechforcedataAxialMomentIgnoreflagGet() returns a boolean flag indicating whether the axial moment applied is ignored in case of a Total Bearing Load At Point (TBLAP).

ProMechforcedataAxialMomentIgnoreflagSet() sets a boolean flag indicating whether the axial moment is to be applied in case of a Total Bearing Load At Point (TBLAP).

The function **ProMechforcedataMomentSet()** sets the moment component for the load data.

The function **ProMechforcedataRefpntGet()** returns the reference point for the specified load if the load was applied at a given point. Use the function **ProMechforcedataRefpntSet()** to set the reference point.

The function **ProMechforcedataFree()** frees the memory of the force load data handle.

Pressure, Gravity, and Bearing Loads

The functions described in this section provide read access to the data and contents of the Mechanics pressure, gravity, and bearing loads.

Functions Introduced:

- **ProMechpressuredataAlloc()**
- **ProMechpressuredataValueGet()**
- **ProMechpressuredataValueSet()**
- **ProMechpressuredataFree()**
- **ProMechbearingdataValueGet()**
- **ProMechbearingdataValueSet()**
- **ProMechbearingdataFree()**
- **ProMechgravitydataAlloc()**
- **ProMechgravitydataValueGet()**
- **ProMechgravitydataValueSet()**
- **ProMechgravitydataFree()**

The function **ProMechpressuredataAlloc()** allocates memory for the Mechanica pressure load data.

The function **ProMechpressuredataValueGet()** returns the value of the pressure load.

The function **ProMechpressuredataValueSet()** sets the value of the pressure load.

The function **ProMechpressuredataFree()** frees the memory of the pressure load data handle.

The function **ProMechbearingdataValueGet()** returns the value of the bearing load.

The function **ProMechbearingdataValue Set()** sets the value of the bearing load.

The function **ProMechbearingdataFree()** frees the memory of the bearing load data handle.

The function **ProMechgravitydataAlloc()** allocates memory for the Mechanica gravity load data.

The function **ProMechgravitydataValueGet()** returns the value of the gravity load.

The function **ProMechgravitydataValueSet()** sets the value of the gravity load.

The function **ProMechgravitydataFree()** frees the memory of the gravity load data handle.

Centrifugal Loads

A centrifugal load results due to the rigid body rotation of the model and is applied on the entire model. The functions described in this section provide read access to the data and structure of the Mechanica centrifugal loads.

Functions Introduced:

- **ProMechcentrifugaldataAlloc()**
- **ProMechcentrifugaldataVelocityGet()**
- **ProMechcentrifugaldataVelocitySet()**
- **ProMechcentrifugaldataAccelerationGet()**
- **ProMechcentrifugaldataAccelerationSet()**
- **ProMechcentrifugaldataFree()**

The function **ProMechcentrifugaldataAlloc()** allocates memory for the centrifugal load data.

The function **ProMechcentrifugaldataVelocityGet()** returns the magnitude and direction of the angular velocity of the centrifugal load.

The function **ProMechcentrifugaldataVelocitySet()** sets the value of the angular velocity of the centrifugal load.

The function **ProMechcentrifugaldataAccelerationGet()** returns the acceleration vector for the centrifugal load. The acceleration vector is the rate of change of the angular velocity vector, and represents a change in magnitude or direction of the angular velocity.

The function **ProMechcentrifugaldataAccelerationSet()** sets the acceleration vector for the centrifugal load.

The function **ProMechcentrifugaldataFree()** releases the memory assigned to the centrifugal load data.

Temperature Loads

Temperature loads enable you to simulate a temperature change over your model. Temperature loads provide valuable information on how the structure of your model deforms due to a particular temperature change.

Mechanica Temperature Loads

The functions described in this section provide access to the data and contents of Mechanica temperature loads.

Functions Introduced:

- **ProMechtemperatedataAlloc()**
- **ProMechtemperatedataValueGet()**
- **ProMechtemperatedataValueSet()**
- **ProMechtemperatedataFree()**

The function **ProMechtemperatedataAlloc()** allocates memory for the temperature load data.

The function **ProMechtemperatedataValueGet()** returns a structure that contains the value of the temperature and the spatial variation of the load.

The function **ProMechtemperatedataValueSet()** sets the value of the temperature and the spatial variation of the load.

The function **ProMechtemperatedataFree()** releases the memory assigned to the temperature load data.

Structural Temperature Loads

Structural temperature loads are thermal loads resulting from a temperature change over a geometric entity or a set of geometric entities. This load is applied to points, curves, or surfaces. If the model is an assembly, the structural load can be applied to specific assembly components.

The functions described in this section provide access to the data and contents of the Mechanica structural temperature loads.

Functions Introduced:

- **ProMechstructtempdataAlloc()**
- **ProMechstructtempdataValueGet()**
- **ProMechstructtempdataValueSet()**
- **ProMechstructtempdataReftempGet()**
- **ProMechstructtempdataReftempSet()**
- **ProMechstructtempdataFree()**

The function **ProMechstructtempdataAlloc()** allocates memory for the structural temperature load data.

The function **ProMechstructtempdataValueGet()** returns a structure containing the temperature applied to the load and the variation of the load.

The function **ProMechstructtempdataValueSet()** sets the value of the structural temperature data.

The function **ProMechstructtempdataReftempGet()** returns the reference temperature of the structural temperature object. The reference temperature is the stress-free temperature for the model.

The function **ProMechstructtempdataReftempSet()** sets the reference temperature of the structural temperature data.

The function **ProMechstructtempdataFree()** releases the memory assigned to structural temperature load data.

Global Temperature Loads

A global temperature load specifies a thermal load resulting from a temperature change over the entire model.

Functions Introduced:

- **ProMechglobaltempdataAlloc()**
- **ProMechglobaltempdataReftempGet()**
- **ProMechglobaltempdataReftempSet()**
- **ProMechglobaltempdataValueGet()**
- **ProMechglobaltempdataValueSet()**
- **ProMechglobaltempdataFree()**

The function **ProMechglobaltempdataAlloc()** allocates memory for the global temperature data structure.

The function **ProMechglobaltempdataReftempGet()** returns the zero stress temperature for the global temperature data.

The function **ProMechglobaltempdataReftempSet()** sets the reference temperature of the global temperature data.

The function **ProMechglobaltempdataValueGet()** returns the value of the temperature applied to the model and the distribution of the load over the model.

The function **ProMechglobaltempdataValueSet()** sets the value of the temperature data.

The function **ProMechglobaltempdataFree()** releases the memory assigned to the global temperature data structure.

MEC/T Temperature Loads

The MEC/T temperature load applies a temperature load across the entire model based on a temperature field developed from the results of a steady state or transient thermal analysis. The functions described in this section provide read and write access to the data and contents of the MEC/T temperature loads.

Functions Introduced:

- **ProMechmecttempdataAlloc()**
- **ProMechmecttempdataAnalysisidGet()**
- **ProMechmecttempdataAnalysisidSet()**
- **ProMechmecttempdataLoadsetidGet()**
- **ProMechmecttempdataLoadsetidSet()**
- **ProMechmecttempdataLoadsetSet()**
- **ProMechmecttempdataLoadsetGet()**
- **ProMechmecttempdataReftempGet()**
- **ProMechmecttempdataReftempSet()**
- **ProMechmecttempdataTimestepGet()**
- **ProMechmecttempdataTimestepSet()**
- **ProMechmecttempdataDesignstudySet()**
- **ProMechmecttempdataDesignstudyGet()**
- **ProMechmecttempdataFree()**

The function **ProMechmecttempdataAlloc()** allocates memory for the MEC/T temperature data structure.

The function **ProMechmecttempdataAnalysisidGet()** returns the ID of the thermal analysis defined for the model.

The function **ProMechmecttempdataAnalysisidSet()** sets the ID of the thermal analysis.

The function **ProMechmecttempdataLoadsetidGet()** returns the ID of the load set. The load set is applicable only if the MEC/T temperature load is coming from a steady state thermal analysis.

The function **ProMechmecttempdataLoadsetidSet()** sets the ID of the load set.

The function **ProMechmecttempdataLoadsetGet()** returns the load set in the form of the *ProMechObjectref* object for the MEC/T temperature data.

The function **ProMechmecttempdataLoadsetSet()** assigns the load set.

The function **ProMechmecttempdataReftempGet()** returns the reference temperature for the MEC/T temperature data.

The function **ProMechmecttempdataReftempSet()** sets the reference temperature.

The function **ProMechmecttempdataTimestepGet()** returns the time step value if the MEC/T temperature load is coming from a transient thermal analysis.

The function **ProMechmecttempdataTimestepSet()** sets the time step value.

The function **ProMechmecttempdataDesignstudyGet()** returns the name of the design study for the MEC/T temperature load.

The function **ProMechmecttempdataDesignstudySet()** assigns the design study name for the MEC/T temperature load.

The function **ProMechmecttempdataFree()** releases the memory assigned to the MEC/T temperature data structure.

External Temperature Loads

An external temperature load involves importing an externally calculated or measured temperature field as a temperature load. The external temperature field must contain connectivity of a linear solid element mesh, node locations, and temperature values at the nodes.

The functions described in this section provide read and write access to the data and contents of the Mechanica external temperature load.

Functions Introduced:

- **ProMechexttempdataAlloc()**
- **ProMechexttempdataReftempGet()**
- **ProMechexttempdataReftempSet()**
- **ProMechexttempdataFemneutralfileGet()**
- **ProMechexttempdataFemneutralfileSet()**
- **ProMechexttempdataFree()**

The function **ProMechexttempdataAlloc()** allocates memory for the external temperature load data.

The function **ProMechexttempdataReftempGet()** returns the reference temperature data for the external data structure.

The function **ProMechexttempdataReftempSet()** sets the reference temperature data.

The function **ProMechexttempdataFemneutralfileGet()** returns the name of the external temperature field file which must be of the type FEM Neutral Format (.fnf).

The function **ProMechexttempdataFemneutralfileSet()** sets the name of the external temperature field file.

The function **ProMechexttempdataFree()** releases the memory assigned to the external temperature load data.

Heat Loads

Heat loads are entity loads and can be placed on one or more points, edges, curves, surfaces, components, or volumes. Heat loads provide local heat sources and sinks for the model and can be used to model internal heat generation or flux.

The functions described in this section provide access to the data and contents of the Mechanica heat loads.

Functions Introduced:

- **ProMechheatdataAlloc()**
- **ProMechheatdataTypeGet()**
- **ProMechheatdataTypeSet()**
- **ProMechheatdataValueGet()**
- **ProMechheatdataValueSet()**
- **ProMechheatdataTimefunctionidGet()**
- **ProMechheatdataTimefunctionidSet()**
- **ProMechheatdataFree()**

The function **ProMechheatdataAlloc()** allocates memory for the heat load data.

The function **ProMechheatdataTypeGet()** returns the type of distribution for the heat load across the geometric entities. The output argument type can have one of the following values:

- **PRO_MECH_HEAT_TOTAL**—Specifies that the heat load is distributed along the length or area of the entity such that the integral of the load over the selected entity equals the total prescribed value.
- **PRO_MECH_HEAT_PER_UNIT**—Specifies that the heat load is applied to each unit that makes up the selected load.
- **PRO_MECH_HEAT_AT_POINT**—Specifies that the heat load is applied to a single point, a feature, or a pattern of points.
- **PRO_MECH_HEAT_NONE**—No heat load type is assigned.

The function **ProMechheatdataTypeSet()** sets the type of heat load.

The function **ProMechheatdataValueGet()** returns the total or distributed heat transfer rate, depending on the distribution option specified for the entities.

The function **ProMechheatdataValueSet()** sets the value of the heat load.

The function **ProMechheatdataTimefunctionidGet()** returns the time function variation for the specified heat load.

ProMechheatdataTimefunctionidSet() sets the time function variation for the specified heat load.

The function **ProMechheatdataFree()** releases the memory assigned to the heat load data.

Mechanica Load Sets

A load set is a collection of loads that act together on the model. The functions described in this section provide access to data and contents of the Mechanica load set items. Load sets use the ProType field in the ProMechitem structure as PRO_SIMULATION_LOAD_SET.

Functions Introduced:

- **ProMechloadsetDescriptionGet()**
- **ProMechloadsetDescriptionSet()**
- **ProMechloadsetLoadsGet()**
- **ProMechloadsetTypeGet()**
- **ProMechloadLoadsetAssign()**

The function **ProMechloadsetDescriptionGet()** returns the name and the description of the specified load set.

The function **ProMechloadsetDescriptionSet()** sets the description of the specified load set.

The function **ProMechloadsetLoadsGet()** returns an array containing the different loads that are included in the specified load set.

The function **ProMechloadsetTypeGet()** returns the type of the specified load set. The type can be as follows:

- PRO_MECH_LOADSET_STRUCTURAL—Specifies a structural load set.
- PRO_MECH_LOADSET_THERMAL—Specifies a thermal load set.

Use the function **ProMechloadLoadsetAssign()** to assign a load to a particular loadset.

Mechanica Constraints

To perform analyses on the models you need to apply constraints to at least one area of the model. The constraints are associated with the model geometry and can be applied to a single geometric entity or to multiple entities.

Accessing Mechanics Constraints

The functions described in this section provide access to the data and contents of the Mechanics constraints item. Constraints use the `ProType` field in the `ProMechItem` structure as `PRO_SIMULATION_CONSTRAINTS`.

Functions Introduced:

- **ProMechconstrTypeGet()**
- **ProMechconstrConvectiondataGet()**
- **ProMechconstrDisplacementdataGet()**
- **ProMechconstrSymmetrydataGet()**
- **ProMechconstrTemperaturredataGet()**
- **ProMechconstrThermalsymmetrydataGet()**
- **ProMechconstrReferencesGet()**
- **ProMechconstrConstrsetsGet()**

The function **ProMechconstrTypeGet()** returns the type of constraint contained in the structure `ProModelItem`. The output can have one of the following values:

- `PRO_MECH_CONSTR_CONVECTION`—Specifies a linear convective heat exchange condition for one or more geometric or model entities for thermal mode. Use the function **ProMechconstrConvectiondataGet()** to access the data and contents of this constraint type.
- `PRO_MECH_CONSTR_DISPLACEMENT`—Specifies a displacement constraint for structural mode. Use the function **ProMechconstrDisplacementdataGet()** to access the data and contents of this constraint type.
- `PRO_MECH_CONSTR_SYMMETRY`—Specifies cyclic symmetry for structural mode. Use the function **ProMechconstrSymmetrydataGet()** to access the data and contents of this constraint type.

- **PRO_MECH_CONSTR_TEMPERATURE**—Specifies a temperature boundary condition for one or more geometric or model entities for thermal mode. Use the function **ProMechconstrTemperaturedataGet()** to access the data and contents of this constraint type.
- **PRO_MECH_CONSTR_SYMMETRY_THERM**—Specifies a cyclic symmetry thermal constraint for thermal mode. Use the function **ProMechconstrThermalsymmetrydataGet()** to access the data and contents of this constraint type.

The function **ProMechconstrReferencesGet()** returns the geometric references. It specifies the geometric references used to define the constraint.

The function **ProMechconstrConstrsetsGet()** returns the constraint set(s) that contain the given constraint. Currently, Pro/ENGINEER Mechanica allows a constraint to be assigned to only one set.

Modifying Mechanica Constraints

The functions in this section assign load data to a constraint in the model. The load should have been already created using **ProMechitemCreate()**. The changes made using these functions will not be reflected in the user interface until you call **ProMechitemUpdateComplete()**.

Note: Once a constraint has been created and updated successfully, its constraint type cannot be changed.

The following is the procedure to create a new user-visible constraint:

1. Create the load using **ProMechitemCreate()**.
2. Set the constraint references using **ProMechconstrReferencesSet()**.
3. Set the constraint type-specific data using one of the **ProMechconstr*dataSet()** functions.
4. Assign the constraint to a constraint set using **ProMechconstrConstrsetAssign()**.
5. Check the status of the constraint using **ProMechitemStatusGet()**.
6. Complete the constraint using **ProMechitemUpdateComplete()**.

Functions Introduced:

- **ProMechconstrConvectiondataSet()**
- **ProMechconstrDisplacementdataSet()**
- **ProMechconstrSymmetrydataSet()**
- **ProMechconstrTemperaturedataSet()**
- **ProMechconstrThermalsymmetrydataSet()**
- **ProMechconstrReferencesSet()**
- **ProMechconstrConstrsetAssign()**

The function **ProMechconstrConvectiondataSet()** sets the convection constraint data.

The function **ProMechconstrDisplacementdataSet()** sets the displacement constraint data.

The function **ProMechconstrRadiationdataSet()** sets the value for the radiation constraint data.

The function **ProMechconstrSymmetrydataSet()** sets the value of the symmetry constraint data.

The function **ProMechconstrTemperaturedataSet()** sets the value of the temperature constraint data.

The function **ProMechconstrThermalsymmetrydataSet()** sets the value of the thermal symmetry constraint data.

The function **ProMechconstrReferencesSet()** sets the value of the constraint geometric references. This must be a valid set of references for the constraint type.

Use the function **ProMechconstrConstrsetAssign()** to assign a constraint to a constraint set.

Convection Constraints

Convection constraints specify a boundary condition on the convective heat exchange between a moving fluid and geometric entities and/or element entities within your model.

The functions described in this section provide access to the data and contents of the Mechanics convection constraints.

Functions Introduced:

- **ProMechconvectiondataAlloc()**
- **ProMechconvectiondataBulktempGet()**
- **ProMechconvectiondataBulktempSet()**
- **ProMechconvectiondataBulktempUnset()**
- **ProMechconvectiondataFilmcoefficientGet()**
- **ProMechconvectiondataFilmcoefficientSet()**
- **ProMechconvectiondataTimefunctionidGet()**
- **ProMechconvectiondataTimefunctionidSet()**
- **ProMechconvectiondataFree()**

The function **ProMechconvectiondataAlloc()** allocates memory for the convection load data.

The function **ProMechconvectiondataBulktempGet()** returns the value of the bulk temperature. The bulk temperature specifies the temperature of the fluid in contact with the surface, during convective heat transfer through a surface.

The function **ProMechconvectiondataBulktempSet()** sets the value of the bulk temperature.

The function **ProMechconvectiondataFilmcoefficientGet()** returns the value of the film coefficient. The film coefficient specifies the constant of proportionality between the flux through the surface and the difference between the surface temperature and the bulk temperature, in convective heat transfer through a surface.

Use the function **ProMechconvectiondataFilmcoefficientSet()** to set the value of the film coefficient.

The function **ProMechconvectiondataTimefunctionidGet()** returns the convection constraint specified as a function of time. The function **ProMechconvectiondataTimefunctionidSet()** sets the convection constraint specified as a function of time.

The function **ProMechconvectiondataFree()** releases the memory assigned to the convection load data.

Displacement Constraints

The functions described in this section provide access to the data and contents of the Mechanica displacement constraints.

Functions Introduced:

- **ProMechdisplacementdataAlloc()**
- **ProMechdisplacementdataTypeGet()**
- **ProMechdisplacementdataTypeSet()**
- **ProMechdisplacementdataCsysGet()**
- **ProMechdisplacementdataCsysSet()**
- **ProMechdisplacementdataRotationconstrsGet()**
- **ProMechdisplacementdataRotationconstrsSet()**
- **ProMechdisplacementdataTranslationconstrsGet()**
- **ProMechdisplacementdataTranslationconstrsSet()**
- **ProMechdisplacementdataFree()**
- **ProMechdisplacementregularconstrAlloc()**
- **ProMechdisplacementregularconstrTypeSet()**
- **ProMechdisplacementregularconstrTypeGet()**
- **ProMechdisplacementregularconstrValueSet()**
- **ProMechdisplacementregularconstrValueGet()**
- **ProMechdisplacementdataTranslationinterpretinradiansflagGet()**
- **ProMechdisplacementdataTranslationinterpretinradiansflagSet()**
- **ProMechdisplacementregularconstrFree()**
- **ProMechdisplacementregularconstrProarrayFree()**
- **ProMechdisplacementdataPinconstrSet()**
- **ProMechdisplacementdataPinconstrGet()**
- **ProMechdisplacementpinconstrAlloc()**
- **ProMechdisplacementdataPinangularconstrTypeSet()**
- **ProMechdisplacementdataPinangularconstrTypeGet()**
- **ProMechdisplacementdataPinaxialconstrTypeSet()**
- **ProMechdisplacementdataPinaxialconstrTypeGet()**
- **ProMechdisplacementpinconstrFree()**

The function **ProMechdisplacementdataAlloc()** allocates the memory for the displacement constraint data handle.

The function **ProMechdisplacementdataTypeGet()** returns the type of displacement constraint data. The types of displacement constraints are:

- PRO_MECH_DISPLACEMENT_REGULAR—Specifies an external limit on the movement of a portion of the model.
- PRO_MECH_DISPLACEMENT_PLANE—This constraint type allows full planar movement, but constrains any off-plane displacement.
- PRO_MECH_DISPLACEMENT_PIN— Creates a constraint along a cylindrical surface for 3D models.
- PRO_MECH_DISPLACEMENT_BALL— Creates a constraint along a spherical surface for 3D models.

Use the function **ProMechdisplacementdataTypeSet()** to set the displacement constraint data.

The function **ProMechdisplacementdataCsysGet()** returns the reference coordinate system for the displacement constraint. Use the function **ProMechdisplacementdataCsysSet()** to set the reference coordinate system for the displacement constraint.

The function **ProMechdisplacementdataRotationconstrsGet()** returns the rotational component of the displacement about the X, Y, and Z axis. Use the function **ProMechdisplacementdataRotationconstrsSet()** to set the rotational component of displacement.

The function **ProMechdisplacementdataTranslationconstrsGet()** returns the translational component of the displacement about the X, Y, and Z directions. Use the function **ProMechdisplacementdataTranslationconstrsSet()** to set the translational component of displacement.

The function **ProMechdisplacementregularconstrAlloc()** allocates memory for the regular displacement constraint data structure.

The method **ProMechdisplacementregularconstrTypeGet()** returns the type of setting for the displacement constraint. Valid values are:

- PRO_MECH_DISPLACEMENT_FREE—Allows freedom of movement in the specified direction.
- PRO_MECH_DISPLACEMENT_FIXED—Constrains the entity, preventing movement in the specified direction.

- **PRO_MECH_DISPLACEMENT_ENFORCED**—Specifies an enforced displacement or rotation in the specified direction.

The function **ProMechdisplacementregularconstrTypeSet()** sets the type of the displacement constraint.

The function **ProMechdisplacementregularconstrValueGet()** returns the variation settings of the displacement if the type of displacement constraint is **PRO_MECH_DISPLACEMENT_ENFORCED**. The variation settings are as follows:

- An enforced displacement value in length units for the translational component
- An enforced rotation in radians for the rotational component

Use the function

ProMechdisplacementregularconstrValueSet() to set the variation settings of the displacement.

The function

ProMechdisplacementdataTranslationinterpretinradiansflagGet() returns the value true if the angular translations are interpreted in radians. This is applicable only if the displacement is of type **PRO_MECH_DISPLACEMENT_REGULAR** and if cylindrical or spherical coordinate system is selected for the translations. Use the function

ProMechdisplacementdataTranslationinterpretinradiansflagSet() to set the value of the flag interpret angular translations in radians.

The function **ProMechdisplacementregularconstrFree()** releases the memory assigned to the regular displacement constraint data handle.

The function

ProMechdisplacementregularconstrProarrayFree() releases the memory assigned to an array of regular displacement constraints.

The function **ProMechdisplacementpinconstrAlloc()** allocates memory for the pin constraint data structure.

The function **ProMechdisplacementdataPinconstrGet()** returns the pin constraint data structure. The pin constraint can have the following properties:

- **Angular**—Allows you to control the rotation about the axis of the selected cylindrical surface.

- **Axial**—Allows you to control translation along the axis of the selected cylindrical surface.

The function **ProMechdisplacementdataPinconstrSet()** sets the value for the pin constraint data structure.

The function

ProMechdisplacementdataPinangularconstrTypeGet()

returns the angular constraint type for the pin constraint. Use the function

ProMechdisplacementdataPinangularconstrTypeSet() to set the angular constraint type. Valid values are:

- **PRO_MECH_DISPLACEMENT_FREE**—Allows freedom of movement in the specified direction.
- **PRO_MECH_DISPLACEMENT_FIXED**—Constrains the entity, preventing movement in the specified direction.

Note: The angular constraint cannot be of type **PRO_MECH_DISPLACEMENT_ENFORCED**.

The function

ProMechdisplacementdataPinaxialconstrTypeGet() returns the axial constraint type for the pin constraint.

Use the method

ProMechdisplacementdataPinaxialconstrTypeSet() to set the axial constraint for the pin constraint. Valid values are:

- **PRO_MECH_DISPLACEMENT_FREE**—Allows freedom of movement in the specified direction.
- **PRO_MECH_DISPLACEMENT_FIXED**—Constrains the entity, preventing movement in the specified direction.

Note: The axial constraint cannot be of type **PRO_MECH_DISPLACEMENT_ENFORCED**.

Use the function **ProMechdisplacementpinconstrFree()** to free the memory containing the pin constraint data structure.

Example 3: Copying and Assigning a Displacement Constraint to a New Reference

This example shows how to copy a displacement constraint and assign it to a new reference.

```
#include <ProToolkit.h>
#include <ProMdl.h>
#include <ProMechItemSet.h>
#include <ProMechConstraint.h>
```

```

#include <ProUtil.h>
#include <ProMechConstraintSet.h>
#include <ProMessage.h>

typedef struct
{
    ProMechGeomrefType type;
    ProMechGeomrefSubType subtype;
    int id;
    ProAsmcomppath *path;
} PTMechExGeomRef;
/*=====*\
    FUNCTION: PTMechExGeomReferenceCreate
    PURPOSE: This function is used to allocate the Mechanica geometry
reference.
\*=====*/
ProError PTMechExGeomReferenceCreate (PTMechExGeomRef* reference ,
ProMechGeomref *geom_ref)
{
    ProMechGeomref geomref_entity;

    status = ProMechgeomrefAlloc (&geomref_entity);
    PT_TEST_LOG_SUCC ("ProMechgeomrefAlloc");

/*-----*\
Set the geomref type, subtype,id and asmcomppath
\*-----*/

    status = ProMechgeomrefTypeSet (geomref_entity, reference->type);
    PT_TEST_LOG_SUCC ("ProMechgeomrefTypeSet");

    if (reference->subtype != PRO_VALUE_UNUSED )
/* Only For Ref.: Curve, Edge, Point, Surface */
    {
        status = ProMechgeomrefSubtypeSet (geomref_entity,
reference->subtype);
        PT_TEST_LOG_SUCC ("ProMechgeomrefSubtypeSet");
    }

    if (reference->id != PRO_VALUE_UNUSED) /* For Ref. other than Part */
    {
        status = ProMechgeomrefIdSet (geomref_entity, reference->id);
        PT_TEST_LOG_SUCC ("ProMechgeomrefIdSet");
    }

    if ( reference->path != NULL) /* Only For Ref. in Assembly */
    {
        status = ProMechgeomrefPathSet (geomref_entity, reference->path);
        PT_TEST_LOG_SUCC ("ProMechgeomrefPathSet");
    }
}

```

```

    *geom_ref = geomref_entity;

    return PRO_TK_NO_ERROR;
}
/*=====*\
FUNCTION: PTMechExGeomReferenceArrayCreate
PURPOSE: This fn creates a ProArray of ProMechGeomref items and outputs
         the same.
\*=====*/

ProError PTMechExGeomReferenceArrayCreate (int num_ents, PTMechExGeomRef*
                                           array_refs,
                                           ProMechGeomref **geom_refs)
{
    ProMechGeomref *array_geomrefs;
    int i;

    if ( num_ents == 0 || array_refs == NULL)
        return PRO_TK_BAD_INPUTS;
    else
    {
/*-----*\
Create array of geomref
\*-----*/
        status = ProArrayAlloc ( num_ents, sizeof (ProMechGeomref), 1,
                                (ProArray *) &array_geomrefs);
        PT_TEST_LOG_SUCC ("ProArrayAlloc");

        for (i=0; i<num_ents; i++)
        {
            ProMechGeomref geomref_entity;

            status = PTMechExGeomReferenceCreate ( &array_refs[i],
                                                  &geomref_entity);
            PT_TEST_LOG_SUCC ("PTMechExGeomReferenceCreate");

            array_geomrefs[i] = geomref_entity;
        }
        *geom_refs = array_geomrefs;
    }

    return PRO_TK_NO_ERROR;
}

/*=====*\
FUNCTION: PTMechExDispcnstrPostfilter
PURPOSE: Selection filter to ensure that only displacement cnstrs are
         selected
\*=====*/
static ProError PTMechExDispcnstrPostfilter (ProSelection selection,
                                             ProAppData app_data)

```

```

{
    ProModelitem mech_item;
    ProMechConstrType constr_type;

    status = ProSelectionModelitemGet (selection,
                                       (ProModelitem*)&mech_item);
    PT_TEST_LOG_SUCC ("ProSelectionModelitemGet()");

    if (mech_item.type != PRO_SIMULATION_CONSTRAINT)
    return (PRO_TK_CONTINUE);
    status = ProMechconstrTypeGet (&mech_item, &constr_type);
    PT_TEST_LOG_SUCC ("ProMechconstrTypeGet()");

/*-----*\
    Check if the selection is of valid type
\*-----*/

    if ((constr_type != PRO_MECH_CONSTR_DISPLACEMENT) ||
        ( status != PRO_TK_NO_ERROR))
    return (PRO_TK_CONTINUE);

return (PRO_TK_NO_ERROR);
}
/*=====*\
    FUNCTION: PTMechExConstrCopy
    PURPOSE: Copies the displacement data from one ref to another
\*=====*/
ProError PTMechExConstrCopy()
{
    ProSelFunctions sel_funcs;
    ProMechConstrType constr_type;
    ProMechDisplacementType type;
    ProMechDisplacementConstr* trans_datas;
    ProMechDisplacementConstr* rot_datas;
    ProMechGeomref csys_data;
    int i,n_sels;
    ProSelection* sels;
    ProMechDisplacementData data;
    ProLine buffer;
    char line [PRO_LINE_SIZE];
    PTMechExGeomRef *array_refs;
    ProMechGeomrefType reftype;
    ProMechGeomrefSubType refsubtype;
    ProModelitem mech_item,model_item;
    ProMechItem tk_constraint_item;
    ProMechGeomref *array_geomrefs;
    ProMechStatus mech_status;
    ProCharLine constrset_name;
    ProMdl constraint_part;

```

```

sel_funcs.pre_filter = NULL;
sel_funcs.post_filter = PTMechExDispconstrPostfilter;
sel_funcs.post_select = NULL;
sel_funcs.app_data = NULL;

status = ProMdlCurrentGet(&constraint_part);
PT_TEST_LOG_SUCC ("ProMdlCurrentGet");

/*-----*\
  Select the displacement constraint to get the data
  \*-----*/

status = ProMessageDisplay(msgfile,
                          "PTMechEx Select the displacement constraint");
PT_TEST_LOG_SUCC ("ProMessageDisplay()");

while(1)
{
    status =ProSelect ("sim_load",
                      1,
                      NULL,
                      &sel_funcs,
                      NULL,
                      NULL,
                      &sels,
                      &n_sels);
    PT_TEST_LOG_SUCC ("ProSelect()");

    if( status != PRO_TK_NO_ERROR || n_sels <= 0 )
        break;

    status = ProSelectionModelitemGet (sels[0], &mech_item);
    PT_TEST_LOG_SUCC ("ProSelectionModelitemGet()");

    status = ProModelitemInit (mech_item.owner,
                              mech_item.id,
                              mech_item.type,
                              &mech_item);
    PT_TEST_LOG_SUCC ("ProModelitemInit()");

/*-----*\
  Get displacement data from the mech item
  \*-----*/

    status = ProMechconstrDisplacementdataGet (&mech_item, &data);
    PT_TEST_LOG_SUCC ("ProMechconstrDisplacementdataGet()");

    status = ProMechdisplacementdataTypeGet (data, &type);
    PT_TEST_LOG_SUCC ("ProMechdisplacementdataTypeGet()");

```

```

/*-----*\
Create new mech item
\*-----*/

    status = ProMechitemCreate ( (ProSolid) constraint_part,
                                PRO_SIMULATION_CONSTRAINT,
                                &tk_constraint_item);
    PT_TEST_LOG_SUCC ("ProMechitemCreate");

    status = ProMechitemNameSet ( &tk_constraint_item,
                                L"tk_created_constr");
    PT_TEST_LOG_SUCC ("ProMechitemNameSet");

/*-----*\
Select the new references
\*-----*/

    status = ProMessageDisplay(msgfile, "PTMechEx Select references");
    PT_TEST_LOG_SUCC("ProMessageDisplay");

    status = ProSelect("surface,edge,curve,point",-1, NULL, NULL, NULL,
                      NULL, &sels, &n_sels);
    PT_TEST_LOG_SUCC ("ProSelect()");

    if( status != PRO_TK_NO_ERROR || n_sels <= 0 )
        break;

    status =ProArrayAlloc (n_sels,sizeof (PTMechExGeomRef),1,
                          (ProArray *) &array_refs);
    PT_TEST_LOG_SUCC ("ProArrayAlloc");

/*-----*\
Populate the new refereces with type/subtype
\*-----*/
    for ( i =0 ; i<n_sels; i++)
    {
        ProSelectionModelitemGet(sels[i],&model_item);

        switch (model_item.type)
        {
            case PRO_EDGE :
                reftype = PRO_MECH_EDGE;
                refsubtype =PRO_MECH_CURVE_NORMAL;
                break;

            case PRO_CURVE :
                reftype = PRO_MECH_CURVE;
                refsubtype = PRO_MECH_CURVE_NORMAL;
                break;
        }
    }

```

```

case PRO_POINT :
    reftype = PRO_MECH_POINT;
    refsubtype = PRO_MECH_POINT_SINGLE;
    break;

case PRO_SURFACE :
    reftype = PRO_MECH_SURFACE;
    refsubtype = PRO_MECH_SURFACE_NORMAL;
    break;
}
array_refs[i].type = reftype;
array_refs[i].subtype = refsubtype;
array_refs[i].id = model_item.id;
array_refs[i].path = NULL;
}
/*-----*\
Create array of the geom refs from the array of refs
\*-----*/

    status = PTMechExGeomReferenceArrayCreate ( n_sels, array_refs,
                                                &array_geomrefs);
    PT_TEST_LOG_SUCC ("PTMechExGeomRefereceArrayCreate");

/*-----*\
Set the array of geom ref to the newly created constraint
\*-----*/

    status = ProMechconstrReferencesSet
        (&tk_constraint_item,array_geomrefs);
    PT_TEST_LOG_SUCC ("ProMechconstrReferencesGet()");

/*-----*\
Set the displacement data to the newly created constraint
\*-----*/

    status = ProMechconstrDisplacementdataSet
        (&tk_constraint_item,data);
    PT_TEST_LOG_SUCC ("ProMechconstrDisplacementdataSet");

/*-----*\
Get the name of the constraint set from where the displacement
conatrsint selected
\*-----*/

    status = PTMechExConstraintsetNameGet(&mech_item,constrset_name);
    PT_TEST_LOG_SUCC ("PTMechExConstraintsetNameGet");

/*-----*\
Add the newly created constraint in to the constraint set
\*-----*/

```

```

status = PTMechExConstraintAddinConstraintset
        (constraint_part, constrset_name, &tk_constraint_item);
PT_TEST_LOG_SUCC ("PTMechExConstraintAddinConstraintset");

status = ProMechitemStatusGet (&tk_constraint_item, &mech_status);
PT_TEST_LOG_SUCC ("ProMechitemStatusGet");

/*-----*\
Print Message from the result
\*-----*/

if (mech_status == PRO_MECH_ACTIVE)
{
    status = ProMessageDisplay(msgfile,
        "PTMechEx New displacement constraint created successfully");
    PT_TEST_LOG_SUCC("ProMessageDisplay()");
}
else
{
    status = ProMessageDisplay(msgfile, "PTMechEx New displacement
        constraint creation failed", msgbuf);
    PT_TEST_LOG_SUCC("ProMessageDisplay()");
}

status = ProMechitemUpdateComplete (&tk_constraint_item);
PT_TEST_LOG_SUCC ("ProMechitemUpdateComplete");

}

return PRO_TK_NO_ERROR;
}

```

Symmetry Constraints

A symmetry constraint allows you to analyze a section of a symmetric model that simulates the behavior of the whole part or assembly to which it belongs.

The functions described in this section provide read and write access to the data and contents of the Mechanics symmetry constraints.

Functions Introduced:

- **ProMechsymmetrydataAlloc()**
- **ProMechsymmetrydataTypeGet()**
- **ProMechsymmetrydataTypeSet()**
- **ProMechsymmetrydataAxisGet()**
- **ProMechsymmetrydataAxisSet()**
- **ProMechsymmetrydataSide1Get()**
- **ProMechsymmetrydataSide1Set()**
- **ProMechsymmetrydataSide2Get()**
- **ProMechsymmetrydataSide2Set()**
- **ProMechsymmetrydataFree()**

The function **ProMechsymmetrydataAlloc()** allocates the memory for the symmetry constraint data handle.

The function **ProMechsymmetrydataTypeGet()** returns the type of symmetry constraint. The argument type can have the following values:

- **PRO_MECH_SYMMETRY_CYCLIC**—Specifies a cyclic symmetry
- **PRO_MECH_SYMMETRY_MIRROR**—Specifies a mirror symmetry

Use the function **ProMechsymmetrydataTypeSet()** to set the type of symmetry constraint.

The function **ProMechsymmetrydataAxisGet()** returns the axis of symmetry of the section. Use the function **ProMechsymmetrydataAxisSet()** to set the axis of symmetry.

The function **ProMechsymmetrydataSide1Get()** returns the first side of the cut section. Use the function **ProMechsymmetrydataSide1Set()** to set the geometric references of the first side.

The function **ProMechsymmetrydataSide2Get()** returns the second side of the cut section. Use the function **ProMechsymmetrydataSide2Set()** to set the geometric references of the second side.

The function **ProMechsymmetrydataFree()** releases the memory assigned to the symmetry constraint data handle.

Mechanica Constraint Sets

A constraint set is a collection of constraints that act together, and at the same time, on the model. Constraint sets do not contain loads.

The functions referring to the Mechanica constraint sets use the structure `ProModelItem`. Constraint sets use the `ProType` field in the `ProMechitem` structure as `PRO_SIMULATION_CONSTR_SET`.

Functions Introduced:

- **ProMechconstrsetTypeGet()**
- **ProMechconstrsetDescriptionGet()**
- **ProMechconstrsetDescriptionSet()**
- **ProMechconstrsetConstrsGet()**
- **ProMechconstrConstrsetAssign()**

The function **ProMechconstrsetTypeGet()** returns the type of the constraint sets that are applied to the model. The types are as follows:

- `PRO_MECH_LOADSET_STRUCTURAL`—Specifies a structural constraint set.
- `PRO_MECH_LOADSET_THERMAL`—Specifies a thermal constraint set.

The function **ProMechconstrsetDescriptionGet()** returns the description of the constraint set. The function **ProMechconstrsetDescriptionSet()** enables you to change the description of the constraint set.

The function **ProMechconstrsetConstrsGet()** returns the constraints specified in the constraint set.

Use the function **ProMechconstrConstrsetAssign()** to assign a constraint to a constraint set.

Mechanica Beams

A beam is a one-dimensional idealization that, in three dimensions, represents a structure whose length is much greater than its other two dimensions.

The functions described in this section provide read and write access to the data and contents of Mechanics beam items. Beams use the *ProType* field in the *ProMechitem* structure as `PRO_SIMULATION_BEAM`.

Functions Introduced:

- **ProMechbeamdataAlloc()**
- **ProMechbeamTypeGet()**
- **ProMechbeamMaterialIdGet()**
- **ProMechbeamMaterialIdSet()**
- **ProMechbeamReferencesGet()**
- **ProMechbeamReferencesSet()**
- **ProMechbeamBeamdataGet()**
- **ProMechbeamBeamdataSet()**
- **ProMechbeamTrussdataGet()**
- **ProMechbeamTrussdataSet()**
- **ProMechbeamsideAlloc()**
- **ProMechbeamdataCompcurvrelGet()**
- **ProMechbeamdataCompcurvrelSet()**
- **ProMechbeamdataSidesGet()**
- **ProMechbeamdataSidesSet()**
- **ProMechbeamsideOrientGet()**
- **ProMechbeamsideOrientSet()**
- **ProMechbeamsideReleaseGet()**
- **ProMechbeamsideReleaseSet()**
- **ProMechbeamsideSectionGet()**
- **ProMechbeamsideSectionSet()**
- **ProMechbeamsideFree()**
- **ProMechbeamsideProarrayFree()**
- **ProMechbeamdataStressrecoveryGet()**
- **ProMechbeamdataStressrecoverySet()**
- **ProMechbeamdataXyshearGet()**
- **ProMechbeamdataXyshearSet()**

- **ProMechbeamdataXzshearGet()**
- **ProMechbeamdataXzshearSet()**
- **ProMechbeamdataYdirectionGet()**
- **ProMechbeamdataYdirectionSet()**
- **ProMechbeamdataFree()**
- **ProMechtrussdataAlloc()**
- **ProMechtrussdataSectionGet()**
- **ProMechtrussdataSectionSet()**
- **ProMechtrussdataFree()**

The function **ProMechbeamdataAlloc()** allocates the memory for the beam data handle.

The function **ProMechbeamTypeGet()** returns the types of beams.

The different types of beams are as follows:

- **PRO_MECH_BEAM_BEAM**—Specifies a beam. Use the function **ProMechbeamBeamdataGet()** to access the data structure for the beam. Use the function **ProMechbeamBeamdataSet()** to modify the beam.
- **PROMECH_BEAM_TRUSS**—Specifies a truss. Use the function **ProMechbeamTrussdataGet()** to access the data structure for the truss. Use the function **ProMechbeamTrussdataSet()** to modify the truss.

The function **ProMechbeamMaterialIdGet()** returns the ID of the Pro/ENGINEER material used to create the beam. Use the function **ProMechbeamMaterialIdSet()** to set the material ID for the beam.

The function **ProMechbeamReferencesGet()** returns the start and end points of the beam or a curve or surface of the beam. Use the function **ProMechbeamReferencesSet()** to set the beam references.

The function **ProMechbeamsideAlloc()** allocates the memory for the beam side data handle.

The function **ProMechbeamdataCompervrelGet()** returns a boolean that indicates whether the beam releases are applied only to the ends of the selected composite curve or to the ends of the individual curves. Use the function

ProMechbeamdataCompervrelSet() to set whether the beam releases are to be applied only to the ends of the selected composite curve or to the ends of the individual curves.

The function **ProMechbeamdataSidesGet()** returns a structure containing the cross section properties for beam elements, the orientation properties for angle and offsets, and the degrees of freedom at each beam end. Use the function

ProMechbeamdataSidesSet() to set the cross section properties and orientation properties of the beam data.

The function **ProMechbeamsideOrientGet()** returns the orientation ID for the specified beam elements. Use the function **ProMechbeamsideOrientSet()** to set the orientation ID.

The function **ProMechbeamsideReleaseGet()** returns the beam release ID for the specified beam elements. Use the function **ProMechbeamsideReleaseSet()** to set the beam release ID.

The function **ProMechbeamsideSectionGet()** returns the cross section ID for the specified beam elements. Use the function **ProMechbeamsideSectionSet()** to set the cross section ID.

The function **ProMechbeamsideFree()** releases the memory assigned to the beam side handle.

The function **ProMechbeamsideProarrayFree()** releases the memory assigned to a *ProArray* of beam side handles.

The function **ProMechbeamdataStressrecoveryGet()** returns a boolean that indicates whether stress recovery is included while creating the beam. If this option is specified as *true*, you can access the beam stress at a specific location on the beam section. Use the function **ProMechbeamdataStressrecoverySet()** to set the value of the boolean.

The function **ProMechbeamdataXyshearGet()** returns the shear relief coefficient due to taper for the XY planes. Use the function **ProMechbeamdataXyshearSet()** to set the shear relief coefficient relative to the XY plane.

The function **ProMechbeamdataXzshearGet()** returns the shear relief coefficient due to taper for the XZ planes. Use the function **ProMechbeamdataXzshearSet()** to set the shear relief coefficient relative to the XZ plane.

The function **ProMechbeamdataYdirectionGet()** returns the orientation of the XY plane of a beam by defining its Y-direction. Use the function **ProMechbeamdataYdirectionSet()** to set the Y-direction of the beam.

The function **ProMechbeamdataFree()** releases the memory assigned to the beam data handle.

The function **ProMechtrussdataAlloc()** allocates the memory for the truss data handle.

The function **ProMechtrussdataSectionGet()** returns the cross section ID for the specified truss elements. Use the function **ProMechtrussdataSectionSet()** to set the beam section ID for the truss data.

The function **ProMechtrussdataFree()** releases the memory assigned to the truss data handle.

Mechanica Beam Sections

The functions described in this section provide read and write access to the data and contents of the Mechanica beam section items. Beam sections use the ProType field in the ProMechitem structure as PRO_SIMULATION_BEAM_SECTION.

Functions Introduced:

- **ProMechsectiondataAlloc()**
- **ProMechsectiondataSectiontypeGet()**
- **ProMechbeamsectionDataGet()**
- **ProMechbeamsectionDataSet()**
- **ProMechbeamsectionDescriptionGet()**
- **ProMechbeamsectionDescriptionSet()**
- **ProMechsectiondataNonstructcogGet()**
- **ProMechsectiondataNonstructcogSet()**
- **ProMechsectiondataNonstructmassGet()**
- **ProMechsectiondataNonstructmassSet()**
- **ProMechsectiondataNonstructmomentGet()**
- **ProMechsectiondataNonstructmomentSet()**
- **ProMechsectiondataWarpcoeffGet()**

- **ProMechsectiondataWarpcoeffSet()**
- **ProMechsectiondataChannelsectdataGet()**
- **ProMechsectiondataChannelsectdataSet()**
- **ProMechsectiondataCirclesectdataGet()**
- **ProMechsectiondataCirclesectdataSet()**
- **ProMechsectiondataDiamondsectdataGet()**
- **ProMechsectiondataDiamondsectdataSet()**
- **ProMechsectiondataEllipsesectdataGet()**
- **ProMechsectiondataEllipsesectdataSet()**
- **ProMechsectiondataGeneralsectdataGet()**
- **ProMechsectiondataGeneralsectdataSet()**
- **ProMechsectiondataHollowcirclesectdataGet()**
- **ProMechsectiondataHollowcirclesectdataSet()**
- **ProMechsectiondataHollowellipsesectdataGet()**
- **ProMechsectiondataHollowellipsesectdataSet()**
- **ProMechsectiondataHollowrectsectdataGet()**
- **ProMechsectiondataHollowrectsectdataSet()**
- **ProMechsectiondataIbeamsectdataGet()**
- **ProMechsectiondataIbeamsectdataSet()**
- **ProMechsectiondataLsectionsectdataGet()**
- **ProMechsectiondataLsectionsectdataSet()**
- **ProMechsectiondataRectanglesectdataGet()**
- **ProMechsectiondataRectanglesectdataSet()**
- **ProMechsectiondataSketchedsectdataGet()**
- **ProMechsectiondataSketchedsectdataSet()**
- **ProMechsectiondataSquaresectdataGet()**
- **ProMechsectiondataSquaresectdataSet()**
- **ProMechsectiondataFree()**

The function **ProMechsectiondataAlloc()** allocates the memory for the beam section data handle.

The function **ProMechbeamsectionDataGet()** provides access to the data structure containing the properties of the beam section data. Use the function **ProMechbeamsectionDataSet()** to set the properties of the beam section data.

The function **ProMechsectiondataSectiontypeGet()** returns the types of beam sections.

The types of beam sections are as follows:

- **PRO_MECH_BEAM_SECTION_SKETCHED**—Specifies a cross section created using either the sketch solid beam or sketch thin beam type. Use the function **ProMechsectiondataSketchedsectdataGet()** to access the sketched beam data structure. Use the function **ProMechsectiondataSketchedsectdataSet()** to set the properties of the sketched beam data structure.
- **PRO_MECH_BEAM_SECTION_SQUARE**—Specifies a square beam section type. The cross section dimension is specified by the length of the sides of the square. Use the function **ProMechsectiondataSquaresectdataGet()** to access the square section data structure. Use the function **ProMechsectiondataSquaresectdataSet()** to set the properties of the square section data structure.
- **PRO_MECH_BEAM_SECTION_RECTANGLE**—Specifies a rectangular beam section type. The cross section dimension is specified by the height and width of the rectangle. Use the function **ProMechsectiondataRectanglesectdataGet()** to access the rectangular section data structure. Use the function **ProMechsectiondataRectanglesectdataSet()** to access the rectangular section data structure.
- **PRO_MECH_BEAM_SECTION_HOLLOW_RECTANGLE**—Specifies a hollow rectangular beam section type. The cross section dimension for this beam type is specified by the Outer height and width, and the inner height and width of the rectangle. Use the function **ProMechsectiondataHollowrectsectdataGet()** to access the hollow section data structure. Use the function **ProMechsectiondataHollowrectsectdataSet()** to access the hollow section data structure.

- **PRO_MECH_BEAM_SECTION_CHANNEL**—Specifies a channel beam section type. The cross section dimension is specified by the flange width, flange thickness, web height, and web thickness. Use the function **ProMechsectiondataChannelsectdataGet()** to access the channel section data structure. Use the function **ProMechsectiondataChannelsectdataSet()** to access the channel section data structure.
- **PRO_MECH_BEAM_SECTION_I_BEAM**—Specifies an I-beam section type. The cross section dimension for this beam type is specified by the flange width, flange thickness, web height, and web thickness. Use the function **ProMechsectiondataIbeamsectdataGet()** to access the I-beam section data structure. Use the function **ProMechsectiondataIbeamsectdataSet()** to access the I-beam section data structure.
- **PRO_MECH_BEAM_SECTION_L_SECTION**—Specifies an L-section beam section type. The cross section dimension for this beam type is specified by the overall flange width, flange thickness, web height, and web thickness. Use the function **ProMechsectiondataLsectionsectdataGet()** to access the L-section data structure. Use the function **ProMechsectiondataLsectionsectdataSet()** to access the L-section data structure.
- **PRO_MECH_BEAM_SECTION_DIAMOND**—Specifies a diamond beam section type. The cross section dimension is specified by the width and height of the sides. Use the function **ProMechsectiondataDiamondsectdataGet()** to access the diamond beam section data structure. Use the function **ProMechsectiondataDiamondsectdataSet()** to modify the diamond beam section data structure.
- **PRO_MECH_BEAM_SECTION_SOLID_CIRCLE**—Specifies a solid circle beam section type. The cross section dimension for this beam type is specified by the radius of the circular beam cross-section. Use the function **ProMechsectiondataCirclesectdataGet()** to access the solid circle beam section data structure. Use the function **ProMechsectiondataCirclesectdataSet()** to modify the solid circle beam section data structure.

- **PRO_MECH_BEAM_SECTION_HOLLOW_CIRCLE**—Specifies a hollow circle beam section type. The cross section dimension for this beam type is specified by the outside radius and the inside radius of the hollow beam cross-section. Use the function **ProMechsectiondataHollowcirclesectdataGet()** to access the hollow circle beam section data structure. Use the function **ProMechsectiondataHollowcirclesectdataSet()** to modify the hollow circle beam section data structure.
- **PRO_MECH_BEAM_SECTION_SOLID_ELLIPSE**—Specifies a solid ellipse beam section type. The cross section dimension for this beam type is specified by the length of the major axis and the length of the minor axis. Use the function **ProMechsectiondataEllipsesectdataGet()** to access the solid ellipse beam data structure. Use the function **ProMechsectiondataEllipsesectdataSet()** to modify the solid ellipse beam data structure.
- **PRO_MECH_BEAM_SECTION_HOLLOW_ELLIPSE**—Specifies a hollow ellipse beam section type. The cross section dimension for this beam type is specified by the length of the major axis, length of the minor axis, and the inside major axis. Use the function **ProMechsectiondataHollowellipsesectdataGet()** to access the hollow ellipse beam data structure. Use the function **ProMechsectiondataHollowellipsesectdataSet()** to access the hollow ellipse beam data structure.
- **PRO_MECH_BEAM_SECTION_GENERAL**—Specifies a general beam section type. A general beam section type does not have a predefined shape. Use the function **ProMechsectiondataGeneralsectdataGet()** to access the general beam data structure. Use the function **ProMechsectiondataGeneralsectdataSet()** to access the general beam data structure.

The function **ProMechbeamsectionDescriptionGet()** returns the description for the specified beam section. Use the function **ProMechbeamsectionDescriptionSet()** to set the description of the specified beam section.

The function **ProMechsectiondataNonstructcogGet()** returns the Y and Z coordinates of the non-structural mass center of gravity. A non-structural mass is a mass that responds to gravity, but does not strengthen the structure. Non-structural masses can have different moments of inertia and gravitational centers than the specified beam. The function **ProMechsectiondataNonstructcogSet()** sets the Y and Z coordinates of the non-structural mass center of gravity.

The function **ProMechsectiondataNonstructmassGet()** returns the non-structural mass per unit length. Use the function **ProMechsectiondataNonstructmassSet()** to set the non-structural mass per unit length the beam section.

The function **ProMechsectiondataNonstructmomentGet()** returns the non-structural mass moment of inertia per unit length. Use the function **ProMechsectiondataNonstructmomentSet()** to set the non-structural mass moment of inertia per unit length of the beam section.

The function **ProMechsectiondataWarpcoeffGet()** returns the warp coefficient of the beam section. Use the function **ProMechsectiondataWarpcoeffSet()** to set the warp coefficient of the beam section.

The function **ProMechsectiondataFree()** releases the memory assigned to the beam section data.

Sketched Beam Section

The functions described in this section provide read and write access to properties of the sketched beam section.

Functions Introduced:

- **ProMechsketchedsctndataAlloc()**
- **ProMechsketchedsctndataFeatureidGet()**
- **ProMechsketchedsctndataOrienttypeGet()**
- **ProMechsketchedsctndataOrienttypeSet()**
- **ProMechsketchedsctndataShearcenterGet()**
- **ProMechsketchedsctndataShearcenterSet()**
- **ProMechsketchedsctndataFree()**

The function **ProMechsketchedsctndataAlloc()** allocates the memory for the sketched beam section data handle.

The function **ProMechsketchedsctndataFeatureidGet()** returns the sketch feature id of the sketched beam section.

The function **ProMechsketchedsctndataOrienttypeGet()** returns the type of orientation for the sketched beam section. The beam orientation defines the Y direction of the beam, that is, how it rotates on the XY plane. The types of orientation is as follows:

- **PRO_MECH_BEAM_SECTION_SKET_XY_AS_YZ**—The beam section X and Y coordinates correspond to the beam Y and Z directions, respectively.
- **PRO_MECH_BEAM_SECTION_SKET_XY_AS_ZY**—The beam section X and Y coordinates correspond to the beam Z and Y directions, respectively.

Use the function **ProMechsketchedsectndataOrienttypeSet()** to set the type of orientation for the sketched beam section.

The function **ProMechsketchedsectndataShearcenterGet()** returns the shear center of the beam section. The shear center is the point on a beam section about which the section rotates under deflection. The function

ProMechsketchedsectndataShearcenterSet() sets the shear center of the beam section.

The function **ProMechsketchedsectndataFree()** releases the memory assigned to sketched beam section data.

General Beam Section

The functions described in this section provide read and write access to properties of the general beam section.

Functions Introduced:

- **ProMechgeneralsectndataAlloc()**
- **ProMechgeneralsectndataAreaGet()**
- **ProMechgeneralsectndataAreaSet()**
- **ProMechgeneralsectndataAreaproductGet()**
- **ProMechgeneralsectndataAreaproductSet()**
- **ProMechgeneralsectndataMomentsGet()**
- **ProMechgeneralsectndataMomentsSet()**
- **ProMechgeneralsectndataShearcenterGet()**
- **ProMechgeneralsectndataShearcenterSet()**
- **ProMechgeneralsectndataShearfactorGet()**
- **ProMechgeneralsectndataShearfactorSet()**
- **ProMechgeneralsectndataTorsionstiffnessGet()**
- **ProMechgeneralsectndataTorsionstiffnessSet()**
- **ProMechgeneralsectndataStressrecoverypntsGet()**
- **ProMechgeneralsectndataStressrecoverypntsSet()**
- **ProMechgeneralsectndataFree()**

The function **ProMechgeneralsectndataAlloc()** allocates the memory for the general beam section data handle.

The function **ProMechgeneralsectndataAreaGet()** returns the cross-sectional area for each beam section. Use the function **ProMechgeneralsectndataAreaSet()** to set the cross-sectional area for the beam section.

The function **ProMechgeneralsectndataAreaproductGet()** returns the area product of the moments of inertia. Use the function **ProMechgeneralsectndataAreaproductSet()** to set the area product of inertia.

The function **ProMechgeneralsectndataMomentsGet()** returns the second moments of area for each beam section. These properties describe the stiffness in bending about a beam's principle Y and Z axes. Use the function **ProMechgeneralsectndataMomentsSet()** to set the second moments of area for each beam section.

The function **ProMechgeneralstndataShearcenterGet()** returns the Shear DY and Shear DZ values. These values specify the distance between shear center (the point on a beam section about which the section rotates under deflection) and the centroid of the beam section, with respect to the principal axes. Use the function **ProMechgeneralstndataShearcenterSet()** to set the Shear DY and Shear DZ values.

The function **ProMechgeneralstndataShearfactorGet()** returns the Shear FY and Shear FZ values for the beam section. These values represent the ratio of a beam's effective "shear area" to its true cross-sectional area for shear in the principal Y and Z directions. Use the function **ProMechgeneralstndataShearfactorSet()** to set the Shear FY and Shear FZ values for the beam section.

The function **ProMechgeneralstndataTorsionstiffnessGet()** returns the second polar moment of area for each beam section. This property describes the stiffness in torsion. Use the function **ProMechgeneralstndataTorsionstiffnessSet()** to set the torsion stiffness for each beam section.

The function **ProMechgeneralstndataStressrecoverypntsGet()** returns the stress recovery points of the beam section. Use the **ProMechgeneralstndataStressrecoverypntsSet()** to set the stress recovery points of the beam section.

The function **ProMechgeneralstndataFree()** releases the memory assigned to the sketched beam section data.

Beam Orientations

The functions described in this section provide read and write access to the data and contents of the Mechanics beam orientations data structure. Beam orientations use the ProType field in the ProMechitem structure as PRO_SIMULATION_BEAM_ORIENT.

Functions Introduced:

- **ProMechbeamorientDataGet()**
- **ProMechbeamorientDataSet()**
- **ProMechbeamorientDescriptionGet()**
- **ProMechbeamorientDescriptionSet()**
- **ProMechbeamorientdataTypeGet()**

- **ProMechbeamorientdataTypeSet()**
- **ProMechbeamorientdataAngleGet()**
- **ProMechbeamorientdataAngleSet()**
- **ProMechbeamorientdataVectorGet()**
- **ProMechbeamorientdataVectorSet()**
- **ProMechbeamorientdataFree()**

The function **ProMechbeamorientDataGet()** provides access to the data structure containing the properties of the beam orientation. Use the function **ProMechbeamorientDataSet()** to set the properties of the beam orientation data.

The function **ProMechbeamorientDescriptionGet()** returns the description of the beam orientation. Use the function **ProMechbeamorientDescriptionSet()** to set the description of the beam orientation.

The function **ProMechbeamorientdataTypeGet()** returns the type of the orientation specified for the Mechanica beam orientation item. The types of orientations are as follows:

- **PRO_MECH_BEAM_ORIENT_OFFSET_SHAPE_ORIGIN**—
Specifies the point of origin of the beam shape coordinate system.
- **PRO_MECH_BEAM_ORIENT_OFFSET_CENTROID**—
Specifies the origin of the principal coordinate system which is at the centroid of the section. For general sections and all standard sections, it is coincident with **PRO_MECH_BEAM_ORIENT_OFFSET_SHAPE_ORIGIN**.
- **PRO_MECH_BEAM_ORIENT_OFFSET_SHEAR_CENTER**—
Specifies the point on a beam section about which the section rotates under deflection.

Use the function **ProMechbeamorientdataTypeSet()** to set the type of orientation.

The function **ProMechbeamorientdataAngleGet()** returns the angle of the beam orientation. Use the function **ProMechbeamorientdataAngleSet()** to set the angle of the beam orientation.

The function **ProMechbeamorientdataVectorGet()** returns the direction of the beam vector. Use the function **ProMechbeamorientdataVectorSet()** to set the direction of the beam vector.

Use the function **ProMechbeamorientdataFree()** to free the memory allocated to the beam orientation data structure.

Beam Releases

The beam releases specify the degrees of freedom you want to release for a beam end or beam ends. Beam releases determine the degrees of freedom that do not participate in a connection at the end of a beam. You can specify beam releases for both straight and curved beams.

The beam releases data structure contains a combination of the six degrees of freedom relative to the beam's local axes, that is, translation in X, Y, and Z and rotation in X, Y, and Z.

The functions described in this section provide read and write access to the data and contents of the Mechanics beam releases data structure. Beam releases use the ProType field in the ProMechitem structure as PRO_SIMULATION_BEAM_RELEASE.

Functions Introduced:

- **ProMechbeamreleasedataAlloc()**
- **ProMechbeamreleaseDataGet()**
- **ProMechbeamreleaseDataSet()**
- **ProMechbeamreleaseDescriptionGet()**
- **ProMechbeamreleaseDescriptionSet()**
- **ProMechbeamreleasedataRotationflagsGet()**
- **ProMechbeamreleasedataRotationflagsSet()**
- **ProMechbeamreleasedataTranslationflagsGet()**
- **ProMechbeamreleasedataTranslationflagsSet()**
- **ProMechbeamreleasedataFree()**

The function **ProMechbeamreleasedataAlloc()** allocates the memory for the beam release data handle.

The function **ProMechbeamreleaseDataGet()** provides access to the data structure containing the properties of the beam release item.

The translation and rotation flags correspond to translation or rotation about the X, Y, and Z directions. Set the value of each flag to PRO_B_TRUE, to indicate that the beam is not constrained in the specified direction. Set the value to PRO_B_FALSE to indicate that the beam is constrained in the specified direction.

The function **ProMechbeamreleasedataFree()** releases the memory assigned to the beam release data handle.

Mechanica Spring Items

A spring connects two points or a point to the ground in the specified model. It provides the stiffness that you specify at the location on the model where you place it. The stiffness can be translational (force per unit length) or torsional (torque). The force generated by the spring is proportional to the amount of displacement that occurs. The functions described in this section enable you to access the data and contents of the Mechanica spring items. Springs use the ProType field in the ProMechitem structure as PRO_SIMULATION_SPRING.

Functions Introduced:

- **ProMechspringReferencesGet()**
- **ProMechspringReferencesSet()**
- **ProMechspringTypeGet()**
- **ProMechspringAdvanceddataGet()**
- **ProMechspringAdvanceddataSet()**
- **ProMechspringGrounddataGet()**
- **ProMechspringGrounddataSet()**
- **ProMechspringSimplifieddataGet()**
- **ProMechspringSimplifieddataSet()**
- **ProMechsinglespringdataAlloc()**
- **ProMechsinglespringdataTorsionalstiffnessGet()**
- **ProMechsinglespringdataTorsionalstiffnessSet()**
- **ProMechsinglespringdataExtensionalstiffnessGet()**
- **ProMechsinglespringdataExtensionalstiffnessSet()**
- **ProMechsinglespringdataFree()**
- **ProMechadvancedspringdataAlloc()**

- **ProMechadvancedspringdataPropertiesGet()**
- **ProMechadvancedspringdataPropertiesSet()**
- **ProMechadvancedspringdataRotationGet()**
- **ProMechadvancedspringdataRotationSet()**
- **ProMechadvancedspringdataYdirectionGet()**
- **ProMechadvancedspringdataYdirectionSet()**
- **ProMechadvancedspringdataFree()**
- **ProMechgroundspringdataAlloc()**
- **ProMechgroundspringdataCsysGet()**
- **ProMechgroundspringdataCsysSet()**
- **ProMechgroundspringdataPropertiesGet()**
- **ProMechgroundspringdataPropertiesSet()**
- **ProMechgroundspringdataFree()**

The function **ProMechspringReferencesGet()** returns the geometrical references specified while modeling the spring. The references define the location of the spring on the model. Use the function **ProMechspringReferencesSet()** to set the geometrical references for the spring.

The function **ProMechspringTypeGet()** returns the type of the specified spring. The output argument *Type* has one of the following values:

- **PRO_MECH_SPRING_SIMPLE**—Specifies a simple spring. This type of spring connects two points, two vertices, a point to an edge, a point to a surface, a point to a pattern of points, a point to a single point feature. The extensional and torsional stiffness properties will be defined for this spring.

Use the function **ProMechspringSimpledataGet()** to provide access to the data structure containing the simple spring data. Use the function **ProMechspringSimpledataSet()** to modify the data structure containing the simple spring data.

- **PRO_MECH_SPRING_GROUND**—Specifies a To Ground spring. This type of string connects a point, a single point feature, or a single pattern of points to ground. The spring stiffness properties and the orientation are defined for this spring.

A separate properties object can be defined for this type of spring.

Use the function **ProMechspringGrounddataGet()** to provide access to the data structure containing the ground spring data. Use the function

ProMechspringGrounddataSet() to modify the data structure containing the ground spring data.

- **PRO_MECH_SPRING_ADVANCED**—Specifies an Advanced spring. This type of spring connects two points, a point to an edge, a point to a surface, a point to a pattern of points, or a point to a single point feature. The stiffness properties, orientation properties, and an additional rotation are defined for this spring. Use the function **ProMechspringAdvanceddataGet()** to provide access to the data structure containing the advanced spring data. Use the function **ProMechspringAdvanceddataSet()** to modify the data structure containing the advanced spring data.

Mechanica Spring Property Items

The functions described in this section provide access to the data and contents of the Mechanica spring property items. Spring properties use the ProType field in the ProMechitem structure as PRO_SIMULATION_SPRING_PROPS.

Functions Introduced:

- **ProMechspringpropsdataAlloc()**
- **ProMechspringpropsDataGet()**
- **ProMechspringpropsDataSet()**
- **ProMechspringpropsDescriptionGet()**
- **ProMechspringpropsDescriptionSet()**
- **ProMechspringpropsdataDampingcoefficientsGet()**
- **ProMechspringpropsdataDampingcoefficientsSet()**
- **ProMechspringpropsdataExtensionalcoefficientsGet()**
- **ProMechspringpropsdataExtensionalcoefficientsSet()**
- **ProMechspringpropsdataTorsionalcoefficientsGet()**
- **ProMechspringpropsdataTorsionalcoefficientsSet()**
- **ProMechspringpropsdataCouplingcoefficientsGet()**
- **ProMechspringpropsdataCouplingcoefficientsSet()**
- **ProMechspringpropsdataAutocouplingGet()**

- **ProMechspringpropsdataAutocouplingSet()**
- **ProMechspringpropsdataFree()**

The function **ProMechspringpropsDataGet()** provides access to the data structure containing the Spring Properties data.

The functions listed above provide read and write access to the definition of the spring properties. You can access the name, description, extensional and torsional stiffness, and the damping coefficient for the spring properties.

Mechanica Mass Items

A mass is an idealization that you can use to represent a concentrated mass without a specified shape. The mass of an object determines how that object resists translation and rotation. You can also add mass that is distributed over features such as curves, edges, or surfaces.

The functions described in this section provide access to the data and contents of the Mechanica mass items. Masses use the ProType field in the ProMechitem structure as PRO_SIMULATION_MASS.

Functions Introduced:

- **ProMechmassDistributionGet()**
- **ProMechmassDistributionSet()**
- **ProMechmassReferencesGet()**
- **ProMechmassReferencesSet()**
- **ProMechmassTypeGet()**
- **ProMechmassSimpledataGet()**
- **ProMechmassSimpledataSet()**
- **ProMechmassAdvanceddataGet()**
- **ProMechmassAdvanceddataSet()**
- **ProMechmassComponentdataGet()**
- **ProMechmassComponentpointdataSet()**
- **ProMechmassComponentdistributeddataSet()**
- **ProMechsimplemassdataAlloc()**
- **ProMechsimplemassdataMassGet()**
- **ProMechsimplemassdataMassSet()**

- **ProMechsimplemassdataFree()**
- **ProMechadvancedmassdataAlloc()**
- **ProMechadvancedmassdataCsysGet()**
- **ProMechadvancedmassdataCsysSet()**
- **ProMechadvancedmassdataPropertiesGet()**
- **ProMechadvancedmassdataPropertiesSet()**
- **ProMechadvancedmassdataFree()**
- **ProMechcomponentmassdataAlloc()**
- **ProMechcomponentmassdataComponentGet()**
- **ProMechcomponentmassdataComponentSet()**
- **ProMechcomponentmassdataFree()**

The function **ProMechmassDistributionGet()** returns the types of masses that can be applied to curves, surfaces, and edges in FEM Mode.

The output argument *type* can have the following values:

- **PRO_MECH_MASS_DISTR_AT_POINT**—Specifies that the mass is added to a point, vertex, multiple single points, point features, and point patterns.
- **PRO_MECH_MASS_DISTR_TOTAL**—Specifies the total distribution of mass along a surface or curve.
- **PRO_MECH_MASS_DISTR_PER_UNIT**—Specifies the distribution of mass along a curve or surface per unit length or per unit area respectively.

Use the function **ProMechmassDistributionSet()** to set the types of masses that can be applied to curves, surfaces, and edges in FEM Mode.

The function **ProMechmassReferencesGet()** returns the mass reference objects. The references can be either curves, edges, or surfaces. Use the function **ProMechmassReferencesSet()** to set the mass reference objects.

The function **ProMechmassTypeGet()** returns the types of masses defined for a point or vertex. The output argument *type* can have one of the following values:

- **PRO_MECH_MASS_SIMPLE**—Specifies a simple mass type. Specify an integer as the mass value and points, single point, multiple single points, point features, point patterns, edges, curves, or surfaces as the reference for the simple mass type. Use the function **ProMechmassSimpledataGet()** to access the simple mass data structure. Use the function **ProMechmassSimpledataSet()** to modify the simple mass data structure.
- **PRO_MECH_MASS_ADVANCED**—Specifies the advanced mass type. Specify the coordinate system and the mass property object for a single point, multiple single points, point features, and point patterns. Use the function **ProMechmassAdvanceddataGet()** to access the advanced mass data structure. Use the function **ProMechmassAdvanceddataSet()** to modify the advanced mass data structure.
- **PRO_MECH_MASS_COMP_AT_POINT**—Specifies the component mass data for a part or subassembly of an assembly. For this type of mass, the mass definition is specified using the component's mass, moment of inertia, and center of gravity. This mass type can be created using points, edges or curves, or Surfaces as the reference. Use the function **ProMechmassComponentdataGet()** to access the component mass data structure. Use the function **ProMechmassComponentpointdataSet()** to modify the component at point mass data structure. This mass type is applicable only for the assembly mode.
- **PRO_MECH_MASS_COMP_DISTRIBUTED**—Specifies the component distributed mass data for a part or subassembly of an assembly. For this type of mass, only the component's mass is used to specify the mass definition. This mass type can be created using points, edges or curves, or surfaces as the reference. Use the function **ProMechmassComponentdataGet()** to access the component mass data structure. Use the function **ProMechmassComponentdistributeddataSet()** to modify the component distributed mass data structure. This mass type is applicable only for the assembly mode.

The function **ProMechsimplemassdataAlloc()** allocates memory for a simple mass data structure.

The function **ProMechsimplemassdataMassGet()** returns the value of the mass for a simple mass data and the function **ProMechsimplemassdataMassSet()** sets the value of the mass for a simple mass data.

The function **ProMechadvancedmassdataAlloc()** allocates memory for the advanced mass data structure.

The function **ProMechadvancedmassdataCsysGet()** returns the reference co-ordinate system for the advanced mass data and the function **ProMechadvancedmassdataCsysSet()** sets the reference co-ordinate system for the advanced mass data.

The function **ProMechadvancedmassdataPropertiesGet()** returns the mass property for the advanced mass data.

The function **ProMechadvancedmassdataPropertiesSet()** sets the mass property for the advanced mass data.

The function **ProMechcomponentmassdataAlloc()** allocates memory for the component mass data structure.

The function **ProMechcomponentmassdataComponentGet()** specifies a reference for the component mass data. You can specify only one datum point or vertex as reference for the mass type `PRO_MECH_MASS_COMP_AT_POINT`. You can specify points, edges, curves, or surfaces as reference for the mass type `PRO_MECH_MASS_COMP_DISTRIBUTED`.

The function **ProMechcomponentmassdataComponentSet()** sets the reference type for the component mass data.

Mechanica Mass Properties

The functions described in this section provide read and write access to the data and contents of the Mechanica property items. Mass properties use the `ProType` field in the `ProMechitem` structure as `PRO_SIMULATION_MASS_PROPS`.

Function Introduced:

- **ProMechmasspropsDescriptionGet()**
- **ProMechmasspropsDescriptionSet()**
- **ProMechmasspropsMassGet()**
- **ProMechmasspropsMassSet()**
- **ProMechmasspropsMomentsGet()**
- **ProMechmasspropsMomentsSet()**

The function **ProMechmasspropsDescriptionGet()** returns the description for the mass property. Use the function **ProMechmasspropsmassSet()** to set the description for the mass property.

The function **ProMechmasspropsMassGet()** returns the value of the mass specified for the mass properties object. Use the function **ProMechmasspropsMassSet()** to set the value of the mass.

The function **ProMechmasspropsMomentsGet()** returns the moment of inertia about each mass element's center of gravity with respect to the axes and principal planes of the WCS. Use the function **ProMechmasspropsMomentsSet()** to set the moment of inertia.

Material Orientations

The functions described in this section specify material orientation for surfaces, volumes, shells, solids, 2D solids, and 2D plates. These functions provide read and write access to the data and contents of Mechanica material orientation objects. Material orientations use the ProType field in the ProMechitem structure as PRO_SIMULATION_MATL_ORIENT.

Functions Introduced:

- **ProMechmaterialorientdataAlloc()**
- **ProMechmaterialorientDataGet()**
- **ProMechmaterialorientDataSet()**
- **ProMechmaterialorientDescriptionGet()**
- **ProMechmaterialorientDescriptionSet()**
- **ProMechmaterialorientdataObjecttypeGet()**
- **ProMechmaterialorientdataObjecttypeSet()**
- **ProMechmaterialorientdataCsysdataGet()**
- **ProMechmaterialorientdataCsysdataSet()**
- **ProMechmaterialorientdataProjectiondataGet()**
- **ProMechmaterialorientdataProjectiondataSet()**
- **ProMechmaterialorientdataRotationGet()**
- **ProMechmaterialorientdataRotationSet()**
- **ProMechmaterialorientdataSurfacerotatationGet()**

- **ProMechmaterialorientdataSurfacerotationSet()**
- **ProMechmaterialorientdataTypeGet()**
- **ProMechmaterialorientdataSurfacerotationUnset()**
- **ProMechmaterialorientdataFirstdirectionSet()**
- **ProMechmaterialorientdataSeconddirectionSet()**
- **ProMechmaterialorientdataFree()**
- **ProMechmaterialorientcsysAlloc()**
- **ProMechmaterialorientcsysCsysGet()**
- **ProMechmaterialorientcsysCsysSet()**
- **ProMechmaterialorientcsysProjectiontypeGet()**
- **ProMechmaterialorientcsysProjectiontypeSet()**
- **ProMechmaterialorientcsysXaxisGet()**
- **ProMechmaterialorientcsysXaxisSet()**
- **ProMechmaterialorientcsysYaxisGet()**
- **ProMechmaterialorientcsysYaxisSet()**
- **ProMechmaterialorientcsysZaxisGet()**
- **ProMechmaterialorientcsysZaxisSet()**
- **ProMechmaterialorientcsysFree()**
- **ProMechmaterialorientprojAlloc()**
- **ProMechmaterialorientprojPointsGet()**
- **ProMechmaterialorientprojPointsSet()**
- **ProMechmaterialorientprojTypeGet()**
- **ProMechmaterialorientprojXyzvectorGet()**
- **ProMechmaterialorientprojXyzvectorSet()**
- **ProMechmaterialorientprojFree()**

The function **ProMechMaterialorientDataGet()** provides access to the data structure containing the properties of the material orientation item. Use the function

ProMechmaterialorientDataSet() to set the data structure containing the properties of the material orientation item.

The function **ProMechmaterialorientDescriptionGet()** returns the description of the material orientation. Use the function

ProMechmaterialorientDescriptionSet() to set the description of the material orientation.

The function **ProMachmaterialorientdataObjecttypeGet()** returns the type of object to which the material orientation is applied. The types of object are:

- PRO_MECH_MATLORI_MODEL—Specifies a model.
- PRO_MECH_MATLORI_SURFACE—Specifies a surface.

Use the function **ProMachmaterialorientdataObjecttypeSet()** to set the type of object.

The function **ProMachmaterialorientReferencesGet()** returns the geometric references specified for the material orientation object.

The function **ProMachmaterialorientdataCsysdataGet()** returns a handle to the data structure containing the coordinate system data for the material orientation. Use the function **ProMachmaterialorientdataCsysdataSet()** to set the coordinate system data for the material orientation.

The function **ProMachmaterialorientdataTypeGet()** returns the type of the material direction. The types are as follows:

- PRO_MECH_MATLORI_COORD_SYSTEM—Specifies that the material orientation direction is determined by the reference coordinate system.
- PRO_MECH_MATLORI_1_DIR—Specifies that the material orientation direction is determined by the first parametric direction of the material. Use the function **ProMachmaterialorientdataFirstdirectionSet()** to set the first direction of the material as the orientation type.
- PRO_MECH_MATLORI_2_DIR—Specifies that the material orientation direction is determined by the second parametric direction of the material. Use the function **ProMachmaterialorientdataSeconddirectionSet()** to set the second direction of the material as the orientation type.
- PRO_MECH_MATLORI_PROJ_VECTOR—Specifies that the material orientation direction is determined by the projection vector.

The function **ProMachmaterialorientdataProjectiondataGet()** returns the structure containing the projection data for the material orientation. Use the function **ProMachmaterialorientdataProjectiondataSet()** to set the projection data for the material orientation.

The function **ProMachmaterialorientdataRotationGet()** returns additional rotations about one or more material directions. Use the function **ProMachmaterialorientdataRotationSet()** to set the additional rotations about one or more material directions.

The function

ProMachmaterialorientdataSurfacerotationGet() returns the rotation angle for the material orientation if the orientation type is PRO_MECH_MATLORI_SURFACE.

Use the function

ProMachmaterialorientdataSurfacerotationSet() to set the rotation angle for the material orientation if the orientation type is PRO_MECH_MATLORI_SURFACE.

Use the function

ProMachmaterialorientdataSurfacerotationUnset() removes the rotation angle for the material orientation if the orientation type is PRO_MECH_MATLORI_SURFACE.

The function **ProMachmaterialorientprojTypeGet()** returns the type of projection assigned to the material orientation. The types of materials are:

- PRO_MECH_MATLORI_PROJ_XYZ—Specifies the values for the X, Y, and Z components to define the projection vector for the material orientation. Use the function **ProMachmaterialorientprojXyzvectorGet()** to access the projection vector. Use the function **ProMachmaterialorientprojXyzvectorSet()** to set the projection vector for the material orientation.
- PRO_MECH_MATLORI_PROJ_POINTS—Specifies the two points used to define the projection vector for the material orientation. Use the function **ProMachmaterialorientprojPointsGet()** to access the two points. Use the function **ProMachmaterialorientprojPointsSet()** to set the two points used for projection.

The function **ProMachmaterialorientcsysCsysGet()** returns the coordinate system used to specify the material directions. Use the function **ProMachmaterialorientcsysCsysSet()** to set the coordinate system for the material directions.

The function **ProMachmaterialorientcsysProjectiontypeGet()** returns the projection vector for the material orientation object. The valid projection types are as follows:

- **PRO_MECH_MATLORI_CSYS_PROJ_CLOSEST**—Specifies the material Direction 1 through a series of calculations.
- **PRO_MECH_MATLORI_CSYS_PROJ_X**—Specifies that the material direction 1 is along the direction of the X axis of the referenced coordinate system projected onto the surface.

Use the function

ProMechmaterialorientcsysProjectiontypeSet() to set the projection type for the material orientation data.

The function **ProMechmaterialorientcsysXaxisGet()** returns the material direction to which the x-axis of the coordinate system is mapped. Use the function

ProMechmaterialorientcsysXaxisSet() to set the material direction to which the x-axis of the coordinate system is mapped.

The function **ProMechmaterialorientcsysYaxisGet()** returns the material direction to which the y-axis of the coordinate system is mapped. Use the function

ProMechmaterialorientcsysYaxisSet() to set the material direction to which the y-axis of the coordinate system is mapped.

The function **ProMechmaterialorientcsysZaxisGet()** returns the material direction to which the z-axis of the coordinate system is mapped. Use the function

ProMechmaterialorientcsysZaxisSet() to set the material direction to which the z-axis of the coordinate system is mapped.

Mechanica Shells

Shells are used to model a thin layer of a defined thickness for a specified part. If the part is relatively thin compared to its length and width, use of shell modeling is more efficient.

The functions described in this section provide read and write access to the data and contents of the Mechanica shell objects. Shells use the ProType field in the ProMechitem structure as PRO_SIMULATION_SHELL.

Functions Introduced:

- **ProMechshellTypeGet()**
- **ProMechshellMaterialGet()**
- **ProMechshellMaterialIdGet()**
- **ProMechshellMaterialIdSet()**

- **ProMechshellReferencesGet()**
- **ProMechshellReferencesSet()**
- **ProMechshellSimplifiedataGet()**
- **ProMechshellSimplifiedataSet()**
- **ProMechshellAdvanceddataGet()**
- **ProMechshellAdvanceddataSet()**
- **ProMechshellsimpleAlloc()**
- **ProMechshellsimpleThicknessGet()**
- **ProMechshellsimpleThicknessSet()**
- **ProMechshellsimpleFree()**
- **ProMechshelladvancedAlloc()**
- **ProMechshelladvancedMaterialorientGet()**
- **ProMechshelladvancedMaterialorientSet()**
- **ProMechshelladvancedShellpropsGet()**
- **ProMechshelladvancedShellpropsSet()**
- **ProMechshelladvancedFree()**

The function **ProMechshellTypeGet()** returns the shell types. You can define the following types of shells:

- **PRO_MECH_SHELL_SIMPLE**—Specifies a simple shell of uniform thickness. Use the function **ProMechshellSimplifiedataGet()** to access the data structure for the simple shell. Use the function **ProMechshellSimplifiedataSet()** to modify the data structure for the simple shell.
- **PRO_MECH_SHELL_ADVANCED**—Specifies an advanced shell that uses specified shell properties. Use the function **ProMechshellAdvanceddataGet()** to access the data structure for the advanced shells. Use the function **ProMechshellAdvanceddataSet()** to modify the data structure for the advanced shell.

The function **ProMechshellMaterialGet()** returns the material defined for the shell.

Use the function **ProMechshellMaterialIdSet()** to set the material id for the shell.

The function **ProMechshellReferencesGet()** returns the surfaces associated with the shell. Use the function **ProMechshellReferencesSet()** to set the references for the shell.

The function **ProMechshellsimpleThicknessGet()** returns the value of the thickness for the shell. Use the function **ProMechshellsimpleThicknessSet()** to set value of the thickness of the shell.

The function **ProMechshelladvancedMaterialorientGet()** returns the material orientations assigned to the advanced shell. Use the function **ProMechshelladvancedMaterialorientSet()** to set the material orientation for the advanced shell.

The function **ProMechshelladvancedShellpropsGet()** returns the shell properties associated with the model. Use the function **ProMechshelladvancedShellpropsSet()** to set the shell properties associated with the model.

Example 4: Creating Material Orientations Referencing a Selected Coordinate System

This example shows how to create a new material orientation item in the current model, referencing a selected coordinate system.

```
#include <ProMechMaterialOrientSet.h>
#include <ProUtil.h>
#include <ProMessage.h>

typedef struct
{
    ProMechGeomrefType type;
    ProMechGeomrefSubType subtype;
    int id;
    ProAsmcomppath *path;
} PTMechExGeomRef;

/*=====*\
FUNCTION: PTMechExGeomReferenceCreate
PURPOSE: This function is used to allocate the Mechanica geometry
reference.
\*=====*/

ProError PTMechExGeomReferenceCreate (PTMechExGeomRef* reference ,
ProMechGeomref *geom_ref)
{
    ProMechGeomref geomref_entity;

    status = ProMechgeomrefAlloc (&geomref_entity);
    PT_TEST_LOG_SUCC ("ProMechgeomrefAlloc");
```

```

/*-----*\
Set the geomref type, subtype,id and asmcomppath
\*-----*/

status = ProMechgeomrefTypeSet (geomref_entity, reference->type);
PT_TEST_LOG_SUCC ("ProMechgeomrefTypeSet");

/* Only For Ref.: Curve, Edge, Point, Surface */
if (reference->subtype != PRO_VALUE_UNUSED )

{
    status = ProMechgeomrefSubtypeSet (geomref_entity,
                                       reference->subtype);
    PT_TEST_LOG_SUCC ("ProMechgeomrefSubtypeSet");
}

if (reference->id != PRO_VALUE_UNUSED) /* For Ref. other than Part */
{
    status = ProMechgeomrefIdSet (geomref_entity, reference->id);
    PT_TEST_LOG_SUCC ("ProMechgeomrefIdSet");
}

if ( reference->path != NULL) /* Only For Ref. in Assembly */
{
    status = ProMechgeomrefPathSet (geomref_entity, reference->path);
    PT_TEST_LOG_SUCC ("ProMechgeomrefPathSet");
}

*geom_ref = geomref_entity;

return PRO_TK_NO_ERROR;
}

/*=====*\
FUNCTION: PTMechExMatOrient
PURPOSE: Add csys material orientation item
\*=====*/

ProError PTMechExMatOrient()

{
    ProName tk_matorient_wname;
    ProMechItem tk_constraint_item;
    ProMechMaterialOrientData orient_data;
    ProMechMaterialOrientCsys csys_data;
    ProMechStatus mech_status;
    int n_sels;
    ProSelection *sels;
    ProModelitem mech_item;
    ProMdl matorient_part;
    ProMechGeomref csys;

```

```

ProMechMaterialOrientRotation rotation = {1.0, 2.0, 3.0};
PTMechExGeomRef csys_ref;

status = ProMdlCurrentGet(&matorient_part);
PT_TEST_LOG_SUCC ("ProMdlCurrentGet");

/*-----*\
  Select csys
\*-----*/

status = ProMessageDisplay(msgfile, "PTMechEx Select csys", msgbuf);
PT_TEST_LOG_SUCC ("ProMessageDisplay()");

status =ProSelect ("csys",
                  1,
                  NULL,
                  NULL,
                  NULL,
                  NULL,
                  &sels,
                  &n_sels);
PT_TEST_LOG_SUCC ("ProSelect()");

status = ProSelectionModelitemGet (sels[0], &mech_item);
PT_TEST_LOG_SUCC ("ProSelectionModelitemGet()");

/*-----*\
  Creating Surface Type Projected Closest-axis Mat. Orientation.
\*-----*/

csys_ref.type      = PRO_MECH_COORD_SYSTEM;
csys_ref.subtype   = PRO_MECH_CSYS_CARTESIAN;
csys_ref.id        = mech_item.id ;
csys_ref.path      = NULL;

/*-----*\
  Create new mech item
\*-----*/

status = ProMechitemCreate ((ProSolid) matorient_part,
                           PRO_SIMULATION_MATL_ORIENT,
                           &tk_constraint_item);
PT_TEST_LOG_SUCC ("ProMechitemCreate");

ProStringToWstring(tk_matorient_wname, "tk_created_mat_orient");

status = ProMechitemNameSet ( &tk_constraint_item, tk_matorient_wname);
PT_TEST_LOG_SUCC ("ProMechitemNameSet");

/*-----*\
  Set the Material orient Description
\*-----*/

```



```

status = ProMechmaterialorientDescriptionSet (&tk_constraint_item,
                                              L"Mat Orient - Srf Type -
                                              Proj Closest axis");
PT_TEST_LOG_SUCC ("ProMechmaterialorientDescriptionSet");

/*-----*\
Allocate new Material orientation
\*-----*/

status = ProMechmaterialorientdataAlloc (&orient_data);
PT_TEST_LOG_SUCC ("ProMechmaterialorientdataAlloc");

/*-----*\
Set the Material orientation data object type
\*-----*/

status = ProMechmaterialorientdataObjecttypeSet (orient_data,
                                                  PRO_MECH_MATLORI_SURFACE);
PT_TEST_LOG_SUCC ("ProMechmaterialorientdataObjecttypeSet");

/*-----*\
Allocate csys data
\*-----*/

status = ProMechmaterialorientcsysAlloc (&csys_data);
PT_TEST_LOG_SUCC ("ProMechmaterialorientcsysAlloc");

/*-----*\
Create references
\*-----*/

status = PTMechExGeomReferenceCreate (&csys_ref, &csys);
PT_TEST_LOG_SUCC ("PTMechExConstraintCreate");

/*-----*\
Populate csys data
\*-----*/

status = ProMechmaterialorientcsysCsysSet (csys_data, csys);
PT_TEST_LOG_SUCC ("ProMechmaterialorientcsysCsysSet");

status = ProMechmaterialorientcsysProjectiontypeSet (csys_data,
                                                      PRO_MECH_MATLORI_CSYS_PROJ_CLOSEST);
PT_TEST_LOG_SUCC ("ProMechmaterialorientcsysProjectiontypeSet");

/*-----*\
Populate Material orient data
\*-----*/

```

```

    status = ProMechmaterialorientdataCsysdataSet (orient_data,
                                                    csys_data);
    PT_TEST_LOG_SUCC ("ProMechmaterialorientdataCsysdataSet");

    status = ProMechmaterialorientdataRotationSet ( orient_data, rotation);
    PT_TEST_LOG_SUCC ("ProMechmaterialorientdataRotationSet");

/*-----*\
    Set the Material orient data to the new constraint
\*-----*/

    status = ProMechmaterialorientDataSet (&tk_constraint_item,
                                           orient_data);
    PT_TEST_LOG_SUCC ("ProMechmaterialorientDataSet");

    status = ProMechitemUpdateComplete ( &tk_constraint_item);
    PT_TEST_LOG_SUCC ("ProMechitemUpdateComplete");

    status = ProMechitemStatusGet ( &tk_constraint_item, &mech_status);
    PT_TEST_LOG_SUCC ("ProMechitemStatusGet");

/*-----*\
    Print Message from the result
\*-----*/

    if (mech_status == PRO_MECH_ACTIVE)
    {
        status = ProMessageDisplay(msgfile, "PTMechEx Material orientation
                                           created successfully");
        PT_TEST_LOG_SUCC("ProMessageDisplay()");
    }
    else
    {
        status = ProMessageDisplay(msgfile, "PTMechEx Material orientation
                                           creation failed");
        PT_TEST_LOG_SUCC("ProMessageDisplay()");
    }

    return PRO_TK_NO_ERROR;
}

```

Shell Properties

Shell properties are used to create shells that are not homogeneous, or to create shells that are comprised of several layers. A shell property can be assigned to a face, region, or datum surface.

The functions described in this section provide read and write access to data and contents of the Mechanics shell property objects. Shell properties use the ProType field in the ProMechitem structure as PRO_SIMULATION_SHELL_PROPS.

Functions Introduced:

- **ProMechshellpropsTypeGet()**
- **ProMechshellpropsDescriptionGet()**
- **ProMechshellpropsDescriptionSet()**
- **ProMechshellpropsHomogeneousdataGet()**
- **ProMechshellpropsHomogeneousdataSet()**
- **ProMechshellpropsLaminatelayupdataGet()**
- **ProMechshellpropsLaminatelayupdataSet()**
- **ProMechshellpropsLaminatestiffdataGet()**
- **ProMechshellpropsLaminatestiffdataSet()**
- **ProMechshlprophomogeneousAlloc()**
- **ProMechshlprophomogeneousThicknessGet()**
- **ProMechshlprophomogeneousThicknessSet()**
- **ProMechshlprophomogeneousFree()**
- **ProMechshlproplaminatestiffAlloc()**
- **ProMechshlproplaminatestiffAppliedstressGet()**
- **ProMechshlproplaminatestiffAppliedstressSet()**
- **ProMechshlproplaminatestiffBendingstiffnessGet()**
- **ProMechshlproplaminatestiffBendingstiffnessSet()**
- **ProMechshlproplaminatestiffCouplingstiffnessGet()**
- **ProMechshlproplaminatestiffCouplingstiffnessSet()**
- **ProMechshlproplaminatestiffExtensionalstiffnessGet()**
- **ProMechshlproplaminatestiffExtensionalstiffnessSet()**
- **ProMechshlproplaminatestiffIntertiaperunitareaGet()**
- **ProMechshlproplaminatestiffIntertiaperunitareaSet()**
- **ProMechshlproplaminatestiffMassperunitareaGet()**
- **ProMechshlproplaminatestiffMassperunitareaSet()**
- **ProMechshlproplaminatestiffTansverseshearGet()**

- **ProMechshlprolaminatetestiffTansverseshearSet()**
- **ProMechshlprolaminatetestiffThermalresforceGet()**
- **ProMechshlprolaminatetestiffThermalresforceSet()**
- **ProMechshlprolaminatetestiffThermalresmomentGet()**
- **ProMechshlprolaminatetestiffThermalresmomentSet()**
- **ProMechshlprolaminatetestiffFree()**
- **ProMechstressscalcddataAlloc()**
- **ProMechstressscalcddataCzGet()**
- **ProMechstressscalcddataCzSet()**
- **ProMechstressscalcddataMaterialIdGet()**
- **ProMechstressscalcddataMaterialIdSet()**
- **ProMechstressscalcddataPlyorientationGet()**
- **ProMechstressscalcddataPlyorientationSet()**
- **ProMechstressscalcddataFree()**
- **ProMechstressscalcddataProarrayFree()**
- **ProMechstiffmatrixEntry11Get()**
- **ProMechstiffmatrixEntry11Set()**
- **ProMechstiffmatrixEntry12Get()**
- **ProMechstiffmatrixEntry12Set()**
- **ProMechstiffmatrixEntry16Get()**
- **ProMechstiffmatrixEntry16Set()**
- **ProMechstiffmatrixEntry22Get()**
- **ProMechstiffmatrixEntry22Set()**
- **ProMechstiffmatrixEntry26Get()**
- **ProMechstiffmatrixEntry26Set()**
- **ProMechstiffmatrixEntry66Get()**
- **ProMechstiffmatrixEntry66Set()**
- **ProMechstiffmatrixFree()**
- **ProMechtransverseshearEntry44Get()**
- **ProMechtransverseshearEntry44Set()**
- **ProMechtransverseshearEntry45Get()**

- **ProMechtransverseshearEntry45Set()**
- **ProMechtransverseshearEntry55Get()**
- **ProMechtransverseshearEntry55Set()**
- **ProMechtransverseshearFree()**
- **ProMechthermalrescoeffEntry11Get()**
- **ProMechthermalrescoeffEntry11Set()**
- **ProMechthermalrescoeffEntry12Get()**
- **ProMechthermalrescoeffEntry12Set()**
- **ProMechthermalrescoeffEntry22Get()**
- **ProMechthermalrescoeffEntry22Set()**
- **ProMechthermalrescoeffFree()**
- **ProMechshlproplaminatelayupAlloc()**
- **ProMechshlproplaminatelayupLayersGet()**
- **ProMechshlproplaminatelayupLayersSet()**
- **ProMechshlproplaminatelayupTypeGet()**
- **ProMechshlproplaminatelayupTypeSet()**
- **ProMechshlproplaminatelayupFree()**
- **ProMechshlproplamlayuplayerMaterialIdGet()**
- **ProMechshlproplamlayuplayerMaterialIdSet()**
- **ProMechshlproplamlayuplayerNumberGet()**
- **ProMechshlproplamlayuplayerNumberSet()**
- **ProMechshlproplamlayuplayerOrientationGet()**
- **ProMechshlproplamlayuplayerOrientationSet()**
- **ProMechshlproplamlayuplayerThicknessGet()**
- **ProMechshlproplamlayuplayerThicknessSet()**
- **ProMechshlproplamlayuplayerShellpropsGet()**
- **ProMechshlproplamlayuplayerShellpropsSet()**
- **ProMechshlproplamlayuplayerFree()**
- **ProMechshlproplamlayuplayerProarrayFree()**

The function **ProMechshellpropsTypeGet()** returns the type of properties that have been defined for the shell. The types of shell properties are:

- **PRO_MECH_SHLPROP_HOMOGENEOUS**—Assigned to homogenous shells. A homogeneous shell consists of a single material whose properties do not vary through the thickness of the shell. Use the function **ProMechshellpropsHomogeneousdataGet()** to access the data structure for homogeneous shell property. Use the function **ProMechshellpropsHomogeneousdataSet()** to set the homogeneous shell property.
- **PRO_MECH_SHLPROP_LAMINATE_STIFFNESS**—Assigned to laminate shells to specify their degree of stiffness. Laminate shells consists of one or more materials whose properties may vary through the thickness of the shell. Use the function **ProMechshellpropsLaminatestiffdataGet()** to access the data structure for the laminate stiffness shell property. Use the function **ProMechshellpropsLaminatestiffdataSet()** to set the laminate stiffness shell property.
- **PRO_MECH_SHLPROP_LAMINATE_LAYUP**—Assigned to laminate shells to define them as layers of shells. Use the function **ProMechshellpropsLaminatelayupdataGet()** to access the data structure for the laminate layup shell property. Use the function **ProMechshellpropsLaminatelayupdataSet()** to set the laminate layup shell property.

The function **ProMechshellpropsDescriptionGet()** returns the description for the shell properties. Use the function **ProMechshellpropsDescriptionSet()** to modify the description for the shell properties.

The function **ProMechshlprophomogeneousThicknessGet()** returns the shell thickness defined for shells. Use the function **ProMechshlprophomogeneousThicknessSet()** to set the thickness for the shell properties.

The function **ProMechshlproplaminatestiffAppliedstressGet()** returns the calculation of stresses and strains for the laminate stiffness shell property type. The resultant array contains the shell "Top" location and the shell "Bottom" location. The values specified in these areas are used to calculate the stresses and strains for the corresponding areas. Use the function **ProMechshlproplaminatestiffAppliedstressSet()** to set the value of the stress and strain in the calculation array.

The function **ProMechstresscalcddataCzGet()** returns the distance from the midsurface of the shell at which the stresses and strains for the laminate shell is calculated. The Cz is defined relative to material direction 3 of the material orientation assigned to the shell. Use the function **ProMechstresscalcddataCzSet()** to set the Cz value used to calculate the stress and strain.

The function **ProMechstresscalcddataMaterialIdGet()** returns the material specified at the CZ location. Use the function **ProMechstresscalcddataMaterialIdSet()** to set the material id.

The function **ProMechstresscalcddataPlyorientationGet()** returns the orientation of the ply material. The ply orientation angle is measured as a counter-clockwise rotation from material direction 1 about material direction 3. Use the function **ProMechstresscalcddataPlyorientationSet()** to set the orientation of the ply material.

The function **ProMechshlproplaminatetestiffBendingstiffnessGet()** returns the shell bending stiffness matrix. Use the function **ProMechshlproplaminatetestiffBendingstiffnessSet()** to set the shell bending stiffness matrix.

The function **ProMechshlproplaminatetestiffCouplingstiffnessGet()** returns the shell coupling stiffness matrix. Use the function **ProMechshlproplaminatetestiffCouplingstiffnessSet()** to set the shell coupling stiffness matrix.

The function **ProMechshlproplaminatetestiffExtensionalstiffnessGet()** returns the shell extensional stiffness matrix. Use the function **ProMechshlproplaminatetestiffExtensionalstiffnessSet()** to set the shell extensional stiffness matrix.

The functions **ProMechstiffmatrixEntry11Get()**, **ProMechstiffmatrixEntry12Get()**, **ProMechstiffmatrixEntry16Get()**, **ProMechstiffmatrixEntry22Get()**, **ProMechstiffmatrixEntry26Get()**, and **ProMechstiffmatrixEntry66Get()** provide access the elements of the stiffness matrix.

Use the functions **ProMechstiffmatrixEntry11Set()**, **ProMechstiffmatrixEntry12Set()**, **ProMechstiffmatrixEntry16Set()**, **ProMechstiffmatrixEntry22Set()**, **ProMechstiffmatrixEntry26Set()**, and **ProMechstiffmatrixEntry66Set()** to modify the elements of the stiffness matrix.

The function

ProMechshlproplaminatetestiffIntertiaperunitareaGet() returns the rotary inertia per unit area for the laminate stiffness properties. Use the function **ProMechshlproplaminatetestiffIntertiaperunitareaSet()** to set the inertia per unit area for the laminate stiffness properties.

The function

ProMechshlproplaminatetestiffMassperunitareaGet() returns the mass per unit area for the laminate stiffness properties. Use the function **ProMechshlproplaminatetestiffMassperunitareaSet()** to set the mass per unit area for the laminate stiffness properties.

The function

ProMechshlproplaminatetestiffTansverseshearGet() returns the transverse shear stiffness for the laminate shell. Use the function **ProMechshlproplaminatetestiffTansverseshearSet()** to set the transverse shear stiffness for the laminate shell.

The functions **ProMechtransverseshearEntry44Get()**, **ProMechtransverseshearEntry45Get()**, and **ProMechtransverseshearEntry55Get()** provide access to the elements 44, 45, and 55 of the transverse shear matrix. Use the functions **ProMechtransverseshearEntry44Set()**, **ProMechtransverseshearEntry45Set()**, and **ProMechtransverseshearEntry55Set()** to set the elements 44, 45, and 55 of the transverse shear matrix.

The functions

ProMechshlproplaminatetestiffThermalresforceGet() and **ProMechshlproplaminatetestiffThermalresmomentGet()** return the thermal resultant coefficients for the laminate shell. The thermal coefficients are specified as Force and Moment. Use the functions **ProMechshlproplaminatetestiffThermalresforceSet()** and **ProMechshlproplaminatetestiffThermalresmomentSet()** to set the thermal resultant coefficients for the laminate shell.

The functions **ProMechthermalrescoeffEntry11Get()**, **ProMechthermalrescoeffEntry12Get()**, and, **ProMechthermalrescoeffEntry22Get()** provide access to the elements 11, 12, and 22 of the thermal resultant coefficient matrix. Use the functions **ProMechthermalrescoeffEntry11Set()**, **ProMechthermalrescoeffEntry12Set()**, **ProMechthermalrescoeffEntry22Set()** to set the elements 11, 12, and 22 of the thermal resultant coefficient matrix.

The function **ProMechshlproplaminatelayupLayersGet()** returns an array of layers, or plies, stacked on each other to form the laminate. Use the function **ProMechshlproplaminatelayupLayersSet()** to set the layers assigned to the laminate shell properties.

The function **ProMechshlproplaminatelayupTypeGet()** returns the layer repetition pattern for the laminate layup type shells. The types of repetition patterns are:

- **PRO_MECH_LAMLAYUP_SYMMETRIC**—Specifies that the layers are repeated in reverse order.
- **PRO_MECH_LAMLAYUP_ANTISYMMETRIC**—Specifies that the layers are repeated in reverse order, and the orientation is also changed.
- **PRO_MECH_LAMLAYUP_NEITHER**—Specifies that the layers are not repeated.

Use the function **ProMechshlproplaminatelayupTypeSet()** to set the type of laminate layup shell properties.

The function **ProMechshlproplamlayuplayerMaterialIdGet()** returns the name of the material assigned to the specified layer. Use the function **ProMechshlproplamlayuplayerMaterialIdSet()** to set the id of the specified laminate layer.

The function **ProMechshlproplamlayuplayerNumberGet()** returns the number of times a particular layer is repeated for the laminate. Use the function **ProMechshlproplamlayuplayerNumberSet()** to set the number for the laminate layup layer.

The function **ProMechshlproplamlayuplayerOrientationGet()** returns the orientation of the specified layer of the laminate. Use the function **ProMechshlproplamlayuplayerOrientationSet()** to set the orientation of the specified layer of the laminate.

The function **ProMechshlproplamlayuplayerThicknessGet()** returns the thickness of the specified layer of the laminate. Use the function **ProMechshlproplamlayuplayerThicknessSet()** to set the thickness of the specified layer of the laminate.

The function **ProMechshlproplamlayuplayerShellpropsGet()** returns the shell properties for the specified laminate layer. Use the function **ProMechshlproplamlayuplayerShellpropsSet()** to set the shell properties.

Shell Pairs

The shell pairs are created based on the surfaces belonging to a part. The shell pairs are compressed to a mid surface or set of mid surfaces and shell elements are assigned to it.

The functions described in this section provide read and write access to the data and contents of the Mechanics shell pair objects. Shell pairs use the ProType field in the ProMechitem structure as PRO_SIMULATION_SHELL_PAIR.

Functions Introduced:

- **ProMechshlpairrefsAlloc()**
- **ProMechshellpairMaterialIdGet()**
- **ProMechshellpairMaterialIdSet()**
- **ProMechshellpairMaterialOrientIdGet()**
- **ProMechshellpairMaterialOrientIdSet()**
- **ProMechshellpairReferencesGet()**
- **ProMechshellpairReferencesSet()**
- **ProMechshlpairrefsTypeGet()**
- **ProMechshlpairrefsTypeSet()**
- **ProMechshlpairrefsPlacementtypeGet()**
- **ProMechshlpairrefsPlacementtypeSet()**
- **ProMechshlpairrefsBottomreferencesGet()**
- **ProMechshlpairrefsBottomreferencesSet()**
- **ProMechshlpairrefsTopreferencesGet()**
- **ProMechshlpairrefsTopreferencesSet()**
- **ProMechshlpairrefsSelectedplacementGet()**

- **ProMechshlpairrefsSelectedplacementSet()**
- **ProMechshlpairrefsExtendAdjacentSurfacesGet()**
- **ProMechshlpairrefsExtendAdjacentSurfacesSet()**
- **ProMechshlpairrefsFree()**

The function **ProMechshellpairMaterialIdGet()** returns the material used to create the shell pair. Use the function **ProMechshellpairMaterialIdSet()** to set the material id of the shell pair.

The function **ProMechshellpairReferencesGet()** returns the geometric references for the shell pair item. Use the function **ProMechshellpairReferencesSet()** to set the geometric references for the shell pair item.

The function **ProMechshlpairrefsTypeGet()** returns the type of shell pair references. The types of shell pairs are:

- **PRO_MECH_SHELL_PAIR_CONSTANT**—Specifies a shell pair with constant thickness. All opposing surfaces parallel to each other and equidistant from the opposing surface for a constant-thickness shell pair.
- **PRO_MECH_SHELL_PAIR_VARIABLE**—Specifies a variable thickness shell pair. Both opposing surfaces are neither parallel nor concentric for a variable thickness shell pair.
- **PRO_MECH_SHELL_PAIR_MULTI_CONSTANT**—Specifies a shell pair with multiple pairs of surfaces. For a pair of surfaces, each surface is parallel to and equidistant from the opposing surface. However, the distance between the surfaces for each of the multiple pairs may vary.

Note: In Pro/ENGINEER Wildfire 5.0, when you open a model with shell pairs created in Pro/ENGINEER Wildfire 4.0 or earlier, Pro/ENGINEER searches for the shell pairs with thickness type VARIABLE and determines if the thickness type was VARIABLE or not. If the shell pairs are not VARIABLE, Pro/ENGINEER Wildfire 5.0 changes the thickness type of such shell pairs from VARIABLE to MULTI_CONSTANT in the database.

Use the function **ProMechshlpairrefsTypeSet()** to set the type of shell pair references.

Note: Pro/ENGINEER Wildfire 5.0 onward, the shell pair type `PRO_MECH_SHELL_PAIR_VARIABLE` is not supported in the Native mode. If used, **ProMechshlpairrefsTypeSet()** returns an error type `PRO_TK_BAD_INPUTS`.

The functions **ProMechshlpairrefsTopreferencesGet()** and **ProMechshlpairrefsBottomreferencesGet()** provide access to the top and bottom references for the shell pair. This should be defined for all shell pair types. Use the functions **ProMechshlpairrefsBottomreferencesSet()** and **ProMechshlpairrefsTopreferencesSet()** to set the top and bottom references for the shell pair.

The function **ProMechshlpairrefsPlacemnttypeGet()** returns the placement of shell pair. The types of placement references are as follows:

- `PRO_MECH_SHELL_PAIR_PLACEMENT_TOP`—Specifies that the placement uses the top surface of the surface pair.
- `PRO_MECH_SHELL_PAIR_PLACEMENT_BOTTOM`—Specifies that the placement uses the bottom surface of the surface pair.
- `PRO_MECH_SHELL_PAIR_PLACEMENT_MIDDLE`—Specifies that the placement uses the mid surface of the surface pair.
- `PRO_MECH_SHELL_PAIR_PLACEMENT_SELECTED`—Specifies that the placement uses an arbitrary selected surface, which can be a datum surface.

Use the function **ProMechshlpairrefsPlacemnttypeSet()** to set the placement type of the shell pair.

Use the function **ProMechshlpairrefsSelectedplacementGet()** to access the data structure containing the selected surface. Use the function **ProMechshlpairrefsSelectedplacementSet()** to set the surface for the placement reference of type `SELECTED`.

Use the function **ProMechshlpairrefsExtendAdjacentSurfacesGet()** to retrieve the value of the flag that indicates whether the adjacent surfaces in a shell pair will be extended during meshing or not. For a mixed model, you can set this flag to extend surfaces adjacent to the top and bottom surfaces of the shell pair by using the function **ProMechshlpairrefsExtendAdjacentSurfacesSet()**. However, this is possible only if the angle between the shell pair surfaces and their adjacent surfaces is less than the value specified by the `sim_extend_surf_max_angle` configuration option.

Note:

- For a midsurface model, Pro/ENGINEER Mechanica extends the adjacent surfaces regardless of the value of `sim_extend_surf_max_angle`. The default value of `sim_extend_surf_max_angle` is 30 degrees.
- You must initialize the Mechanica environment to use these functions. For shell pairs defined in Pro/ENGINEER Mechanica 4.0 and earlier, this flag is set to 1 by default.

Use the function **ProMechshlpairrefsFree()** to free the simple shell data handle in Pro/ENGINEER Mechanica.

Interfaces

Interfaces, also called connections, are used to connect surfaces. When you create an interface in Mechanica, specify how Mechanica will treat the connected surfaces during meshing and analysis. Interfaces use the `ProType` field in the `ProMechitem` structure as `PRO_SIMULATION_INTERFACE`.

Functions Introduced:

- **ProMechinterfaceTypeGet()**

- **ProMechinterfaceReferencesGet()**
- **ProMechinterfaceReferencesSet()**
- **ProMechinterfacebonddataAlloc()**
- **ProMechinterfaceBonddataGet()**
- **ProMechinterfaceBonddataSet()**
- **ProMechinterfacebonddataFree()**
- **ProMechinterfacefreedataAlloc()**
- **ProMechinterfaceFreedataGet()**
- **ProMechinterfaceFreedataSet()**
- **ProMechinterfacefreedataFree()**
- **ProMechinterfacecontactdataAlloc()**
- **ProMechinterfaceContactdataGet()**
- **ProMechinterfaceContactdataSet()**
- **ProMechinterfacecontactdataFree()**
- **ProMechinterfacethrresistdataAlloc()**
- **ProMechinterfaceThrresistdataGet()**

- **ProMechinterfaceThrrresistdataSet()**
- **ProMechinterfaceThrrresistdataFree()**

The function **ProMechinterfaceTypeGet()** returns the type of interface used to connect surfaces. The types of interfaces are:

- Structural—Specifies the default for the interfaces created between the geometry in a structural model for meshing and running.
 - **PRO_MECH_INTERFACE_BOND_STRUCT**—Specifies that the contacting surfaces as bonded. This means that matching nodes on contacting surfaces merge.
 - **PRO_MECH_INTERFACE_CONTACT_STRUCT**—Specifies an interface between components when you want the components to have the freedom to remain separate from each other, and also when you want the components to transfer forces between them when they touch, or come into contact with each other.
 - **PRO_MECH_INTERFACE_FREE_STRUCT**—Specifies two surfaces as contacting, but does not merge the nodes. Meshes on the contacting surfaces are identical, and matching nodes are coincident.
- Thermal—Specifies the default for the interfaces created between the geometry in a thermal model for meshing and running.
 - **PRO_MECH_INTERFACE_BOND_THERM**—Specifies that coincident geometry is bonded.
 - **PRO_MECH_INTERFACE_FREE_THERM**—Specifies that no geometry in the assembly is merged.
 - **PRO_MECH_INTERFACE_RESIST_THERM**—Specify this interface type to create a thermal resistance interfaces at run-time.

The function **ProMechinterfaceReferencesGet()** returns the geometric entities selected to create the interface. Use the function **ProMechinterfaceReferencesSet()** to set the geometric entities selected to create the interface.

The function **ProMechinterfacebonddataAlloc()** allocates memory for the Mechanica bonded interface data.

The function **ProMechinterfaceBonddataGet()** provides access to the bonded interface data. The function returns whether the specified bonded interface should use links between pairs of nodes.

Use the function **ProMechinterfaceBonddataSet()** to assign the bonded interface data to the Mechanica item.

Use the function **ProMechinterfacebonddataFree()** to free the memory of the bonded interface data.

The function **ProMechinterfacefreedataAlloc()** allocates memory for the Mechanica free interface data structure.

The function **ProMechinterfaceFreedataGet()** provides access to the free interface data. The function returns a flag that indicates whether the meshes generated on the surfaces of the interface are coincident or not. Use the function **ProMechinterfaceFreedataSet()** to assign the free interface data to the Mechanica item.

The function **ProMechinterfacecontactdataAlloc()** allocates memory for the Mechanica contact interface data structure.

The function **ProMechinterfaceContactdataGet()** provides access to the properties of the contact interface data. Use the function **ProMechinterfaceContactdataSet()** to assign the contact interface data to the Mechanica item.

The function **ProMechinterfacethrresistdataAlloc()** allocates memory for the Mechanica thermal resistance interface data structure.

The function **ProMechinterfaceThrresistdataGet()** provides access to the thermal resistance interface data. Use the function **ProMechinterfaceThrresistdataSet()** to assign the thermal resistance interface data to the Mechanica item.

Bonded Interface

Functions Introduced:

- **ProMechbondinterfacedataMergenodesGet()**
- **ProMechbondinterfacedataMergenodesSet()**

The function **ProMechbondinterfacedataMergenodesGet()** specifies if coincident nodes of components or surfaces touching each other are merged during meshing. Use the function **ProMechbondinterfacedataMergenodesSet()** to set whether coincident nodes for bonded interface should merge.

Contact Interface

Functions Introduced:

- **ProMechcontactinterfacedataSeparationdistanceExprGet()**
- **ProMechcontactinterfacedataSeparationdistanceExprSet()**
- **ProMechcontactinterfacedataAnglebetweensurfacesExprGet()**
- **ProMechcontactinterfacedataAnglebetweensurfacesExprSet()**
- **ProMechcontactinterfacedataCheckonlyplanarGet()**
- **ProMechcontactinterfacedataCheckonlyplanarSet()**
- **ProMechcontactinterfacedataInfiniteFrictionGet()**
- **ProMechcontactinterfacedataInfiniteFrictionSet()**
- **ProMechcontactinterfacedataSlippageGet()**
- **ProMechcontactinterfacedataSlippageSet()**
- **ProMechcontactinterfacedataCoefffrictionGet()**
- **ProMechcontactinterfacedataCoefffrictionSet()**
- **ProMechcontactinterfacedataSplitsurfacesGet()**
- **ProMechcontactinterfacedataSplitsurfacesSet()**
- **ProMechcontactinterfacedataCompatiblemeshGet()**
- **ProMechcontactinterfacedataCompatiblemeshSet()**

To create a contact interface between two components of an assembly, with the reference type as component-component, specify the selection filter tolerance criteria. You can specify the selection filter tolerance criteria using the following functions.

Use the function

ProMechcontactinterfacedataSeparationdistanceExprGet() to retrieve the separation distance. Separation distance is the distance between the surface pairs that you want to use to define a contact interface. This separation distance is the limit beyond which a contact interface cannot be created. If the distance between the surfaces of two components is smaller than the separation distance, the surfaces are used for creation of the contact interface.

The separation distance is returned as an expression of type **ProMechExpression**.

Note: The function **ProMechcontactinterfacedataSeparationdistanceExprGet()** supersedes the function **ProMechcontactinterfacedataSeparationdistanceGet()**.

The function

ProMechcontactinterfacedataSeparationdistanceExprSet() sets the separation distance. Specify this distance as an expression of type `ProMechExpression`.

Note: The function **ProMechcontactinterfacedataSeparationdistanceExprSet()** supersedes the function **ProMechcontactinterfacedataSeparationdistanceSet()**.

The function

ProMechcontactinterfacedataAnglebetweensurfacesExprGet() returns the angle between planar surfaces while creating contact interface. This angle is returned as an expression of type `ProMechExpression`.

Note: The function **ProMechcontactinterfacedataAnglebetweensurfacesExprGet()** supersedes the function **ProMechcontactinterfacedataAnglebetweensurfacesGet()**.

The function

ProMechcontactinterfacedataAnglebetweensurfacesExprSet() sets the angle between surfaces while creating contact interface. Specify this angle as an expression of type `ProMechExpression`.

Note: The function **ProMechcontactinterfacedataAnglebetweensurfacesExprSet()** supersedes the function **ProMechcontactinterfacedataAnglebetweensurfacesSet()**.

The function

ProMechcontactinterfacedataCheckonlyplanarGet() returns a true if the contact is created between planar surfaces in a component-component type interface.

Use the function

ProMechcontactinterfacedataCheckonlyplanarSet() to create contacts only between planar surfaces.

The properties of the contact interface are as follows:

The function

ProMechcontactinterfacedataInfiniteFrictionGet() specifies if the contact interface uses infinite friction between surfaces or nodes. The function

ProMechcontactinterfacedataInfiniteFrictionSet() to create a contact with infinite friction. The two components or surfaces cannot slide relative to each other.

The function **ProMechcontactinterfacedataSlippageGet()** returns true if slippage has occurred in the contact region during the analysis. Use the function

ProMechcontactinterfacedataSlippageSet() to check for slippage in the contact area during analysis.

The function **ProMechcontactinterfacedataCoefffrictionGet()** returns the coefficient of friction used to calculate the slippage in the contact region during analysis. Use the function

ProMechcontactinterfacedataCoefffrictionSet() to set the coefficient of friction for computing the slippage. Specify a positive value as the coefficient of friction.

The function **ProMechcontactinterfacedataSplitsurfacesGet()** returns true if you split the surface shared by the volumes used to define the interface.

Use the function

ProMechcontactinterfacedataSplitsurfacesSet() to specify whether the interface should split surfaces.

The function

ProMechcontactinterfacedataCompatiblemeshGet() specifies if a compatible mesh is created when components in the assembly are touching.

Use the function

ProMechcontactinterfacedataCompatiblemeshSet() to create geometrically-consistent node locations when the mesh for the surfaces of your interface is generated.

Thermal Resistance Interface

Functions Introduced:

- **ProMechthrrresistinterfacedataConductivityGet()**
- **ProMechthrrresistinterfacedataConductivitySet()**

The function

ProMechthrrresistinterfacedataConductivityGet() returns the heat transfer coefficient for the interface. The heat transfer coefficient is the heat that passes through the interface per unit time per unit area when the temperature difference between opposite interface surfaces is one unit.

Use the function

ProMechthrrresistinterfacedataConductivitySet() to set the heat transfer coefficient for the interface. Specify a positive number as the heat transfer coefficient.

Free Interface

Functions Introduced:

- **ProMechfreeinterfacedataSplitsurfacesGet()**
- **ProMechfreeinterfacedataSplitsurfacesSet()**

The function **ProMechfreeinterfacedataSplitsurfacesGet()** returns true if you split the surface shared by the volumes used to define the free interface.

The function **ProMechfreeinterfacedataSplitsurfacesSet()** to specify whether the free interface should split surfaces.

Gaps

A gap is a nonlinear element, used to model connection between points, edges and curves, or surfaces in your model by connecting two nodes in separated geometries. Gaps use the ProType field in the ProMechitem structure as PRO_SIMULATION_GAP.

Functions Introduced:

- **ProMechgapTypeGet()**
- **ProMechgapReferencesGet()**
- **ProMechgapReferencesSet()**
- **ProMechgapSimplesdataGet()**
- **ProMechgapSimplesdataSet()**

- **ProMechsimplegapdataAlloc()**
- **ProMechsimplegapdataFree()**
- **ProMechsimplegapdataYdirectionGet()**
- **ProMechsimplegapdataYdirectionSet()**
- **ProMechsimplegapdataDistributiontypeGet()**
- **ProMechsimplegapdataDistributiontypeSet()**
- **ProMechsimplegapdataAxialstiffnessGet()**
- **ProMechsimplegapdataAxialstiffnessSet()**
- **ProMechsimplegapdataTransversestiffnessGet()**
- **ProMechsimplegapdataTransversestiffnessSet()**
- **ProMechsimplegapdataClearanceGet()**
- **ProMechsimplegapdataClearanceSet()**

The function **ProMechgapTypeGet()** returns the type of gap for the specified Mechanica Gap.

The function **ProMechgapReferencesGet()** returns the geometric entities of the model that are selected to create the gap. Use the function **ProMechgapReferencesSet()** to set the valid geometric references for the specified gap.

The function **ProMechgapSimpledataGet()** returns the data structure for the Mechanica gap data. The Mechanica gap data structure defines the y-direction and the stiffness properties of the gap. Use the function **ProMechgapSimpledataSet()** to set the Mechanica gap data structure.

The function **ProMechsimplegapdataAlloc()** allocates memory for the Mechanica gap data structure.

ProMechsimplegapdataYdirectionGet() returns the orientation of the XY-plane of the gap. Use the function **ProMechsimplegapdataYdirectionSet()** to set the Y-direction of the gap data.

The function **ProMechsimplegapdataDistributiontypeGet()** returns the method used to calculate the axial and transverse stiffnesses for the gap data. The types of distribution are as follows:

- **PRO_MECH_GAP_DISTR_TOTAL**—Specifies the sum of stiffness of all the contact elements.

- **PRO_MECH_GAP_DISTR_PER_UNIT**—Specifies that the stiffness value is calculated using the area of the first selected surface.

Use the function

ProMechsimplegapdataDistributiontypeSet() to set the distribution type for the gap data.

The function **ProMechsimplegapdataAxialstiffnessGet()** returns the axial stiffness for the gap. The axial stiffness defines a stiffness or spring factor. Use the function

ProMechsimplegapdataAxialstiffnessSet() to set the axial stiffness for the gap.

The function **ProMechsimplegapdataTransversestiffnessGet()** returns the transverse stiffness for the gap. The transverse stiffness defines the elastic stiffness of the material. Use the function **ProMechsimplegapdataTransversestiffnessSet()** to set the transverse stiffness of the gap.

The function **ProMechsimplegapdataClearanceGet()** returns the distance between two nodes at which the axial and transverse stiffnesses are active due to displacement during analysis. Use the function **ProMechsimplegapdataClearanceSet()** to set the clearance for the gap data.

Mesh Control

Mesh controls define the characteristics of a mesh. Mesh controls specify the minimum or maximum size of the elements, the distribution of nodes along edges, hard points and hard curves, mesh ID numbering, mesh ID offsets, and the displacement coordinate system. The functions described in this section provide access to the data and contents of the Mechanics mesh control objects. Mesh controls use the ProType field in the ProMechitem structure as PRO_SIMULATION_MESH_CNTRL.

Functions Introduced:

- **ProMechmeshcntrlTypeGet()**
- **ProMechmeshcntrlAutogemedgedistrdataGet()**
- **ProMechmeshcntrlAutogemedgedistrdataSet()**
- **ProMechmeshcntrlAutogemminedgedataGet()**

- **ProMechmeshcntrlAutogemminededdataSet()**
- **ProMechmeshcntrlAutogemelemsizedataGet**
- **ProMechmeshcntrlAutogemelemsizedataSet**
- **ProMechmeshcntrlAutogemedgelencrvdataGet**
- **ProMechmeshcntrlAutogemedgelencrvdataSet**
- **ProMechmeshcntrlEdgedistrdataGet()**
- **ProMechmeshcntrlEdgedistrdataSet()**
- **ProMechmeshcntrlElemsizedataGet()**
- **ProMechmeshcntrlElemsizedataSet()**
- **ProMechmeshcntrlShellcsysGet()**
- **ProMechmeshcntrlShellcsysSet()**
- **ProMechmeshcntrlHardpointGet()**
- **ProMechmeshcntrlHardpointSet()**
- **ProMechmeshcntrlHardcurveSet()**
- **ProMechmeshcntrlldsoffsetGet()**
- **ProMechmeshcntrlldsoffsetSet()**
- **ProMechmeshcntrlNumberingGet()**
- **ProMechmeshcntrlNumberingSet()**
- **ProMechmeshcntrlSuppressGet()**
- **ProMechmeshcntrlSuppressSet()**
- **ProMechmeshcntrlAutogemisolateexcludedataGet()**
- **ProMechmeshcntrlAutogemisolateexcludedataSet()**
- **ProMechmeshcntrlReferencesGet()**
- **ProMechmeshcntrlReferencesSet()**
- **ProMechmcautogemHardpointSet()**
- **ProMechmcautogemHardcurveSet()**

The function **ProMechmeshcntrlTypeGet()** returns the type of mesh control. The types of mesh controls are:

- **PRO_MECH_MC_AGEM_EDGE_DISTR**—Specifies the edge distribution data for AutoGEM. Use the function **ProMechmeshcntrlAutogemedgedistrdataGet()** to access the AutoGEM edge distribution data for the mesh control item. Use the function **ProMechmeshcntrlAutogemedgedistrdataSet()** to modify the AutoGEM edge distribution data for the mesh control item.
Note: If you are creating a new mesh control using this type of data, you should assign the model as a reference using the function **ProMechmeshcntrlReferencesSet()**.
- **PRO_MECH_MC_AGEM_MIN_EDGE**—Specifies the minimum edge and face angles for AutoGEM. Use the function **ProMechmeshcntrlAutogemminededgedataGet()** to access the AutoGEM minimum edge data for the mesh control item. Use the function **ProMechmeshcntrlAutogemminededgedataSet()** to modify the AutoGEM minimum edge data for the mesh control item.
Note: If you are creating a new mesh control using this type of data, you should assign the model as a reference using the function **ProMechmeshcntrlReferencesSet()**.
- **PRO_MECH_MC_AGEM_MAX_ELEMENT_SIZE**—Specifies the maximum element size for the AutoGEM mesh. Use the function **ProMechmeshcntrlAutogemelemsizedataGet()** to access the AutoGEM maximum element data for the mesh control item. Use the function **ProMechmeshcntrlAutogemelemsizedataSet()** to modify the AutoGEM maximum element data for the mesh control item.
Note: If you are creating a new mesh control using this type of data, you should assign the model as a reference using the function **ProMechmeshcntrlReferencesSet()**.
- **PRO_MECH_MC_AGEM_EDGE_LEN_CRV**—Specifies the ratio of edge lengths of mesh elements adjacent to concave surfaces to the radius of curvature of the concave surfaces. Use the function **ProMechmeshcntrlAutogemedgelencrvdataGet()** to access the edge length by curvature data for the mesh control item. Use the function **ProMechmeshcntrlAutogemedgelencrvdataSet()** to set the edge length by curvature ratio.

- **PRO_MECH_MC_EDGE_DISTRIBUTION**—Specifies the number of nodes on one or more edges of the curves in the model. Use the function **ProMechmeshcntrlEdgedistrdataGet()** to access the edge distribution data for the mesh control item. Use the function **ProMechmeshcntrlEdgedistrdataSet()** to set the edge distribution data.

Note: If you are creating a new mesh control using this type of data, you should assign the model as a reference using the function **ProMechmeshcntrlReferencesSet()**.
- **PRO_MECH_MC_DISPL_CSYS**—Specifies the displacement coordinate system used for displaying results for nodes associated with points, edges, curves, or surfaces.
- **PRO_MECH_MC_MAX_ELEMENT_SIZE**—Specifies the maximum element size for the mesh. Use the function **ProMechmeshcntrlElemsizedataGet()** to access the maximum element size data for the mesh control item. Use the function **ProMechmeshcntrlElemsizedataSet()** to set the maximum element size for the mesh.

Note: If you are creating a new mesh control using this type of data, you should assign the model as a reference using the function **ProMechmeshcntrlReferencesSet()**.
- **PRO_MECH_MC_MIN_ELEMENT_SIZE**—Specifies the minimum element size for the mesh. Use the function **ProMechmeshcntrlElemsizedataGet()** to access the minimum element size data for the mesh control item. Use the function **ProMechmeshcntrlElemsizedataSet()** to set the maximum element size for the mesh.
- **PRO_MECH_MC_SHELL_CSYS**—Specifies the coordinate system used for displaying results for nodes associated with shell or quilt surfaces. Use the function **ProMechmeshcntrlShellcsysGet()** to access the shell coordinate system for the mesh control item. Use the function **ProMechmeshcntrlShellcsysSet()** to set the shell coordinate system for the mesh control item.
- **PRO_MECH_MC_HARD_POINT**—Specifies a datum point that is defined as a hard point. Use the function **ProMechmeshcntrlHardpointGet()** to access the hard point data for the mesh control item. Use the function **ProMechmeshcntrlHardpointSet()** to set the hard point data for the mesh control item.

- PRO_MECH_MC_HARD_CURVE—Specifies a datum curve that is defined as a hard curve. This type has no data, only references. Use the function **ProMechmeshctrlHardcurveSet()** to set the mesh control item to be of the type hard curve. The reference for the item will be used for the control.
- PRO_MECH_MC_IDS_OFFSET—Specifies the offset for the node IDs and element IDs for a component. Use the function **ProMechmeshctrlIdsoffsetGet()** to access the offset data for the mesh control item. Use the function **ProMechmeshctrlIdsoffsetSet()** to set the offset data.
- PRO_MECH_MC_MESH_NUMBERING—Specifies a node and an element ID range for a component. Use the function **ProMechmeshctrlNumberingGet()** to access the numbering data for the mesh control item. Use the function **ProMechmeshctrlNumberingSet()** to set the numbering data for the mesh control item.
- PRO_MECH_MC_SUPPRESS—Specifies the components that are ignored while applying the mesh control to the assembly level. Use the function **ProMechmeshctrlSuppressGet()** to access the suppress data for the mesh control item. Use the function **ProMechmeshctrlSuppressSet()** to set the suppress data for the mesh control item.
- PRO_MECH_MC_AGEM_ISOLATE_EXCLUDE—Specifies points, edges, curves, and surfaces from the model to isolate during analysis. The function **ProMechmeshctrlAutogemisolateexcludedataGet()** returns the list of entities that AutoGEM can detect and isolate using mesh refinement. Use the function **ProMechmeshctrlAutogemisolateexcludedataSet()** to set the entities for AutoGEM isolation using mesh refinement.
- PRO_MECH_MC_AGEM_HARD_POINT—Specifies points, point features, or point patterns on the model to guide the AutoGEM mesh creation process. This control type has only references and no data associated with it. Use the function **ProMechmcautogemHardpointSet()** to set the AutoGEM mesh control item as hard point.
- PRO_MECH_MC_AGEM_HARD_CURVE—Specifies the datum curves on the model to guide the AutoGEM mesh creation process. This control type has only references and no data associated with it. Use the function **ProMechmcautogemHardcurveSet()** to set the AutoGEM mesh control item as hard curve.

- **PRO_MECH_MC_ALL**—Specifies all the types of mesh controls.

Note: If you are creating a new mesh control using this type of data, you should assign the model as a reference using the function **ProMechmeshcntrlReferencesSet()**.

The function **ProMechmeshcntrlReferencesGet()** returns the references for each of the mesh control type. Use the function **ProMechmeshcntrlReferencesSet()** to set the references used by the mesh control item.

Accessing AutoGEM Edge Distribution and Minimum Edge Mesh Control Data

The functions described in this section provide access to the data for the AutoGEM edge distribution Mesh Control.

Functions Introduced:

- **ProMechmcautogemedgedistrAlloc()**
- **ProMechmcautogemedgedistrNodesGet()**
- **ProMechmcautogemedgedistrNodesSet()**
- **ProMechmcautogemedgedistrRatioGet()**
- **ProMechmcautogemedgedistrRatioSet()**
- **ProMechmcautogemedgedistrStrictGet()**
- **ProMechmcautogemedgedistrStrictSet()**
- **ProMechmcautogemedgedistrFree()**
- **ProMechmcautogemminedgeAlloc()**
- **ProMechmcautogemminedgeEdgesGet()**
- **ProMechmcautogemminedgeEdgesSet()**
- **ProMechmcautogemminedgeLengthGet()**
- **ProMechmcautogemminedgeLengthSet()**
- **ProMechmcautogemminedgeFree()**

The function **ProMechmcautogemedgedistrNodesGet()** returns the number of nodes distributed along the selected edge. Use the function **ProMechmcautogemedgedistrNodesSet()** to set the number of nodes.

The function **ProMechmcautogemedgedistrRatioGet()** returns the aspect ratio defined while creating and editing elements. The aspect ratio is defined as the ratio of a length to the width of any surface in the model. Use the function **ProMechmcautogemedgedistrRatioSet()** to set the aspect ratio for the AutoGEM edge distribution mesh control data.

The function **ProMechmcautogemedgedistrStrictGet()** returns the value true, if the aspect ratio is within the maximum allowable range while creating and editing elements. Use the function **ProMechmcautogemedgedistrStrictSet()** to set the aspect ratio within the maximum allowable range.

Use the function **ProMechmcautogemedgedistrFree()** to free the memory containing the AutoGEM edge distribution data structure.

The function **ProMechmcautogemminedgeEdgesGet()** specifies the number of edges in a model for the AutoGEM minimum edge data. Use the function **ProMechmcautogemminedgeEdgesSet()** to set the number of edges.

The function **ProMechmcautogemminedgeLengthGet()** specifies the minimum length of any edge in the model. Use the function **ProMechmcautogemminedgeLengthSet()** to specify the minimum length of the edge for the AutoGEM min edge mesh control data.

Use the function **ProMechmcautogemminedgeFree()** to free the memory containing the AutoGEM minimum edge data structure.

Accessing the AutoGEM Edge Length by Curvature Mesh Control Data

The functions described in this section enable you to create a denser element mesh adjacent to areas such as curves, fillets, and holes, that are likely to have high stress, using the edge length by curve ratio.

Functions Introduced:

- **ProMechmcautogemedgelencrvAlloc()**
- **ProMechmcautogemedgelencrvRatioGet()**
- **ProMechmcautogemedgelencrvRatioSet()**
- **ProMechmcautogemedgelencrvIgnoreRadiusGet()**
- **ProMechmcautogemedgelencrvIgnoreRadiusSet()**

- **ProMechmcautogemedgelencrvMinradiusGet()**
- **ProMechmcautogemedgelencrvMinradiusSet()**
- **ProMechmcautogemedgelencrvFree()**

The function **ProMechmcautogemedgelencrvAlloc()** allocates memory for the Mechanica mesh control edge length by curvature data handle.

The function **ProMechmcautogemedgelencrvRatioGet()** returns the ratio of the expected edge lengths of mesh elements to the radius of the concave surface. Use the function **ProMechmcautogemedgelencrvRatioSet()** to set this ratio. Specify the ratio as a positive real number.

The function **ProMechmcautogemedgelencrvMinradiusGet()** returns the cut-off value for the radius of curvature below which the AutoGEM mesh control data will not be applied. Use the function **ProMechmcautogemedgelencrvMinradiusSet()** to set the cut-off value for the radius of curvature. This value can be specified in the current units of the model or as a percentage of the model size.

ProMechmcautogemedgelencrvIgnoreRadiusGet() returns true, if the curves having a radius of curvature lower than the cut-off value must be ignored by the AutoGEM mesh control data. Use the function

ProMechmcautogemedgelencrvIgnoreRadiusSet() to specify whether to ignore curves having a radius of curvature lower than the cut-off value.

Use the function **ProMechmcautogemedgelencrvFree()** to free the memory containing the edge length by curvature mesh control data structure.

Accessing the AutoGEM Maximum Element Size Mesh Control Data

The functions described in this section enable you to control the size of the elements created by the mesh generator for components, volumes, surfaces, edges, or curves.

Functions Introduced:

- **ProMechmcautogemelemsizeAlloc()**
- **ProMechmcautogemelemsizeSizeGet()**
- **ProMechmcautogemelemsizeSizeSet()**
- **ProMechmcautogemelemsizeFree()**

The function **ProMechmcautogemelemsizeAlloc()** allocates memory for the AutoGEM maximum element size mesh control data.

The function **ProMechmcautogemelemsizeSizeGet()** returns the maximum element size in the mesh. Use the function **ProMechmcautogemelemsizeSizeSet()** to set the element size.

Use the function **ProMechmcautogemelemsizeFree()** to free the memory containing the maximum element size mesh control data structure.

Accessing Edge Distribution Mesh Control Data

The functions described in this section provide access to the edge distribution mesh control data.

Functions Introduced:

- **ProMechmcedgedistrAlloc()**
- **ProMechmcedgedistrNodesGet()**
- **ProMechmcedgedistrNodesSet()**
- **ProMechmcedgedistrRatioGet()**
- **ProMechmcedgedistrRatioSet()**
- **ProMechmcedgedistrStrictGet()**
- **ProMechmcedgedistrStrictSet()**
- **ProMechmcedgedistrFree()**

The function **ProMechmcedgedistrNodesGet()** returns the minimum number of nodes that are distributed along the selected edge or curve. Use the function **ProMechmcedgedistrNodesSet()** to set the number of nodes for the mesh control data.

The function **ProMechmcedgedistrRatioGet()** returns the ratio of the first interval on the edge or curve to the last interval on the edge or curve. Use the function **ProMechmcedgedistrRatioSet()** to set the ratio for the mesh control data.

The function **ProMechmcedgedistrStrictGet()** returns a boolean flag indicating whether the number of nodes must be used exactly. Use the function **ProMechmcedgedistrStrictSet()** to set the maximum allowable nodes for the mesh control data.

Accessing AutoGEM Isolation Data

The functions in this section provide access to the entities that AutoGEM can detect and isolate using mesh refinement.

Functions Introduced:

- **ProMechmcautogemisolateexcludeAlloc()**
- **ProMechmcautogemisolateexcludeExcludeGet()**
- **ProMechmcautogemisolateexcludeExcludeSet()**
- **ProMechmcautogemisolateexcludeFree()**
- **ProMechmcautogemisolateexcludeShellIsolMaxSizeGet()**
- **ProMechmcautogemisolateexcludeShellIsolTypeGet()**
- **ProMechmcautogemisolateexcludeSolidIsolMaxSizeGet()**
- **ProMechmcautogemisolateexcludeSolidIsolTypeGet()**
- **ProMechmcautogemisolateexcludeShellIsolMaxSizeSet()**
- **ProMechmcautogemisolateexcludeShellIsolTypeSet()**
- **ProMechmcautogemisolateexcludeSolidIsolMaxSizeSet()**
- **ProMechmcautogemisolateexcludeSolidIsolTypeSet()**

The function **ProMechmcautogemisolateexcludeExcludeGet()** returns the value of the exclude boolean for the AutoGEM isolation for exclusion mesh control data. Use the function **ProMechmcautogemisolateexcludeExcludeSet()** to set the value of the exclude boolean for the AutoGEM isolation for exclusion mesh control data.

The function **ProMechmcautogemisolateexcludeShellIsolMaxSizeGet()** returns the value of maximum element size of isolation of shells for the AutoGEM isolation for exclusion mesh control data. Use the function **ProMechmcautogemisolateexcludeShellIsolMaxSizeSet()** to set the value of maximum element size of isolation of shells for the AutoGEM isolation.

The function

ProMechmcautogemisolateexcludeShellIsolTypeGet()

returns the value of shell isolation type for the AutoGEM isolation for exclusion mesh control data. Use the function

ProMechmcautogemisolateexcludeShellIsolTypeSet() to set the value of shell isolation type for the AutoGEM isolation for exclusion mesh control data.

The function

ProMechmcautogemisolateexcludeSolidIsolMaxSizeGet()

returns the value of maximum element size of isolation of solids for the AutoGEM isolation for exclusion mesh control data. Use the function

ProMechmcautogemisolateexcludeSolidIsolMaxSizeSet() to set the maximum element size of isolation of solids for the AutoGEM isolation.

The function

ProMechmcautogemisolateexcludeSolidIsolTypeGet()

returns the value of solid isolation type for the AutoGEM isolation for exclusion mesh control data. Use the function

ProMechmcautogemisolateexcludeSolidIsolTypeSet() to set the value of the solid isolation type for the AutoGEM isolation.

Accessing the Displacement Coordinate System Data

The functions described in this section provide read and write access to the displacement coordinate system control data.

Functions Introduced:

- **ProMechmcdisplacementcsysAlloc()**
- **ProMechmcdisplacementcsysCsysGet()**
- **ProMechmcdisplacementcsysCsysSet()**
- **ProMechmcdisplacementcsysFree()**

The function **ProMechmcdisplacementcsysCsysGet()** returns the coordinate system used to display the results for the nodes associated with points, edges, curves, or surfaces.

The function **ProMechmcdisplacementcsysCsysSet()** sets the co-ordinate system for the mesh control data.

Accessing the Mesh Control Element Size Data

Functions Introduced:

- **ProMechmcelemsizeSizeGet()**
- **ProMechmcelemsizeSizeSet()**
- **ProMechmcelemsizeFree()**

The function **ProMechmcelemsizeSizeGet()** returns the size of the elements for the mesh control data. Use the function **ProMechmcelemsizeSizeSet()** to set the size of the elements for the mesh control data.

Accessing the Mesh Control Shell Coordinate System Data

Functions Introduced:

- **ProMechmshellcsysCsysGet()**
- **ProMechmshellcsysCsysSet()**
- **ProMechmshellcsysDirectionGet()**
- **ProMechmshellcsysDirectionSet()**
- **ProMechmshellcsysFree()**

The function **ProMechmshellcsysCsysGet()** returns the coordinate system specified for the mesh control data. Use the function **ProMechmshellcsysCsysSet()** to set the coordinate system specified for the mesh control data.

The function **ProMechmshellcsysDirectionGet()** returns the positive direction along the x, y, and z axis. Use the function **ProMechmshellcsysDirectionSet()** to set the positive direction along the x, y, and z axis.

Accessing the Mesh Control Hard Point Data

Functions Introduced:

- **ProMechmchardpntNodeGet()**
- **ProMechmchardpntNodeSet()**
- **ProMechmchardpntFree()**

The function **ProMechmchardpntNodeGet()** returns the id of the node that is defined as a hard point. Use the function **ProMechmchardpntNodeSet()** to set the id of the node for the mesh control data.

Accessing the Mesh Control ID Offset Data

Functions Introduced:

- **ProMechmcidoffsetOffsetGet()**
- **ProMechmcidoffsetOffsetSet()**
- **ProMechmcidoffsetFree()**

The function **ProMechmcidoffsetOffsetGet()** returns a positive integer value to be added to the IDs of each node, element, and local mesh entity. Use the function **ProMechmcidoffsetOffsetSet()** to set the offset value for the mesh control data.

Accessing the Mesh Control Numbering Data

Functions Introduced:

- **ProMechmcnumberingFirstGet()**
- **ProMechmcnumberingFirstSet()**
- **ProMechmcnumberingIncrementGet()**
- **ProMechmcnumberingIncrementSet()**
- **ProMechmcnumberingLastGet()**
- **ProMechmcnumberingLastSet()**
- **ProMechmcnumberingFree()**

The function **ProMechmcnumberingFirstGet()** returns the first ID for the nodes, elements, and local mesh entities. Use the function **ProMechmcnumberingFirstSet()** to set the first value of the mesh control data.

The function **ProMechmcnumberingIncrementGet()** returns the increment id for the nodes, elements, and local mesh entities. Use the function **ProMechmcnumberingIncrementSet()** to set the increment value for the mesh control data.

The function **ProMechmcnumberingLastGet()** returns the last ID for the nodes, elements, and local mesh entities. Use the function **ProMechmcnumberingLastSet()** to set the last value for the mesh control data.

Accessing the Suppressed Mesh Control Data

Functions Introduced:

- **ProMechmcsuppressTypeGet()**
- **ProMechmcsuppressTypeSet()**
- **ProMechmcsuppressFree()**

The function **ProMechmcsuppressTypeGet()** returns the type of mesh control data that should be suppressed. Use the function **ProMechmcsuppressTypeSet()** to set the type of data to be suppressed.

Welds

Welds are used to bridge gaps that are formed during shell compression between plates that have been mated because they touch or overlap. Welds use the ProType field in the ProMechitem structure as PRO_SIMULATION_WELD.

Functions Introduced:

- **ProMechweldReferencesGet()**
- **ProMechweldReferencesSet()**
- **ProMechweldTypeGet()**
- **ProMechweldperimeterAlloc()**
- **ProMechweldPerimeterdataGet()**
- **ProMechweldPerimeterdataSet()**
- **ProMechweldperimeterEdgesGet()**
- **ProMechweldperimeterEdgesSet()**
- **ProMechweldperimeterFree()**
- **ProMechweldedgeAlloc()**
- **ProMechweldedgeEdgeGet()**
- **ProMechweldedgeEdgeSet()**
- **ProMechweldedgeThicknessGet()**

- **ProMechweldedgeThicknessSet()**
- **ProMechweldedgeMaterialidGet()**
- **ProMechweldedgeMaterialidSet()**
- **ProMechweldedgeFree()**
- **ProMechweldedgeProarrayFree()**
- **ProMechweldspotAlloc()**
- **ProMechweldSpotdataGet()**
- **ProMechweldSpotdataSet()**
- **ProMechweldspotPointsGet()**
- **ProMechweldspotPntsSet()**
- **ProMechweldspotDiameterGet()**
- **ProMechweldspotDiameterSet()**
- **ProMechweldspotMaterialIdGet()**
- **ProMechweldspotMaterialIdSet()**
- **ProMechweldspotFree()**
- **ProMechweldendSet()**
- **ProMechweldEnddataGet()**
- **ProMechweldEnddataSet()**
- **ProMechweldendAlloc()**
- **ProMechweldendTypeGet()**
- **ProMechweldendTypeSet()**
- **ProMechweldendExtendAdjacentSurfacesGet()**
- **ProMechweldendExtendAdjacentSurfacesSet()**
- **ProMechweldendFree()**
- **ProMechweldFeaturedataGet()**
- **ProMechweldFeaturedataSet()**
- **ProMechweldfeatureAlloc()**
- **ProMechweldfeatureMaterialidGet()**
- **ProMechweldfeatureOverrideflagGet()**
- **ProMechweldfeatureThicknessGet()**
- **ProMechweldedgeMaterialidSet()**

- **ProMechweldfeatureOverrideflagSet()**
- **ProMechweldfeatureThicknessSet()**
- **ProMechweldfeatureFree()**

The function **ProMechweldReferencesGet()** returns the geometric entities selected to create the weld. Use the function **ProMechweldReferencesSet()** to set the references for the specified weld.

The function **ProMechweldTypeGet()** returns the type of weld used to connect the gaps. The types of weld are:

- **PRO_MECH_WELD_PERIMETER**—Specifies a perimeter weld. Perimeter welds are used to connect parallel plates along the perimeter of one of the plates in an assembly model.
- **PRO_MECH_WELD_END**—Specifies an end weld. End welds are used to connect plates in assembly models.
- **PRO_MECH_WELD_SPOT**—Specifies a spot weld. Spot welds are used to connect two parallel surfaces at the specified datum point.
- **PRO_MECH_WELD_FEAT**—Specifies a Weld Feature. This weld connection can be used to select Fillet and Groove types of weld connections for inclusion in a mid-surface compressed model.

The function **ProMechweldPerimeterdataGet()** provides access to the perimeter weld data structure. Use the function **ProMechweldPerimeterdataSet()** to set the perimeter weld data structure.

Use the function **ProMechweldperimeterFree()** to free the memory contained in the perimeter weld data structure.

The function **ProMechweldperimeterEdgesGet()** returns the perimeter edges of the top plate to be connected to the base plate. Use the function **ProMechweldperimeterEdgesSet()** to set the perimeter edges in the perimeter weld data.

The function **ProMechweldedgeEdgeGet()** returns the id of the edge selected to create the perimeter weld. Use the function **ProMechweldedgeEdgeSet()** to set the id of the edge.

The function **ProMechweldedgeThicknessGet()** returns the thickness of the weld on the selected edge in the perimeter weld. Use the function **ProMechweldedgeThicknessSet()** to set the thickness of the weld edge data.

The function **ProMechweldedgeMaterialidGet()** returns the type of material used in shell elements created on the Perimeter Weld surfaces.

Use the function **ProMechweldedgeMaterialidSet()** to select the material for the shell elements created on the Perimeter Weld surfaces. Specify the material ID as the input parameter of this function. In addition to the materials already present in the model, the input can be the material ID of one of the following surfaces:

- **PRO_MECH_WELD_MTL_BASE**—Specifies the base surface to which the weld extends.
- **PRO_MECH_WELD_MTL_DOUBLER**—Specifies the doubler surface on which the Perimeter weld is placed.

The functions **ProMechweldedgeFree()** and **ProMechweldedgeProarrayFree()** free the memory containing the edge data structure.

The function **ProMechweldperimeterFree()** frees the memory containing the data structure for the perimeter weld.

The function **ProMechweldSpotdataGet()** provides access to the spot weld data structure. Use the function **ProMechweldSpotdataSet()** to assign data to the spot weld data structure.

The function **ProMechweldspotPointsGet()** returns the datum points at which the spot weld is located. Use the function **ProMechweldspotPntsSet()** to set the points for the spot weld data.

The function **ProMechweldspotDiameterGet()** returns the diameter for the spot weld. Use the function **ProMechweldspotDiameterSet()** to set the diameter for the spot weld data.

The function **ProMechweldspotMaterialIdGet()** returns the material for the spot weld. Use the function **ProMechweldspotMaterialIdSet()** to set the material for the spot weld.

Use the function **ProMechweldspotFree()** to free the memory contained in the spot weld data structure.

Use the function **ProMechweldendSet()** to set the weld to be an end weld.

The function **ProMechweldEnddataGet()** returns the end weld data structure.

Use the function **ProMechweldendAlloc()** to allocate memory for the end weld data handle.

Use the method **ProMechweldEnddataSet()** to set the end weld data structure.

The function **ProMechweldendTypeGet()** returns the type of end weld defined between different surfaces. The valid types of end welds are:

- **PRO_MECH_WELD_END_SINGLE_TO_SINGLE**—Specifies a weld defined between a side surface and a shell or shell paired surface.
- **PRO_MECH_WELD_END_MANY_TO_SINGLE**—Specifies a weld defined from a solid surface to a solid surface.
- **PRO_MECH_WELD_END_SINGLE_TO_MANY**—Specifies a weld surface defined from a solid surface to a shell surface.

Use the function **ProMechweldendTypeSet()** to set the type of end welds.

The function **ProMechweldendExtendAdjacentSurfacesGet()** specifies whether or not surfaces adjacent to the selected source surface will be extended to the target surface to create the weld.

Use the function

ProMechweldendExtendAdjacentSurfacesSet() to control the placement of the weld on the extension of the adjacent surfaces.

Use the function **ProMechweldendFree()** to free the memory of the end weld data structure.

The function **ProMechweldFeaturedataGet()** returns the Weld Feature data structure.

Use the function **ProMechweldfeatureAlloc()** to allocate memory for the Weld Feature data structure.

Use the function **ProMechweldFeaturedataSet()** to set the Weld Feature data structure.

The function **ProMechweldfeatureOverrideflagGet()** specifies if the material and thickness properties that will be applied to the shell idealizations resulting from the weld definitions are inherited from the Weld Feature or not.

Specify the input value of the function

ProMechweldfeatureOverrideflagSet() as true to specify the properties for the shell idealizations. Specify the value as false to inherit the properties from the Weld Feature.

You can use the following functions to access and modify the material property and thickness of the Weld Feature only if you specify the input value as true for the function

ProMechweldfeatureOverrideflagSet() or you select the option **Override Weld Feature Settings** from the Pro/MECHANICA user interface.

The function **ProMechweldfeatureMaterialidGet()** returns the material type used for the Weld Feature. Use the function **ProMechweldfeatureMaterialidSet()** to set the material for the shell idealizations. Specify the material ID as the input parameter of this function.

The function **ProMechweldfeatureThicknessGet()** returns the thickness of the Weld Feature. Use the function **ProMechweldfeatureThicknessSet()** to set the thickness of the shell idealizations.

Use the function **ProMechweldfeatureFree()** to free the memory contained in the Weld Feature data structure.

Mechanica: Mechanica Geometry

In Pro/ENGINEER, when analysis is performed on a model (in either FEM or Mechanica), the Pro/ENGINEER geometry that is seen on the screen is processed to create the Mechanica Geometry. This special temporary geometry is more suitable for analysis than the standard Pro/ENGINEER geometry.

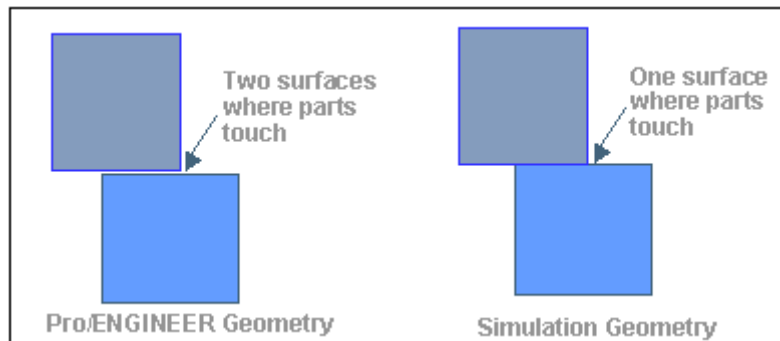
Topic	Page
Introduction	62 - 2
Obtaining Mechanica Geometry from Pro/TOOLKIT	62 - 4
To Create a Surface Region Feature	62 - 18

Introduction

Mechanica Geometry differs from that of standard Pro/ENGINEER geometry in several ways.

- Mechanics geometry provides a non-manifold representation of assembly models.

Figure 62-1: Non-manifold geometry

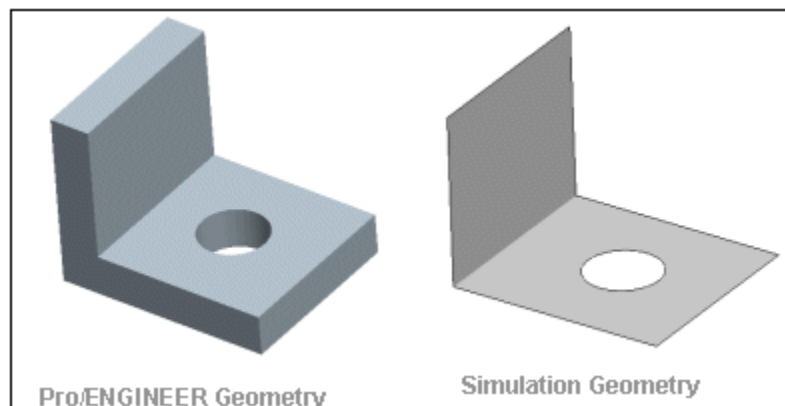


In assemblies, when two parts are mated, there exist two surfaces where the parts are mated, each of which does not know the existence of the other.

For analysis applications (particularly in meshing), you want the geometry to have only one surface in this mated area

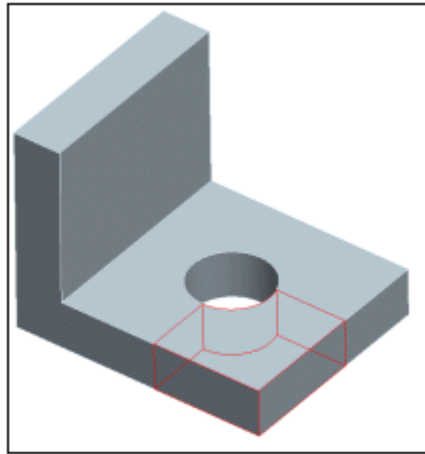
- Mechanics geometry provides mid-surface geometry for models which have shells defined.

Figure 62-2: Mid-surface geometry



- Mechanics geometry provides volume and surface region information.

Figure 62-3: "Tagged" region



Mechanica:
Mechanica Geometry

Without access to the Mechanics geometry, it is difficult to use the information on volume and surface region.

The Mechanics geometry identifies which surfaces are "duplicated" at the boundaries within the solid. This makes it easy for external applications to re-create the non-manifold solid.

- Mechanics Geometry model is devoid of the undesirable features (from the point of view of analysis) present in the Pro/ENGINEER model such as
 - Small, localized misalignments in the geometry
 - Cusps
 - Very small sliver surfaces

Obtaining Mechanica Geometry from Pro/TOOLKIT

Pro/ENGINEER represents the top level access to Mechanica geometry as an opaque handle called *ProMechModel*. The *ProMechModel* handle is generated upon request and can be accessed until it is freed by the application.

There are several modes which can be used to generate the model and its geometry:

- PRO_MECH_MODEL_SOLID signifies solid surfaces only. Shells will be ignored.
- PRO_MECH_MODEL_SHELL signifies shell-compressed surfaces. Non shell-compressed surfaces will not be included.
- PRO_MECH_MODEL_MIXED signifies both solid and shell-compressed surfaces.
- PRO_MECH_MODEL_BOUNDARY signifies shell surfaces occurring on the solid boundary.
- PRO_MECH_MODEL_QUILT signifies shell surfaces on quilts. This option requires that quilts be specified as additional entities to be processed.
- PRO_MECH_MODEL_BAR signifies bars on given datum curves. This option requires that curves be specified as additional entities to be processed.

It is not always necessary to have a Pro/ENGINEER Mechanica license to generate the Mechanica geometry. However, some situations do require the license:

- Use of the options PRO_MECH_MODEL_SHELL, PRO_MECH_MODEL_MIXED and PRO_MECH_MODEL_BOUNDARY is not permitted unless Pro/ENGINEER Mechanica is active. Please see the description for **ProMechanicaIsActive()** in the chapter Accessing Mechanica Items.
- If any component of the model contains surface or volume region feature, Pro/ENGINEER Mechanica must be active to access the geometry of these regions.

Functions Introduced:

- **ProMechmodeldataAlloc()**
- **ProMechmodeldataTypeSet()**
- **ProMechmodeldataEntitiesSet()**
- **ProMechmodeldataPreserveidsSet()**
- **ProMechmodeldataMergeCoincSolidEdgesSet()**
- **ProMechmodelGet()**
- **ProMechmodelFree()**
- **ProMechmodeldataFree()**

Use the function **ProMechmodeldataAlloc()** to allocate an input data structure for generation of Mechanica geometry.

Use the function **ProMechmodeldataTypeSet()** to assign the type of model to be generated.

Use the function **ProMechmodeldataEntitiesSet()** to assign the additional datum points, quilts, and curves to be included in the geometric processing.

Use the function **ProMechmodeldataPreserveidsSet()** to set the preservation of IDs. In the event of the "Preserve IDs" flag being set, Pro/ENGINEER Mechanica tries to preserve the IDs of surfaces and edges in Mechanica geometry between different calls to **ProMechmodelGet()**. This happens even if the original geometry of the model has changed slightly, but the success of this attempt to preserve geometry IDs is not always guaranteed.

Use the function **ProMechmodeldataMergeCoincSolidEdgesSet()** to set the flag that determines whether coincident solid edges will be merged. By default, the flag is PRO_B_FALSE, that is, the solid edges will not be merged.

Use the function **ProMechmodelGet()** to obtain the root handle of a Mechanica geometry model. The input arguments of this function are:

- *solid*—signifies the root solid model (part or assembly).
- *data*—signifies the options for the generation of geometry.

The output arguments are as follows:

- *mech_model* signifies the root handle of the generated geometry.

- *status* signifies the status of generation and can have the following values:
 - PRO_MECH_MODEL_SUCCESS signifies success.
 - PRO_MECH_MODEL_GENERAL_FAILURE signifies general failure.
 - PRO_MECH_MODEL_USER_INTERRUPT signifies that the user interrupted the process before it completed.
 - PRO_MECH_MODEL_SHELL_NO_PAIRS signifies that no shell pairs were defined (for options that used shell compression).
 - PRO_MECH_MODEL_SHELL_SOME_PAIRS signifies that some paired and some unpaired surfaces exist.(for options that used shell compression).

Use the function **ProMechmodeldataFree()** to free a data handle used for generation of a Mechanica geometry model. Use the function **ProMechmodelFree()** to free a Mechanica model handle. This invalidates all geometric entities obtained from this handle.

Accessing the ProMechModel

Functions Introduced:

- **ProMechmodelMdlGet()**
- **ProMechmodelSolidVisit()**
- **ProMechmodelPointVisit()**
- **ProMechmodelCurveVisit()**
- **ProMechmodelToleranceGet()**

The function **ProMechmodelMdlGet()** can be used to obtain the root solid model used for generation of this Pro/MECANICA model.

The function **ProMechmodelSolidVisit()** visits the solid volumes that make up a Mechanica geometry model.

The function **ProMechmodelPointVisit()** visits the datum points that are included in a Mechanica geometry model.

Note: only datum points that have Mechanica items referencing them will be included unless you pass the additional points as input from **ProMechmodeldataEntitiesSet()**.

The function **ProMechmodelCurveVisit()** visits the composite curves that are included in a Pro/MECANICA geometry model.

Note: only datum points that have Mechanica items referencing them will be included unless you pass the additional points as input from **ProMechmodeldataEntitiesSet()**.

Use the function **ProMechmodelToleranceGet()** to obtain the overall tolerance or epsilon value used for the preparation of the Pro/MECANICA geometry.

Accessing the ProMechSolid

An opaque handle called a *ProMechSolid* represents a solid volume member of the model (typically, an assembly member).

Functions Introduced:

- **ProMechsolidVisitAction()**
- **ProMechsolidFilterAction()**
- **ProMechsolidIdGet()**
- **ProMechsolidModelGet()**
- **ProMechsolidTypeGet()**
- **ProMechsolidAsmcomppathGet()**
- **ProMechsolidSurfaceVisit()**
- **ProMechsolidEdgeVisit()**
- **ProMechsolidVertexVisit()**

The function types **ProMechsolidVisitAction()** and **ProMechsolidFilterAction()** are used as arguments to functions that visit *ProMechSolid* objects.

Use the function **ProMechsolidIdGet()** to obtain the ID of the solid volume. This ID is an index in the array of generated solid volumes and is not persistent among different calls to **ProMechmodelCreate()**.

Use the function **ProMechsolidModelGet()** to obtain the root Mechanica geometry model for this solid model.

Use the function **ProMechsolidTypeGet()** to obtain the solid volume type.

Use the function **ProMechsolidAsmcomppathGet()** to obtain the assembly component path from the top level assembly to this solid.

Use the function **ProMechsolidSurfaceVisit()** to visit the surfaces that are included in a given solid volume.

Use the function **ProMechsolidEdgeVisit()** to visit the edges that are included in a given solid volume.

Use the function **ProMechsolidVertexVisit()** to visit the vertices that are included in a given solid volume.

Accessing Mechanica ProMechSurface

An opaque handle called a *ProMechSurface* represents a Mechanica geometry surface. A surface can be of the following types:

- PRO_MECH_FACE_SOLID—A solid surface
- PRO_MECH_FACE_SHELL—A shell (mid-plane) surface
- PRO_MECH_FACE_SIDE—A shell side surface
- PRO_MECH_FACE_SHELL_QLT—A quilt surface
- PRO_MECH_FACE_BOUNDARY—A solid boundary surface
- PRO_MECH_FACE_PERIM_WELD—A generated surface for a perimeter weld object

Functions Introduced:

- **ProMechsurfaceVisitAction()**
- **ProMechsurfaceFilterAction()**
- **ProMechsurfaceIdGet()**
- **ProMechsurfaceOwnerGet()**
- **ProMechsurfaceTypeGet()**
- **ProMechsurfaceContourVisit()**
- **ProMechsurfaceEdgeVisit()**
- **ProMechsurfaceVertexVisit()**
- **ProMechsurfaceIncontactfacesGet()**
- **ProMechsurfaceAncestorsGet()**

The function types **ProMechsurfaceVisitAction()** and **ProMechsurfaceFilterAction()** are used as arguments to functions that visit *ProMechSurface* objects.

Use the function **ProMechsurfaceIdGet()** to obtain the surface ID of the surface.

Note: This ID is not persistent and is not related to the Pro/ENGINEER surface ID.

Use the function **ProMechsurfaceOwnerGet()** to obtain the owner *ProMechsolid* of the surface.

Use the function **ProMechsurfaceTypeGet()** to obtain the type of surface.

Use the function **ProMechsurfaceContourVisit()** to visit the contours that are included in a given surface.

Use the function **ProMechsurfaceEdgeVisit()** to visit the edges that are included in a given surface.

Use the function **ProMechsurfaceVertexVisit()** to visit vertices that are included in a given surface.

Use the function **ProMechsurfaceAncestorsGet()** to obtain the ancestor surfaces for the given surface. These are the actual Pro/ENGINEER geometry surfaces used to construct this Mechanica surface.

Geometry Evaluation of ProMechSurface

For information about how Pro/TOOLKIT represents surface geometry, see the chapter on Core: 3D Geometry and the appendix, Element Trees: References.

Functions Introduced:

- **ProMechsurfaceParamEval()**
- **ProMechsurfaceUvpntVerify()**
- **ProMechsurfaceTessellationGet()**
- **ProMechsurfaceDataGet()**
- **ProMechsurfaceToNURBS()**
- **ProMechsurfaceTransformGet()**
- **ProMechsurfaceThicknessEval()**

Use the function **ProMechsurfaceParamEval()** to find the corresponding UV point on the Mechanica geometry surface on the basis of the XYZ point.

Use the function **ProMechsurfaceUvpntVerify()** to verify whether the specified UV point lies within the boundaries of the Mechanica surface.

Use the function **ProMechsurfaceTessellationGet()** to calculate the tessellation for the provided Mechanica surface.

Use the function **ProMechsurfaceDataGet()** to obtain the geometric representation of the surface.

Use the function **ProMechsurfaceToNURBS()** to obtain the NURBS representation of the Mechanica surface.

Use the function **ProMechsurfaceTransformGet()** to obtain UV transform between the two surfaces in contact.

Use the function **ProMechsurfaceThicknessEval()** to obtain the thickness of the shell Mechanica surface at the given UV point.

Accessing ProMechContour

An opaque handle called a *ProMechContour* represents a contour member of the model. See the chapter on Core: 3D Geometry for a discussion of contours.

Functions Introduced:

- **ProMechcontourVisitAction()**
- **ProMechcontourFilterAction()**
- **ProMechcontourIdGet()**
- **ProMechcontourSurfaceGet()**
- **ProMechcontourTraversalGet()**
- **ProMechcontourEdgeVisit()**
- **ProMechcontourUvpntVerify()**
- **ProMechcontourAreaEval()**
- **ProMechcontourContainingContourGet()**

The function types **ProMechcontourVisitAction()** and **ProMechcontourFilterAction()** are used as arguments to functions that visit *ProMechContour* objects.

Use the function **ProMechcontourIdGet()** to obtain the ID for a given contour. This ID is unique within the surface that owns the contour.

Use the function **ProMechcontourSurfaceGet()** to obtain the surface that contains the contour.

Use the function **ProMechcontourTraversalGet()** to obtain the contour traversal.

Use the function **ProMechcontourEdgeVisit()** to visit the edges that make up a contour.

Use the function **ProMechcontourUvpntVerify()** to verify whether the specified UV point lies within the given Pro/MECHNICA contour.

Use the function **ProMechcontourAreaEval()** to find the surface area inside the given outer contour, accounting for internal voids.

The function **ProMechcontourContainingContourGet()** returns the containing contour for a Mechanica contour object.

Accessing ProMechEdge

An opaque handle called a *ProMechEdge* represents an edge member of the model.

Functions Introduced:

- **ProMechedgeVisitAction()**
- **ProMechedgeFilterAction()**
- **ProMechedgeIdGet()**
- **ProMechedgeOwnerGet()**
- **ProMechedgeSurfaceVisit()**
- **ProMechedgeContourVisit()**
- **ProMechedgeEndpointsGet()**
- **ProMechedgeIncontactedgesGet()**
- **ProMechedgeAncestorsGet()**

The function types **ProMechedgeVisitAction()** and **ProMechedgeFilterAction()** are used as arguments to functions that visit *ProMechEdge* objects.

Use the function **ProMechedgeIdGet()** to obtain the ID of the given edge.

Note: This ID is not persistent and is not related to the Pro/ENGINEER edge ID.

Use the function **ProMechedgeOwnerGet()** to obtain the owner *ProMechSolid* of the given edge.

Use the function **ProMechedgeSurfaceVisit()** to visit the surfaces that share this edge.

Use the function **ProMechedgeContourVisit()** to visit the contours that contain this edge.

Use the function **ProMechedgeEndpointsGet()** to obtain the endpoints of the given edge.

Use the function **ProMechedgeIncontactedgesGet()** to obtain the list of edges that are in contact with the given edge.

Use the function **ProMechedgeAncestorsGet()** to obtain the ancestor edges for the given edge. These are the actual Pro/ENGINEER geometry edges used to construct this Mechanica edge.

Geometry Evaluation of ProMechEdge

For information about how Pro/TOOLKIT represents edge geometry, see the chapter on Core: 3D Geometry and the appendix, Element Trees: References.

Functions Introduced:

- **ProMechedgeUvdataEval()**
- **ProMechedgeXyzdataEval()**
- **ProMechedgeParamEval()**
- **ProMechedgeLengthEval()**
- **ProMechedgeLengthT1T2Eval()**
- **ProMechedgeParamByLengthEval()**
- **ProMechedgeTessellationGet()**
- **ProMechedgeDataGet()**
- **ProMechedgeToNURBS()**
- **ProMechedgeDirectionGet()**
- **ProMechedgeReldirGet()**

Use the function **ProMechedgeUvdataEval()** to evaluate the Mechanica edge in the UV space of the given surface.

Use the function **ProMechedgeXyzdataEval()** to evaluate the Mechanica edge parameter point in XYZ space.

Use the function **ProMechedgeParamEval()** to find the corresponding normalized parameter on the Mechanica edge by XYZ point.

Use the function **ProMechedgeLengthEval()** to obtain the length of the edge.

Use the function **ProMechedgeLengthT1T2Eval()** to find the length of the Mechanica edge between the given parameters.

Use the function **ProMechedgeParamByLengthEval()** to find the parameter of the point located at the given length from the given parameter.

Use the function **ProMechedgeTessellationGet()** to get the edge tessellation for the Mechanica edges.

Use the function **ProMechedgeToNURBS()** to obtain the NURBs representation of the Mechanica edge.

Use the function **ProMechedgeReldirGet()** to obtain the relative direction of two Mechanica edges in contact.

Use the function **ProMechedgeDataGet()** to obtain the geometric representation of the edge.

Use the function **ProMechedgeDirectionGet()** to obtain the edge direction with respect to the given contour.

Accessing ProMechVertex

An opaque handle called a *ProMechVertex* represents a vertex member of the model.

Functions Introduced:

- **ProMechvertexVisitAction()**
- **ProMechvertexFilterAction()**
- **ProMechvertexIdGet()**
- **ProMechvertexOwnerGet()**
- **ProMechvertexPointGet()**
- **ProMechvertexSurfaceVisit()**
- **ProMechvertexEdgeVisit()**
- **ProMechvertexIncontactverticesGet()**

The function types **ProMechvertexVisitAction()** and **ProMechvertexFilterAction()** are used as arguments to functions that visit *ProMechVertex* objects.

Use the function **ProMechvertexIdGet()** to obtain the ID of the given vertex.

Use the function **ProMechvertexOwnerGet()** to obtain the volume that owns this vertex.

Use the function **ProMechvertexPointGet()** to obtain the coordinate point for a given vertex.

Use the function **ProMechvertexSurfaceVisit()** to visit the surfaces that include this vertex.

Use the function **ProMechvertexEdgeVisit()** to visit the edges that contain this vertex.

Use the function **ProMechvertexIncontactverticesGet()** to obtain the list of vertices that are in contact with the given vertex.

Accessing ProMechPoint

An opaque handle called a *ProMechPoint* represents a datum point member of the model. By default, the Mechanics geometry will include only those points which have Mechanics loads or other items referencing them; you can generate additional points by including them as inputs to **ProMechmodeldataEntitiesSet()**.

Functions Introduced:

- **ProMechpointVisitAction()**
- **ProMechpointFilterAction()**
- **ProMechpointIdGet()**
- **ProMechpointOwnerGet()**
- **ProMechpointPointGet()**
- **ProMechpointPlacemementtypeGet()**
- **ProMechpointPlacementsurfaceGet()**
- **ProMechpointPlacemementedgeGet()**
- **ProMechpointPlacemementvertexGet()**
- **ProMechpointAncestorsGet()**

The function types **ProMechpointVisitAction()** and **ProMechpointFilterAction()** are used as arguments to functions that visit *ProMechPoint* objects.

Use the function **ProMechpointIdGet()** to obtain the ID of the point.

Note: This ID is not persistent and is not related to the Pro/ENGINEER point ID.

Use the function **ProMechpointOwnerGet()** to obtain the owner model of the given point.

Use the function **ProMechpointPointGet()** to obtain the coordinates of the point.

Use the function **ProMechpointPlacementtypeGet()** to obtain the placement type for the point. Following are list of possible placement types:

- PRO_MECH_PNT_FREE signifies that the point is not attached to a solid or shell.
- PRO_MECH_PNT_FACE signifies that the point lies on a *ProMechSurface*.
- PRO_MECH_PNT_EDGE signifies that the point lies on a *ProMechEdge*.
- PRO_MECH_PNT_VERTEX signifies that the point lies on a *ProMechVertex*.

Use the function **ProMechpointPlacementsurfaceGet()** to obtain the placement surface, if the placement type is PRO_MECH_POINT_FACE.

Use the function **ProMechpointPlacementedgeGet()** to obtain the placement edge, if placement type is PRO_MECH_POINT_EDGE.

Use the function **ProMechpointPlacementvertexGet()** to obtain the placement vertex, if the placement type is PRO_MECH_POINT_VERTEX.

Use the function **ProMechpointAncestorsGet()** to obtain the ancestor points for the given point. These are the actual Pro/ENGINEER geometry points used to construct this Mechanica point.

Accessing ProMechCompositeCurve

An opaque handle called a *ProMechCompositeCurve* represents a composite curve member of the model. By default, the Mechanica geometry will include only those curves which have Mechanica loads or other items referencing them. You can generate additional curves by including them as inputs to **ProMechmodeldataEntitiesSet()**.

Functions Introduced:

- **ProMechcompositecurveVisitAction()**
- **ProMechcompositecurveFilterAction()**
- **ProMechcompositecurveIdGet()**
- **ProMechcompositecurveOwnerGet()**
- **ProMechcompositecurveCurveVisit()**
- **ProMechcompositecurveAncestorsGet()**

The function types **ProMechcompositecurveVisitAction()** and **ProMechcompositecurveFilterAction()** are used as arguments to functions that visit *ProMechCompositeCurve* objects.

Use the function **ProMechcompositecurveIdGet()** to obtain the ID of the composite curve.

Note: This ID is not persistent and is not related to the Pro/ENGINEER composite curve ID.

Use the function **ProMechcompositecurveOwnerGet()** to obtain the owner model of the composite curve.

Use the function **ProMechcompositecurveCurveVisit()** to visit the curves that make up this composite curve.

Use the function **ProMechcompositecurveAncestorsGet()** to obtain the ancestor curves for the given curve. These are the actual Pro/ENGINEER geometry curves used to construct this Mechanical curve.

Accessing ProMechCurve

An opaque handle called a *ProMechCurve* represents a curve member of the model. A curve is always the child of a composite curve.

Functions Introduced:

- **ProMechcurveVisitAction()**
- **ProMechcurveFilterAction()**
- **ProMechcurveTypeGet()**
- **ProMechcurveEdgeGet()**
- **ProMechcurveParentGet()**
- **ProMechcurveParamGet()**
- **ProMechcurveEndpointsGet()**

The function types **ProMechcurveVisitAction()** and **ProMechcurveFilterAction()** are used as arguments to functions that visit *ProMechCurve* objects.

Use the function **ProMechcurveTypeGet()** to obtain the curve's type. Curves may be of the following types:

- PRO_MECH_SEGMENT_FREE
- PRO_MECH_SEGMENT_ON_EDGE

Use the function **ProMechcurveEdgeGet()** to obtain the edge, if the curve type is PRO_MECH_CURVE_EDGE.

Use the function **ProMechcurveParentGet()** to obtain the parent composite curve for this curve.

Use the function **ProMechcurveParamGet()** to obtain the parameter along the parent composite curve at which this curve begins.

Use the function **ProMechcurveEndpointsGet()** to obtain the endpoints of the curve.

Geometry Evaluation of ProMechCurves

For information about how Pro/TOOLKIT represents curve geometry, see the chapter on Core: 3D Geometry and the appendix, Element Trees: References.

Functions Introduced:

- **ProMechcurveXyzdataEval()**
- **ProMechcurveParamEval()**
- **ProMechcurveLengthEval()**
- **ProMechcurveLengthT1T2Eval()**
- **ProMechcurveParamByLengthEval()**
- **ProMechcurveDataGet()**
- **ProMechcurveToNURBS()**

Use the function **ProMechcurveXyzdataEval()** evaluates the Mechanica curve parameter point in the XYZ point.

Use the function **ProMechcurveParamEval()** to find the corresponding normalized parameter on the Mechanica curve by XYZ point.

Use the function **ProMechcurveLengthEval()** to obtain the length of the curve.

Use the function **ProMechcurveLengthT1T2Eval()** to find the length of the Mechanics curve between the given parameters.

Use the function **ProMechcurveParamByLengthEval()** to find the parameter of the point located at the given length from the given parameter.

Use the function **ProMechcurveDataGet()** to obtain the geometric representation of the edge.

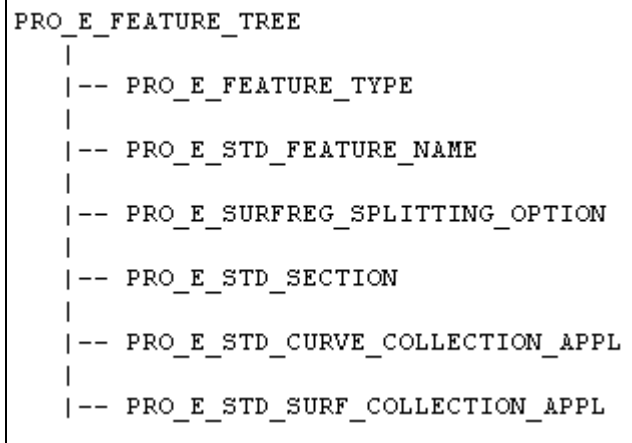
Use the function **ProMechcurveToNURBS()** to obtain the NURBS representation of the Mechanics curve.

To Create a Surface Region Feature

This section describes the use the header file *ProSurfReg.h* to create a surface region feature programmatically. The chapter 'Element Trees: Principles of Feature Creation' provides the necessary background for creating features; we recommend you read that material first.

Note: The Surface Region feature is available and can be regenerated only in the Mechanics environment.

The following figure shows the element tree for the Surface Region feature.



The Surface Region feature element tree contains no non-standard element types. The following table describes special information about the elements in this tree.

Element ID	Value
PRO_E_FEATURE_TYPE	PRO_FEAT_SPLIT_SURF
PRO_E_STD_FEATURE_NAME	Specifies the name of the Surface Region feature. The default value is "Surface Region". This element is optional.
PRO_E_SURFREG_SPLITTING_OPTION	Specifies the method to define the surface contour. Valid values are: PRO_SURFREG_SKETCH—Split the surface using a sketch. PRO_SURFREG_CHAIN—Split the surface using a chain.
PRO_E_STD_SECTION	Specifies a 2D section or a sketched section. Refer to the section 'Creating Features Containing Sections' for details on how to create features that contain sketched sections. Refer to the section 'Creating Section Models' for details on creating 2D sections.
PRO_E_STD_CURVE_COLLECTION_APPL	Specifies a collection of selected curves or edges or both to copy.
PRO_E_STD_SURF_COLLECTION_APPL	Specifies a collection of selected surfaces to copy.

Mechanica: Finite Element Modeling (FEM)

This chapter contains descriptions of the Pro/TOOLKIT functions that support Pro/ENGINEER Finite Element Modeling (FEM).

Topic	Page
Overview	63 - 2
Exporting an FEA Mesh	63 - 2
Getting Constraint Case Names	63 - 5
Getting Mesh Controls	63 - 5
Getting Constraints	63 - 7
Getting Bar Elements and Contacts	63 - 10
Getting Mass Elements	63 - 11
Getting Shell Pairs	63 - 12

Overview

The Finite Element Modeling (FEM) functions in this chapter are designed to give you access to data generated by the Pro/MESH module of Pro/ENGINEER. You can do the following:

- Export a Pro/MESH output file to disk.
- Obtain descriptions of existing constraint case data.
- Obtain descriptions of bar elements, contacts, and mass elements.
- Obtain descriptions of shell pairs defined in the model.

The FEM functions work for a model (part or assembly) and output an array of appropriate C data structures defined in the file *pro_mesh.h*.

If you set or import the configuration file option “fem_geom_associativity all”, you also get an output file named *modelname.fga* that contains elements and nodes, with regard to model geometry identifiers. You can then get more information on these geometry identifiers using the Pro/TOOLKIT geometry functions.

Exporting an FEA Mesh

Function Introduced:

- **ProFemmeshExport()**

The function **ProFemmeshExport()** generates the Finite Element Mesh based on the given parameters, and exports it to the specified file.

The function uses the data structure *ProFemmeshData*, which is defined as follows:

```
typedef struct pro_femmesh_data
{
    ProFemmeshType      mesh_type;
    ProFemshellmeshType shell_type;
    int                 num_quilts;
    ProFemquiltref      * pro_quilt_ref_arr;
    ProFemanalysisType  analysis;
    ProFemelemshapeType elem_shape;
    ProFemsolverType    solver;
    ProFemcsysref       csys_ref;
    int                 num_aux_csys;
```

```

    ProFemcsysref          * aux_csys_ref_arr;
}ProFemmeshData;

```

The *pro_femmesh_data* fields are as follows:

- *mesh_type*—The mesh type. The possible values are as follows:
 - PRO_FEM_SOLID_MESH—Mesh solid parts using tetrahedral solid mesh elements.
 - PRO_FEM_SHELL_MESH—Shell mesh using triangular or quadrangular mesh elements. This type is designed for meshing surfaces.
 - PRO_FEM_MIXED_MESH—Mesh models with a mixture of shell and tetrahedral mesh elements.
 - PRO_FEM_QUILT_MESH—A mesh for any simple or advanced shell idealizations created for quilt surfaces.
 - PRO_FEM_BOUNDARY_MESH—A shell mesh of triangular or quadrilateral elements on the model's exterior surfaces.
 - PRO_FEM_BAR_MESH—A bar mesh for one dimensional idealizations.
- *shell_type*—The type of shell element. This field is ignored for a solid mesh. The possible values are as follows:
 - PRO_FEM_TRIANGLE
 - PRO_FEM_QUADRANGLE
- *num_quilts*—The quilt identifier.
- *pro_quilt_ref_arr*—An array of references of quilt surfaces in the assembly.
- *analysis*—The analysis type. The possible values are as follows:
 - PRO_FEM_ANALYSIS_STRUCTURAL—Structural analysis, including stress, strain, thermal stress, and displacement.
 - PRO_FEM_ANALYSIS_MODAL—Modal analysis, including the constraint sets applied to the model.
 - PRO_FEM_ANALYSIS_THERMAL—Thermal analysis, including temperature, heat flux, and heat gradient.
- *elem_shape*—The type of element to be used for the solver.

- PRO_FEM_MIDPNT_LINEAR—Linear elements are used for the analysis. This includes corner nodes, straight edges, and planar faces.
- PRO_FEM_MIDPNT_PARABOLIC—Parabolic elements are used for the analysis.
- PRO_FEM_MIDPNT_PARABOLIC_FIXED—The excessively curved edges of solid and shell mesh parabolic elements are slightly straightened and then used for analysis.

The number of nodes that are as follows:

Tetrahedron	PRO_FEA_LINEAR: 4 nodes PRO_FEA_PARABOLIC: 10 nodes
Triangle	PRO_FEA_LINEAR: 3 nodes PRO_FEA_PARABOLIC: 6 nodes
Quadrangle	PRO_FEA_LINEAR: 4 nodes PRO_FEA_PARABOLIC: 8 nodes

- *solver*—The type of solver used for analysis. The possible values are:
 - PRO_FEM_FEAS_ANSYS—Specifies an ANSYS solver.
 - PRO_FEM_FEAS_NASTRAN—Specifies a NASTRAN solver.
 - PRO_FEM_FEAS_NEUTRAL—Specifies other solvers that support the FEM Neutral file format for analysis.
- *csys_ref*—An array of geometric references.
- *num_aux_csys*—Additional coordinate system to be included in the analysis.
- *aux_csys_ref_arr*—An array of geometric references for the auxiliary coordinate system.

Note: Prior to calling this function, the model (*pro_solid*) should be displayed in the graphics window.

The input arguments of this function are as follows:

- *pro_solid*— The handle of a Pro/E model (part or assembly).
- *p_mesh_data*—The pointer to the data structure containing the mesh generation parameters.

- *file_name*—The file name to export mesh to.

This function supersedes the function **pro_export_fea_mesh()**.

Getting Constraint Case Names

Function Introduced:

- **profem_get_con_case_names()**

The function **profem_get_con_case_names()** retrieves a list of all the constraint case names.

Getting Mesh Controls

Function Introduced:

- **profem_get_mesh_controls()**

The function **profem_get_mesh_controls()** retrieves all the mesh controls.

The function uses the data structure *Pro_fem_mesh_control*, which is defined as follows:

```
typedef struct pro_fem_mesh_control
{
    int          id;
    int          active;
    int          control_type;
    int          num_regions;
    Pro_fem_region *regions;
    union
    {
        int      num_of_nodes;
        double   value;
    } param;
} Pro_fem_mesh_control;
```

The *Pro_fem_mesh_control* fields are as follows:

- *id*—The identifier, as appears in the **FEM Info** command in Pro/ENGINEER.
- *active*—A flag that determines whether control is active. The possible values are as follows:
 - PRO_FEM_CON_ACTIVE
 - PRO_FEM_CON_SUPPRESSED

- *control_type*—The type of control. The possible values are as follows:
 - PRO_FEM_LOCAL_MAX—Constrain the maximum element size at a specific point, edge, or face of the model.
 - PRO_FEM_GLOBAL_MAX—Constrain the maximum element size over the whole model.
 - PRO_FEM_LOCAL_MIN—Constrain the minimum element size at a specific point, edge, or face of the model.
 - PRO_FEM_GLOBAL_MIN—Constrain the minimum element size over the whole model.
 - PRO_FEM_ON_EDGE—Limit the number of shell element nodes created on a model edge.
 - PRO_FEM_HARD_POINT—Force element nodes to exist at specific points in the model.
 - PRO_FEM_ENTER_POINT—Force the entrance point node for C-FLOW analysis to exist at the specified point in the model.
- *num_regions*—The number of terms in the *regions* array.
- *regions*—The region on which the mesh control is defined. See the section Pro_fem_region Data Structure on page 63 - 6 for the definition of this data structure.
- *num_of_nodes*—If the *control_type* field is PRO_FEM_ON_EDGE, this is the number of nodes. Otherwise, this field is not set.
- *value*—If the *control_type* field is *not* PRO_FEM_ON_EDGE, this is the minimum and maximum size of the elements.

Pro_fem_region Data Structure

The *Pro_fem_region* data structure is defined as follows:

```
typedef struct pro_fem_region
{
    int      active;
    int      region_type;
    int      num_members;
    int      *memb_id_table;
    int      region_id;
} Pro_fem_region;
```

The *Pro_fem_region* fields are as follows:

- *active*—A flag that determines whether the region is active. The possible values are as follows:
 - PRO_FEM_CON_ACTIVE
 - PRO_FEM_CON_SUPPRESSED
- *region_type*—The type of region. The possible values are as follows:
 - PRO_FEM_REG_POINT
 - PRO_FEM_REG_EDGE
 - PRO_FEM_REG_SURFACE
 - PRO_FEM_REG_VOLUME
 - PRO_FEM_REG_ALL_SURFACES
 - PRO_FEM_REG_PATTERN_PNT
 - PRO_FEM_REG_FEATURE_PNT
- *num_members*—The number of terms in the identifier path table.
- *memb_id_table*—The member identifier path table.
- *region_id*—The region on which the load or boundary condition, or bar element ends are defined.

Getting Constraints

Function Introduced:

- **profem_get_constraints()**

The function **profem_get_constraints()** retrieves all the loads and constraints defined for the specified constraint case. It uses the data structure *Pro_fem_constraint*, which is defined as follows:

```
typedef struct pro_fem_constraint
{
    int          id;
    int          active;
    int          constraint_type;
    int          num_regions;
    Pro_fem_region *regions;
    int          displacement_type;
    int          uniform_type;
    Pro_fem_con_value parameter[6];
    int          csys_type;
    int          csys_id;
```

```
} Pro_fem_constraint;
```

The *Pro_fem_constraint* fields are as follows:

- *id*—The identifier, as appears in the **FEM Info** command in Pro/ENGINEER.
- *active*—A flag that determines whether the load or constraint is active. The possible values are as follows:
 - PRO_FEM_CON_ACTIVE
 - PRO_FEM_CON_SUPPRESSED
- *constraint_type*—The type of constraint. The possible values are as follows:
 - PRO_FEM_PRESSURE—Apply pressure loads to surfaces.
 - PRO_FEM_FORCE—Apply force loads to datum points on the model.
 - PRO_FEM_MOMENT—Apply moments to datum points on the model. A moment describes the turning effect of a force about a pivot point.
 - PRO_FEM_DISPLACEMENT—Apply displacement constraints to points, edges, or faces.
 - PRO_FEM_EDGE_PRESSURE—Apply pressure loads along an edge.
 - PRO_FEM_STRUCT_TEMP—Apply structural temperatures to points, edges, or faces on the solid model.
 - PRO_FEM_ACCELERATION—Apply an acceleration load to a model in motion to constrain the model's speed and direction.
 - PRO_FEM_ANG_VELOCITY—Apply an angular velocity load to a model in motion to constrain the model's speed and direction.
 - PRO_FEM_TOTAL_FORCE—Apply total force loads to surfaces of the model.
 - PRO_FEM_TEMPERATURE—Apply temperature constraints to points, edges, or faces on the solid model.
 - PRO_FEM_CONVECTION—Apply convection loads to define the distribution of heat on a model face.
 - PRO_FEM_RADIATION—Apply radiation loads to model faces to define the emission of heat from a model.

- PRO_FEM_SURFACE_FLUX—Apply surface flux loads as heat flux constraints to model faces.
- PRO_FEM_EDGE_FLUX—Apply a heat flux along a model edge.
- PRO_FEM_POINT_HEAT—Apply a point heat load to place a heat source or sink at a specific point on the surface of a model.
- PRO_FEM_VOLUME_HEAT—Apply a volume heat load to specify a volumetric heat generation rate for the entire volume of a solid part.
- *num_regions*—The number of regions in the *regions* array.
- *regions*—The region on which the load or boundary condition is defined. See the section Pro_fem_region Data Structure on page 63 - 6 for the definition of this data type.
- *displacement_type*—The displacement type. The possible values are as follows:
 - PRO_FEM_DISP_STANDARD
 - PRO_FEM_DISP_ALONG_SURF
 - PRO_FEM_DISP_FIXED_RAD_ONLY
 - PRO_FEM_DISP_FIXED_ANGULAR
 - PRO_FEM_DISP_FIXED_AXIAL
- *uniform_type*—A flag that determines whether the constraint is uniform. The possible values are as follows:
 - PRO_FEM_UNIFORM
 - PRO_FEM_NONUNIFORM
- *parameter*—The load or boundary condition value. See the section Pro_fem_con_value Data Structure on page 63 - 10 for the definition of this data type.
- *csys_type*—The type of coordinate system. The possible values are as follows:
 - PRO_FEM_CSYS_CARTESIAN
 - PRO_FEM_CSYS_CYLINDRICAL
 - PRO_FEM_CSYS_SPHERICAL
- *csys_id*—The identifier of the coordinate system in which the vectors are defined.

Pro_fem_con_value Data Structure

The *Pro_fem_con_value* data structure is defined as follows:

```
typedef struct pro_fem_con_value
{
    int            direction_mask;    /* 1 if the displacement was set in
                                     this direction; otherwise, 0 */

    union
    {
        double     value;            /* uniform value */
        PRODEV_Line distrib_law;     /* nonuniform expression string */
    } param;
} Pro_fem_con_value;

typedef wchar_t PRODEV_Line [PRODEV_LINE_SIZE];
```

Getting Bar Elements and Contacts

Functions Introduced:

- **profem_get_bar_elements()**
- **profem_get_contacts()**

The function **profem_get_bar_elements()** retrieves all the bar elements. It uses the data structure *Pro_fem_bar_element*.

The function **profem_get_contacts()** retrieves all the contacts. It uses the data structure *Pro_fem_contact*.

The data structure for these functions is defined as follows:

```
typedef struct pro_fem_bar_contact
{
    int            id;
    int            active;
    int            type;
    Pro_fem_bar_contact_props *p_properties;
    Pro_fem_bar_contact_end *p_end[2];
} Pro_fem_bar_element, Pro_fem_contact;
```

The fields are as follows:

- *id*—The identifier. Use -1 for the default contact type.
- *active*—The flag that determines whether the contact is active. The possible values are as follows:
 - PRO_FEM_CON_ACTIVE

- PRO_FEM_CON_SUPPRESSED
- *type*—The type of contact. The possible values are as follows:
 - PRO_FEM_SPAR
 - PRO_FEM_BEAM
 - PRO_FEM_GAP
 - PRO_FEM_SPRING
 - PRO_FEM_CBEAM
 - PRO_FEM_ADV_SPRING
 - PRO_FEM_FREE
 - PRO_FEM_BOND
- *p_properties*—A union that specifies the properties of the bar contract. Use NULL if not present. See the *pro_mesh.h* file or the online browser for the definition of this union.
- *p_end*—A data structure that specifies the contact end. See the *pro_mesh.h* file or the online browser for the definition of this data structure.

Getting Mass Elements

Function Introduced:

- **profem_get_mass_elements()**

The function **profem_get_mass_elements()** retrieves all the mass elements. It uses the data structure *Pro_fem_mass_element*, which is defined as follows:

```
typedef struct pro_fem_mass_element
{
    int            id;
    int            active;
    Pro_fem_mass_props *p_properties;
    Pro_fem_region  *p_location;
} Pro_fem_mass_element;
```

The data structure fields are as follows:

- *id*—The element identifier.
- *active*—A flag that determines whether the mass element is active. The possible values are as follows:
 - PRO_FEM_CON_ACTIVE

- PRO_FEM_CON_SUPPRESSED
- *p_properties*—The mass properties. See the section Pro_fem_mass_props Data Structure on page 63 - 12 for the definition of this data structure.
- *p_location*—The location of the element. See the section Pro_fem_region Data Structure on page 63 - 6 for the definition of this data structure.

Pro_fem_mass_props Data Structure

The *Pro_fem_mass_props data* structure is defined as follows:

```
typedef struct pro_fem_mass_props
{
    double    value;
    int       csys_type;
    int       csys_id;
    double    moment_of_inertia[3];
} Pro_fem_mass_props;
```

The data structure fields are as follows:

- *value*—The mass value.
- *csys_type*—The type of coordinate system. The possible values are as follows:
 - PRO_FEM_CSYS_CARTESIAN
 - PRO_FEM_CSYS_CYLINDRICAL
 - PRO_FEM_CSYS_SPHERICAL
- *csys_id*—The identifier of the coordinate system.
- *moment_of_inertia*—The principal values of inertia.

Getting Shell Pairs

Function Introduced:

- **profem_get_shell_pairs()**

The function **profem_get_shell_pairs()** retrieves all the shell pair definitions for a part. It uses the data structure *Pro_fem_shell_pair*, which is defined as follows:

```
typedef struct pro_fem_shell_pair
{
    int          num_top_surfaces;      /* number of surfaces on the "red"
```



```

                                side */
int      *top_surface_ids;
int      num_bottom_surfaces; /* number of surfaces on the
                                "yellow" side */
int      *bottom_surface_ids;
int      thickness_type;      /* PRO_FEM_SHP_TH_CONST,
                                PRO_FEM_SHP_TH_VAR, or
                                PRO_FEM_SHP_TH_MULT_CONST */
int      placement_type;      /* PRO_FEM_SHP_PL_SIDE1,
                                PRO_FEM_SHP_PL_SIDE2,
                                PRO_FEM_SHP_PL_MIDDLE, or
                                PRO_FEM_SHP_PL_SELECT */
int      placement_surface_id; /* -1 for the not-selected surface
                                placement */
PRODEV_Name matl_name;        /* material assigned to the shell
                                pair */
} Pro_fem_shell_pair;

```

The data structure fields are as follows:

- *num_top_surfaces*—The number of surfaces on the red side.
- *top_surface_ids*—The identifiers for the top surface.
- *num_bottom_surfaces*—The number of surfaces on the yellow side.
- *bottom_surface_ids*—The identifiers for the bottom surface.
- *thickness_type*—The thickness type. The possible values are as follows:
 - PRO_FEM_SHP_TH_CONST
 - PRO_FEM_SHP_TH_VAR
 - PRO_FEM_SHP_TH_MULT_CONST
- *placement_type*—The placement type. The possible values are as follows:
 - PRO_FEM_SHP_PL_SIDE1
 - PRO_FEM_SHP_PL_SIDE2
 - PRO_FEM_SHP_PL_MIDDLE
 - PRO_FEM_SHP_PL_SELECT
- *placement_surface_id*—The identifier of the placement surface. This is –1 for the not-selected surface placement.
- *matl_name*—The name of the material assigned to the shell pair.

Example 1: Using the FEM Functions

The example code below illustrates how to use all the functions described in this chapter.

```
/*----- Pro/Develop Includes -----*/
#include "prodevelop.h"
#include "pro_mesh.h"

/*----- Pro/Toolkit Includes -----*/
#include <ProToolkit.h>
#include <PTApplsUnicodeUtils.h>
#include <ProMessage.h>
/*----- Application Includes -----*/
#include <TestError.h>

/*----- Function Prototypes -----*/
int user_get_fem_menu();
char *user_load_to_name ();
char *user_mctype_to_name ();
char *user_region_to_name ();
char *user_contact_to_name();
int user_get_fem_action ();

/*-----*\
    User-defined constants for the GET FEM options
\*-----*/
#define USER_FEM_GET_CONS_NAMES    0
#define USER_FEM_GET_CONSTRAINTS   1
#define USER_FEM_MASS_ELEMENTS     2
#define USER_FEM_MESH_CONTROLS     3
#define USER_FEM_GET_BAR_ELEMS     4
#define USER_FEM_GET_CONTACTS      5

/*-----*\
    To convert FEM load constants to strings for file input
    and output
\*-----*/
typedef struct load_to_name
{
    int    load;
    char   name[PRODEV_NAME_SIZE];
} Load_to_name;

static Load_to_name load_names[] =
{
    {PRO_FEM_PRESSURE, "PRO_FEM_PRESSURE"},
    {PRO_FEM_FORCE, "PRO_FEM_FORCE"},
    {PRO_FEM_MOMENT, "PRO_FEM_MOMENT"},
    {PRO_FEM_DISPLACEMENT, "PRO_FEM_DISPLACEMENT"},
    {PRO_FEM_EDGE_PRESSURE, "PRO_FEM_EDGE_PRESSURE"},
}
```

```

        {PRO_FEM_STRUCT_TEMP, "PRO_FEM_STRUCT_TEMP"},
        {PRO_FEM_ACCELERATION, "PRO_FEM_ACCELERATION"},
        {PRO_FEM_ANG_VELOCITY, "PRO_FEM_ANG_VELOCITY"},
        {PRO_FEM_TOTAL_FORCE, "PRO_FEM_TOTAL_FORCE"},
        {-1, "Not Implemented"},
    };

char *user_load_to_name (load)
int    load;
{
    char    name[PRODEV_NAME_SIZE];
    int     i;
    int     nloads = sizeof (load_names) / sizeof (Load_to_name);
    for (i = 0; i < nloads; i++)
        if (load_names[i].load == load)
            break;
    if (i == nloads) i--;
    return (load_names[i].name);
}

/*-----*\
    To convert FEM mesh control constants to strings for
    file input and output
\*-----*/
typedef struct mctype_to_name
{
    int     mctype;
    char    name[PRODEV_NAME_SIZE];
} Mctype_to_name;

static Mctype_to_name mctype_names[] =
{
    {PRO_FEM_LOCAL_MAX, "PRO_FEM_LOCAL_MAX"},
    {PRO_FEM_GLOBAL_MAX, "PRO_FEM_GLOBAL_MAX"},
    {PRO_FEM_LOCAL_MIN, "PRO_FEM_LOCAL_MIN"},
    {PRO_FEM_GLOBAL_MIN, "PRO_FEM_GLOBAL_MIN"},
    {PRO_FEM_ON_EDGE, "PRO_FEM_ON_EDGE"},
    {PRO_FEM_HARD_POINT, "PRO_FEM_HARD_POINT"},
    {PRO_FEM_ENTER_POINT, "PRO_FEM_ENTER_POINT"},
    {-1, "Not Implemented"},
};

char *user_mctype_to_name (mctype)
int    mctype;
{
    char    name[PRODEV_NAME_SIZE];
    int     i;
    int     nmctypes = sizeof (mctype_names) /
        sizeof (Mctype_to_name);
    for (i = 0; i < nmctypes; i++)
        if (mctype_names[i].mctype == mctype)

```

```

        break;
    if (i == nmctypes) i--;
    return (mctype_names[i].name);
}

/*-----*\
    To convert FEM region constants to strings for file
    input and output
\*-----*/

typedef struct region_to_name
{
    int    region;
    char   name[PRODEV_NAME_SIZE];
} Region_to_name;

static Region_to_name region_names[] =
{
    {PRO_FEM_REG_POINT, "PRO_FEM_REG_POINT"},
    {PRO_FEM_REG_EDGE, "PRO_FEM_REG_EDGE"},
    {PRO_FEM_REG_SURFACE, "PRO_FEM_REG_SURFACE"},
    {PRO_FEM_REG_VOLUME, "PRO_FEM_REG_VOLUME"},
    {PRO_FEM_REG_ALL_SURFACES, "PRO_FEM_REG_ALL_SURFACES"},
    {PRO_FEM_REG_PATTERN_PNT, "PRO_FEM_REG_PATTERN_PNT"},
    {PRO_FEM_REG_FEATURE_PNT, "PRO_FEM_REG_FEATURE_PNT"},
    {PRO_FEM_Y_NOT_DEFINED, "PRO_FEM_Y_NOT_DEFINED"},
    {PRO_FEM_AXIS, "PRO_FEM_AXIS"},
    {PRO_FEM_SURFACE, "PRO_FEM_SURFACE"},
    {-1, "Not Implemented"},
};

char *user_region_to_name (region)
int    region;
{
    char   name[PRODEV_NAME_SIZE];
    int    i;
    int    nregions = sizeof (region_names) /
        sizeof (Region_to_name);
    for (i = 0; i < nregions; i++)
        if (region_names[i].region == region )
            break;
    if (i == nregions ) i--;
    return (region_names[i].name);
}

/*-----*\
    To convert FEM contact constants to strings for file
    input and output
\*-----*/

typedef struct contact_to_name

```

```

{
    int    contact;
    char   name[PRODEV_NAME_SIZE];
} Contact_to_name;

static Contact_to_name contact_names[] =
{
    {PRO_FEM_SPAR, "PRO_FEM_SPAR"},
    {PRO_FEM_BEAM, "PRO_FEM_BEAM"},
    {PRO_FEM_GAP, "PRO_FEM_GAP"},
    {PRO_FEM_SPRING, "PRO_FEM_SPRING"},
    {PRO_FEM_CBEAM, "PRO_FEM_CBEAM"},
    {PRO_FEM_ADV_SPRING, "PRO_FEM_ADV_SPRING"},
    {PRO_FEM_FREE, "PRO_FEM_FREE"},
    {PRO_FEM_BOND, "PRO_FEM_BOND"},
    {-1, "Not Implemented"},
};

char *user_contact_to_name(contact)
int    contact;
{
    char   name[PRODEV_NAME_SIZE];
    int    i;
    int    ncontacts = sizeof (contact_names) /
        sizeof (Contact_to_name);
    for (i = 0; i < ncontacts; i++)
        if (contact_names[i].contact == contact )
            break;
    if (i == ncontacts) i--;
    return (contact_names[i].name);
}

/*=====*\
FUNCTION : user_get_fem_action()
PURPOSE  : To process each FEM action for file input
           and output
\*=====*/
int user_get_fem_action (p_model, action)
char *p_model;
int    action;
{
    char          fname[PRODEV_NAME_SIZE],
                name[PRODEV_NAME_SIZE];
    int           i, j, k, nnames, nitems;
    FILE          *fp;
    PRODEV_Name   *wnames;

    Pro_fem_constraint *constr_arr;
    Pro_fem_mesh_control *mesh_ctrls;
    Pro_fem_mass_element *mass_elems;
    Pro_fem_contact     *contacts;

```

```

Pro_fem_bar_element    *bar_elems;

ProError                err;
ProFileName             msg_fil;
ProPath                 path;

/*-----*\
    Open the output file
\*-----*/
ProStringToWstring( msg_fil, "msg_ugfund.txt" );
strcpy (fname, "femdata.txt");
if ((fp = PTApplsUnicodeFopen (fname, "a+")) == NULL)
{
    ProMessageDisplay(msg_fil, "USER %0s F",
        "Cannot open file for write");
    return (0);
}
ProTKFprintf (fp, "\n");

/*-----*\
    Apply the appropriate action corresponding to the menu selection
\*-----*/
switch (action)
{
    case USER_FEM_GET_CONS_NAMES:
        nnames = profem_get_con_case_names (p_model,
            &wnames);
        ProTKFprintf (fp, "Constraint Case Names[%d]\n",
            nnames);
        for (i = 0; i < nnames; i++)
            ProTKFprintf (fp, "\t%d.    %s\n", i,
                ProWstringToString (name, wnames[i]));
        break;

    case USER_FEM_GET_CONSTRAINTS:
        nnames = profem_get_con_case_names (p_model,
            &wnames);
        ProTKFprintf (fp, "Constraints[%d]\n", nnames);
        for (i = 0; i < nnames; i++)
        {
            ProTKFprintf (fp, "\t%d.    %s\n", i,
                ProWstringToString (name, wnames[i]));
            nitems = profem_get_constraints (p_model,
                wnames[i], &constr_arr);
            for (j = 0; j < nitems; j++)
                ProTKFprintf (fp, "\t    %s\n",
                    user_load_to_name (
                        constr_arr[j].constraint_type));
        }
        break;

    case USER_FEM_MASS_ELEMENTS:

```

```

nitems = profem_get_mass_elements (p_model,
                                   &mass_elems);
ProTKFprintf (fp, "Mass Elements[%d]\n", nitems);
for (i = 0; i < nitems; i++)
{
    ProTKFprintf (fp, "\t   MassValue: %8.3f\n",
                  mass_elems[i].p_properties->value);
    ProTKFprintf (fp, "\t   Region: %s\n",
                  user_region_to_name (
                      mass_elems[i].p_location->region_type));
}
break;

case USER_FEM_MESH_CONTROLS:
nitems = profem_get_mesh_controls (p_model,
                                   &mesh_ctrls);
ProTKFprintf (fp, "Mesh Controls[%d]\n", nitems);
for (i = 0; i < nitems; i++)
    ProTKFprintf (fp, "\t   %s\n",
                  user_mctype_to_name (
                      mesh_ctrls[i].control_type));
break;

case USER_FEM_GET_BAR_ELEMS:
nitems = profem_get_bar_elements (p_model,
                                   &bar_elems);
ProTKFprintf (fp, "Bar Elements[%d]\n", nitems);
for (i = 0; i < nitems; i++)
    ProTKFprintf (fp, "\t   %s\n",
                  user_contact_to_name (
                      bar_elems[i].type));
break;

case USER_FEM_GET_CONTACTS:
nitems = profem_get_contacts (p_model,
                              &contacts);
ProTKFprintf (fp, "Contacts[%d]\n", nitems);
for (i = 0; i < nitems; i++)
    ProTKFprintf (fp, "\t   %s\n",
                  user_contact_to_name (
                      contacts[i].type));
break;

default:
    break;
}
/*-----*\
    Close the output file and display it
/*-----*/
fclose (fp);
ProStringToWstring( path, "femdata.txt" );

```

```
err = ProInfoWindowDisplay( path, NULL, NULL );  
ERROR_CHECK( "user_get_fem_action", "ProInfoWindowDisplay", err );  
  
return (0);  
}
```


Mechanism Design: Mechanism Features

This chapter describes the programmatic creation of mechanism modeling entities such as springs and dampers as Pro/ENGINEER features.

We recommend you read the section, Overview of Feature Creation in the chapter, Element Trees: Principles of Feature Creation. It provides the necessary background for creating features using Pro/TOOLKIT.

Topic	Page
Mechanism Spring Feature	64 - 2
Mechanism Damper Feature	64 - 4
Mechanism Belt Feature	64 - 5
Mechanism 3D Contact Feature	64 - 8

Mechanism Spring Feature

A spring generates a translational or rotational spring force in a mechanism. It produces a linear spring force when stretched or compressed, and a torsion force when rotated. The magnitude of the spring force is directly proportional to the amount of displacement from the position of equilibrium.

Springs are created as Pro/ENGINEER features and their values are stored as valid Pro/ENGINEER parameters.

Feature Element Tree for the Mechanism Spring Feature

The element tree for the Mechanism Spring feature is documented in the header file, *ProDamperFeat.h*. The following figure demonstrates the structure of the feature element tree.

Figure 64-1: Feature Element Tree for Mechanism Spring

```
PRO_E_FEATURE_TREE
|
|--PRO_E_STD_FEATURE_NAME
|
|--PRO_E_SPRING_DAMPER_TYPE
|
|--PRO_E_SPRING_DAMPER_REF
|
|--PRO_E_SPRING_K
|
|--PRO_E_SPRING_U
|
|--PRO_E_SPRING_DIAMETER
|
|--PRO_E_SPRING_USE_DIAMETER
|
|--PRO_E_SPRING_ATTACH_POINTS
|
|--PRO_E_SPRING_ATTACH_REF
|
|--PRO_E_SPRING_FLIP_U_ANGLE
```

The following list details special information about the elements in the feature element tree:

- PRO_E_STD_FEAT_NAME—Specifies the name of the mechanism spring feature.
- PRO_E_SPRING_DAMPER_TYPE—Specifies the mechanism spring feature type. It can have the following values:
 - PRO_SPRING_DAMPER_FORCE—Specifies an extension or compression spring.
 - PRO_SPRING_DAMPER_TORQUE—Specifies a torsion spring.
- PRO_E_SPRING_DAMPER_REF—Specifies the spring placement references. For an extension spring or compression spring, you can select the translational axis or two points on two different bodies as the placement references. For a torsion spring, you can select the rotational axis as the placement reference.
- PRO_E_SPRING_K—Specifies the value for the stiffness coefficient of the spring.
- PRO_E_SPRING_U—Specifies the value for the unstretched length of the spring.
- PRO_E_SPRING_DIAMETER—Specifies the value for the spring icon diameter for an extension spring.
- PRO_E_SPRING_USE_DIAMETER—Specifies the Adjust Icon Diameter option (available in the Pro/ENGINEER user interface) that allows you to change the value of the spring icon diameter. This element is not available by default. It takes the following values:
 - PRO_SPRING_USE_DIAMETER_NO
 - PRO_SPRING_USE_DIAMETER_YES
- PRO_E_SPRING_ATTACH_POINTS—Specifies the attachment points for the two ends of a torsion spring. The attachment points can be of the following types:
 - PRO_SPRING_USE_MOTION_AXIS_ZERO—Specifies the JAS (Joint Axis Set) option, where the attachment references are automatically populated with the selected JAS references.
 - PRO_SPRING_CUSTOM_ATTACHMENT_POINTS—Specifies custom attachment references where you can specify two attachment references such as datum planes, datum points, and vertices. The vector between each attachment reference and the selected rotational axis is automatically calculated.

By default, the greatest angle between the two vectors is defined as the spring angle. On selecting both the attachment references, the unstretched value of the torsion spring is automatically updated according to the current angle between the two vectors.

- **PRO_E_SPRING_ATTACH_REF**—Specifies the actual attachment references for the torsion spring depending upon the attachment type set for the element **PRO_E_SPRING_ATTACH_POINTS**.
- **PRO_E_SPRING_FLIP_U_ANGLE**—Specifies the option to flip the direction of the current angle between the attachment references for a torsion spring.

Mechanism Damper Feature

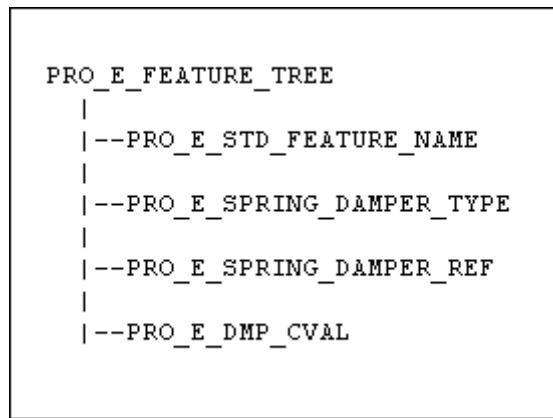
A damper generates a force that removes energy from a moving mechanism and dampens its motion. The damper force is always proportional to the magnitude of velocity of the entity on which you are applying the damper, and acts in the direction opposite to movement.

Dampers are created as Pro/ENGINEER features and their values are stored as valid Pro/ENGINEER parameters.

Feature Element Tree for the Mechanism Damper Feature

The element tree for the Mechanism Damper feature is documented in the header file, *ProDamperFeat.h*. The following figure demonstrates the structure of the feature element tree.

Figure 64-2: Feature Element Tree for Mechanism Damper



The following list details special information about the elements in the feature element tree:

- PRO_E_STD_FEAT_NAME—Specifies the name of the mechanism damper feature.
- PRO_E_SPRING_DAMPER_TYPE—Specifies the mechanism damper feature type. It can have the following values:
 - PRO_SPRING_DAMPER_FORCE—Specifies an extension damper or compression damper.
 - PRO_SPRING_DAMPER_TORQUE—Specifies a torsion damper.
- PRO_E_SPRING_DAMPER_REF—Specifies the damper placement references. For an extension damper or compression damper, you can select the translational or slot axis, or two points on two different bodies as the placement references. For a torsion damper, you can select the rotational axis as the placement reference.
- PRO_E_DMP_CVAL—Specifies the value for the damping coefficient.

Mechanism Belt Feature

A belt is treated as a connection connecting all the pulley bodies. The belt provides the same angular velocity to all pulleys connected by it.

Belts are created as Pro/ENGINEER features and their values are stored as valid Pro/ENGINEER parameters.

Feature Element Tree for the Mechanism Belt Feature

The element tree for the Mechanism Belt feature is documented in the header file, *ProBeltFeat.h*. The following figure demonstrates the structure of the feature element tree.

Figure 64-3: Feature Element Tree for Mechanism Belt

```
PRO_E_FEATURE_TREE
|
|--PRO_E_FEATURE_TYPE
|--PRO_E_STD_FEATURE_NAME
|
|--PRO_E_BELT_PULLEYS
|   |
|   |--PRO_E_BELT_PULLEY_RECORD
|       |--PRO_E_BELT_PULLEY_SEL
|       |--PRO_E_BELT_PULLEY_WRAP_SIDE
|       |--PRO_E_BELT_PULLEY_DIAMETER
|       |--PRO_E_BELT_PULLEY_DIAM_COINCIDENT
|       |--PRO_E_BELT_PULLEY_CONN_NUM
|       |--PRO_E_BELT_PULLEY_FLIP_CONN_BODIES
|       |--PRO_E_BELT_PULLEY_NUM_WRAPS
|
|--PRO_E_BELT_PLANE
|
|--PRO_E_BELT_DEFINE_CUSTOM_U_LENGTH
|--PRO_E_BELT_UNSTRETCHED_LENGTH
|--PRO_E_BELT_STIFFNESS_COEFF
```

The following list details special information about the elements in the feature element tree:

- PRO_E_FEATURE_TYPE—Specifies the feature type.
- PRO_E_STD_FEATURE_NAME—Specifies the name of the mechanism belt feature.
- PRO_E_BELT_PULLEYS—Specifies an array of pulley bodies of the type PRO_E_BELT_PULLEY_RECORD which consists of the following elements:

- **PRO_E_BELT_PULLEY_SEL**—Specifies the geometric reference selected as the pulley body. It can be a cylindrical surface, a circular curve or edge, or a PIN or Cylinder connection. You must define at least two pulley bodies for the belt.

Pro/ENGINEER automatically selects both the sides of the cylindrical surface or the circular curve or edge so that the pulley is complete. In case of a cylindrical or circular reference, the pulley's axis of rotation is automatically detected by detecting a PIN or Cylinder connection aligned to the theoretical axis of the geometric reference and perpendicular to the belt plane. If such a connection does not exist, the reference is considered invalid. In case of a connection reference, the rotation axis of the selected connection is used as the pulley axis.

- **PRO_E_BELT_PULLEY_WRAP_SIDE**—Specifies the pulley wrapping direction. It can be left (-1) or right (+1) and relative to the previous pulley. You can flip the belt to the other contact point detected between the belt and the pulley.
- **PRO_E_BELT_PULLEY_DIAMETER**—Specifies the pulley diameter. A cylindrical curve with the specified diameter is displayed around the selected pulley reference and on the belt plane. The pulley diameter is coincident with the pulley reference by default, except in case of connection references.
- **PRO_E_BELT_PULLEY_DIAM_COINCIDENT**—Indicates if the pulley diameter is coincident with the pulley reference. This element is set to **PRO_B_TRUE**, which means the pulley diameter is set to the value **Coincident** by default for non-connection references.
- **PRO_E_BELT_PULLEY_CONN_NUM**—Specifies the number of the PIN or Cylinder connection selected as the pulley reference from the available valid connections. One of the bodies that defines the connection is considered the pulley body, while the other body connected to it is considered the carrier body.
- **PRO_E_BELT_PULLEY_FLIP_CONN_BODIES**—Specifies the option to flip between the pulley and carrier bodies, in case of connection references. This option is not available in case of geometric references.

- PRO_E_BELT_PULLEY_NUM_WRAPS—Specifies the number of full wraps around the currently selected pulley. The wraps are considered as not overlapping and are not displayed in the 3D icon for the belt. The number of wraps affects the belt length. In case of open ended belts, the number of wraps influences the motion extent of the connected bodies.
- PRO_E_BELT_PLANE—Specifies the planar surface or datum plane that defines the belt plane. The plane should be perpendicular to the rotation axis of the first pulley. This element is optional. If the belt plane is not specified, it is detected based on the selected pulley references as follows:
 - Cylindrical surfaces—If a cylindrical surface is used as the pulley reference, the perpendicular plane positioned in the center of the cylindrical surface (for the first pulley in case of multiple pulleys) is used as the belt plane.
 - Circular edge or curve—If a circular edge or curve is used as the pulley reference, the plane for the corresponding edge or curve (for the first pulley in case of multiple pulleys) is used as the belt plane.
 - PIN/Cylinder connections—In case of connection references, the internal orange body zero point (for the first pulley in case of multiple pulleys) is used as the belt plane.
- PRO_E_BELT_DEFINE_CUSTOM_U_LENGTH—Identifies if the unstretched belt length can be specified by the user, or if it is system-defined. The length is system-defined by default.
- PRO_E_BELT_UNSTRETCHED_LENGTH—Specifies the value for the unstretched belt length. The system-defined length is calculated based on the pulley references and their specified diameters. When the user enters a desired belt length, Pro/ENGINEER tries to reconnect the assembly according to the specified belt length. If the reconnect operation fails, the belt length reverts to the previously entered value.
- PRO_E_BELT_STIFFNESS_COEFF—Specifies the belt stiffness coefficient value.

Mechanism 3D Contact Feature

3D contacts are connections between a pair of selected surfaces from two parts. These connections are similar to cams, joints, gears, and belts. Depending on the surfaces selected, 3D contacts can be of the following types:

- Sphere to Sphere (point contact)
- Sphere to Planar surface (point contact)
- Cylinder to Cylinder (line contact)
- Cylinder to Planar surface (line contact)

3D Contacts are created as Pro/ENGINEER features and their values are stored as valid Pro/ENGINEER parameters.

Feature Element Tree for the Mechanism 3D Contact Feature

The element tree for the Mechanism 3D Contact feature is documented in the header file, *ProContact3dFeat.h*. The following figure demonstrates the structure of the feature element tree.

Note: Improvements to the feature element tree for the 3D Contact feature are planned for the next release of Pro/ENGINEER. Owing to these enhancements, the existing element tree will not be available for use. Thus, you will need to rebuild your existing Pro/TOOLKIT applications according to the new element tree.

Figure 64-4: Feature Element Tree for Mechanism 3D Contact

```
PRO_E_FEATURE_TREE
|--PRO_E_FEATURE_TYPE
|--PRO_E_STD_FEATURE_NAME
|--PRO_E_C3D_REF1
|--PRO_E_C3D_MAT_OPTION1
|--PRO_E_C3D_MAT_NAME1
|--PRO_E_C3D_POISSON1
|--PRO_E_C3D_YOUNG1
|--PRO_E_C3D_DAMPING1
|--PRO_E_C3D_REF2
|--PRO_E_C3D_MAT_OPTION2
|--PRO_E_C3D_MAT_NAME2
|--PRO_E_C3D_POISSON2
|--PRO_E_C3D_YOUNG2
|--PRO_E_C3D_DAMPING2
|--PRO_E_C3D_VERT_RAD
|--PRO_E_C3D_FRICTION
|--PRO_E_C3D_STATIC_FRIC_COEF
|--PRO_E_C3D_KINEM_FRIC_COEF
```

The following list details special information about the elements in the feature element tree:

- PRO_E_FEATURE_TYPE—Specifies the feature type.
- PRO_E_STD_FEATURE_NAME—Specifies the name of the mechanism 3D contact feature.
- PRO_E_C3D_REF1—Specifies a single surface reference selected from the first part participating in the 3D contact. The reference can be a spherical, cylindrical, or planar surface. The first surface determines the second part that can be selected for the connection.
- PRO_E_C3D_MAT_OPTION1—Specifies the material type for the first contact part. The material type is given by the enumerated type *ProC3dMaterialType* that takes the following values:
 - PRO_C3D_MAT_DEFAULT—Specifies the material properties of the participating contact part. This is the default type. If this type is set, the elements PRO_E_C3D_POISSON1, PRO_E_C3D_YOUNG1, and PRO_E_C3D_DAMPING1 are not available for modification and are set by default.
 - PRO_C3D_MAT_SEL_MAT—Allows you to select a material type from the list of materials used in the assembly or from the material library directory. If this type is set, the elements PRO_E_C3D_POISSON1, PRO_E_C3D_YOUNG1, and PRO_E_C3D_DAMPING1 are not available for modification and are set as per the selected material.
 - PRO_C3D_MAT_USE_VALS—Allows you to specify the values for the material properties. The elements PRO_E_C3D_POISSON1, PRO_E_C3D_YOUNG1, and PRO_E_C3D_DAMPING1 are available and can be set as required.
- PRO_E_C3D_MAT_NAME1—Specifies the name of the material type selected for the first contact part. This element can be set only if the element PRO_E_C3D_MAT_OPTION1 is set to PRO_C3D_MAT_SEL_MAT.
- PRO_E_C3D_POISSON1—Specifies the value for Poisson's ratio for the first contact part.
- PRO_E_C3D_YOUNG1—Specifies the value for Young's modulus for the first contact part.

- PRO_E_C3D_DAMPING1—Specifies the value for the damping coefficient for the first contact part.
- PRO_E_C3D_REF2—Specifies multiple surface references selected from the second part participating in the 3D contact. All surfaces from the second contact part are paired to the single surface selected from the first contact part. If you select a vertex from the second part as one of references, a sphere is displayed around the vertex and the vertex is considered as a sphere in the 3D contact. If the second reference is a vertex, the PRO_E_C3D_VERT_RAD element becomes available for you to specify the value for the vertex radius.
- PRO_E_C3D_MAT_OPTION2—Specifies the material type for the second contact part. Refer to the description of the element PRO_E_C3D_MAT_OPTION1 for the material types that you can select.
- PRO_E_C3D_MAT_NAME2—Specifies the name of the material type selected for the second contact part. This element can be set only if the element PRO_E_C3D_MAT_OPTION2 is set to PRO_C3D_MAT_SEL_MAT.
- PRO_E_C3D_POISSON2—Specifies the value for Poisson's ratio for the second contact part.
- PRO_E_C3D_YOUNG2—Specifies the value for Young's modulus for the second contact part.
- PRO_E_C3D_DAMPING2—Specifies the value for the damping coefficient for the second contact part.
- PRO_E_C3D_VERT_RAD—Specifies the value for the vertex radius if a vertex is selected as one the references from the second contact part.
- PRO_E_C3D_FRICTION—Identifies if friction will be used in the contact calculation. This element is set to PRO_B_FALSE by default which means no friction.
- PRO_E_C3D_STATIC_FRIC_COEF—Specifies the value for the static friction coefficient.
- PRO_E_C3D_KINEM_FRIC_COEF—Specifies the value for the kinematic friction coefficient.

Event-driven Programming: Notifications

Notifications allow your Pro/TOOLKIT application to detect certain types of events in Pro/ENGINEER and provide that your own function is called before or after each such event. Pro/ENGINEER notifies your Pro/TOOLKIT application of these events.

Topic	Page
Using Notify	65 - 2
Notification Types	65 - 2

Using Notify

Functions Introduced:

- **ProNotificationSet()**
- **ProNotificationUnset()**

The function **ProNotificationSet()** sets up a notification by specifying the type of event and the pointer to the callback function. The event is specified as a value of the enumerated type *ProNotifyType*. The argument for the callback function has the type *ProFunction*; for consistency, each callback function returns a *ProError* status, even in cases where the status is ignored by Pro/ENGINEER. The callback functions have different arguments, so each callback type has its own typedef that defines its arguments and their types. When calling **ProNotificationSet()**, you must cast the callback function pointer to *ProFunction*.

Note: If you call **ProNotificationSet()** more than once with the same event type, the existing callback is overwritten with the one supplied in the later call.

To cancel a notification, call **ProNotificationUnset()**.

Notification Types

The notification events in Pro/ENGINEER fall into the following classes:

File Management Events

Notifications to all of the file management operations in Pro/ENGINEER, such as **Save**, **Retrieve**, **Copy**, **Rename**, and so on. The possible file management notifications fall into the following subclasses:

Pre-file Management Events

Your callback function is called before the file management event. It is called only for models that are the explicit objects of the file management operation. For example, it is not called when a part is saved as a result of saving a parent assembly.

If the Pro/ENGINEER user initiated the event, the callback is called before the prompt asking the user for the name of the Pro/ENGINEER models on which to act.

The callback function can optionally write output arguments that determine the Pro/ENGINEER models on which the event will operate. In this case, Pro/ENGINEER will not prompt the user.

The callback function can, by returning an error status, cancel the file management event altogether.

Pre-file Management Events

Event type	Callback typedef	Include file
PRO_FILE_OPEN_OK	ProFileOpenOKAction	ProNotify.h
PRO_MDL_SAVE_PRE	ProMdlSavePreAction	ProMdl.h
PRO_MDL_COPY_PRE	ProMdlCopyPreAction	ProMdl.h
PRO_MDL_RENAME_PRE	ProMdlRenamePreAction	ProMdl.h
PRO_MDL_ERASE_PRE	ProMdlErasePreAction	ProMdl.h
PRO_MDL_PURGE_PRE	ProMdlPurgePreAction	ProMdl.h
PRO_MDL_DELETE_PRE	ProMdlDeletePreAction	ProMdl.h
PRO_MDL_RETRIEVE_PRE	ProMdlRetrievePreAction	ProMdl.h
PRO_MDL_CREATE_PRE	ProMdlCreatePreAction	ProMdl.h
PRO_MDL_START	ProMdlStartAction	ProMdl.h
PRO_CHECKIN_UI_PRE	ProCheckinUIPreAction	ProNotify.h

Post-file Management Events

Your callback function is called after the file management operation, and is given input arguments that show which models were the subject of the operation. Like a pre-file management callback, it is called only for models that are the explicit objects of the file management operation.

Post-file Management Events

Event type	Callback typedef	Include file
PRO_MDL_SAVE_POST	ProMdlSavePostAction	ProMdl.h
PRO_MDL_COPY_POST	ProMdlCopyPostAction	ProMdl.h
PRO_MDL_RENAME_POST	ProMdlRenamePostAction	ProMdl.h
PRO_MDL_ERASE_POST	ProMdlErasePostAction	ProMdl.h
PRO_MDL_PURGE_POST	ProMdlPurgePostAction	ProMdl.h
PRO_MDL_DELETE_POST	ProMdlDeletePostAction	ProMdl.h
PRO_MDL_RETRIEVE_POST	ProMdlRetrievePostAction	ProMdl.h
PRO_MDL_CREATE_POST	ProMdlCreatePostAction	ProMdl.h
PRO_UDF_LIB_COMPLETE_POST	ProUdfLibraryCompletePostAction	ProUdf.h

Post All File Management Events

Your the callback function is called for all file management events on models, even if those models were not explicitly specified by the user. For example, if you save an assembly, the callback function is also called for any modified parts that are saved as a result of that action.

Post All File Management Events

Event type	Callback typedef	Include file
PRO_MDL_SAVE_POST_ALL	ProMdlSavePostAllAction	ProMdl.h
PRO_MDL_COPY_POST_ALL	ProMdlCopyPostAllAction	ProMdl.h
PRO_MDL_ERASE_POST_ALL	ProMdlErasePostAllAction	ProMdl.h
PRO_MDL_DELETE_POST_ALL	ProMdlDeletePostAllAction	ProMdl.h
PRO_MDL_RETRIEVE_POST_ALL	ProMdlRetrievePostAllAction	ProMdl.h

File Management Failure Events

Your callback function is called after a file management operation that failed. The function is called with arguments that show the type of file management operation that failed, the models it was operating on, and the type of error encountered (in the form of a *ProError* value).

File Management Failed Events

Event type	Callback typedef	Include file
PRO_MDL_DBMS_FAILURE	ProMdlDbmsFailureAction	ProMdl.h
PRO_MDL_CREATE_CANCEL	ProMdlCreateCancelAction	ProNotify.h

Model and Feature Modification Events

Notifications that signal a change to the content of the model. They often include both pre- and post-notifications of change events and include operations on features, solids, parameters and dimensions. See the section 'Notes on Regeneration Events' for more information.

Use the notification types PRO_FEATURE_REROUTE_PRE and PRO_FEATURE_REROUTE_POST to trap the command Feature Reroute in parts or assemblies. Use notification types PRO_FEATURE_REPLACE_PRE and PRO_FEATURE_REPLACE_POST to trap replacement of assembly components performed in assembly mode with the command **Component, Adv Utils, Replace**.

The notification types PRO_ASMCOMP_ACTIVATE_PRE and PRO_ASMCOMP_ACTIVATE_POST are called before and after an assembly component is activated within the context of an assembly in Pro/ENGINEER using the right mouse button **Activate**. The notification types provide the path to the currently active component, allowing applications to understand the context of an action in the assembly.

Notes on Regeneration Events

Notifications which trigger before or after feature regeneration should be used carefully, because your callback function is being called while the regeneration of a solid is in progress. This section describes some of the important information you should keep in mind when using notification for model events.

At the start of model regeneration, Pro/ENGINEER discards all data structures that describe geometry, although the geometry items themselves are retained (to preserve the integer identifiers). This means that although you can still traverse the features (using **ProSolidFeatVisit()**) and the geometry items in a feature (using **ProFeatureGeomitemVisit()**), geometry items belonging to features not yet regenerated will have no corresponding OHandles. Therefore, functions such as **ProSurfaceInit()** and **ProEdgeInit()** will not work. If you analyze the geometry of the features already regenerated, you will see it as unmodified by the features still to be regenerated.

It is dangerous to attempt modifications to the model or file management operations during a regeneration notification function.

Model Modification Events

Event type	Callback typedef	Include file
PRO_DIM_MODIFY_VALUE_PRE	ProDimModifyValuePreAction	ProDimension.h
PRO_FEATURE_CREATE_PRE	ProFeatureCreatePreAction	ProFeature.h
PRO_FEATURE_CREATE_POST	ProFeatureCreatePostAction	ProFeature.h
PRO_FEATURE_COPY_POST	ProFeatureCopyPostAction	ProFeature.h
PRO_FEATURE_DELETE_PRE	ProFeatureDeletePreAction	ProFeature.h
PRO_FEATURE_DELETE_POST	ProFeatureDeletePostAction	ProFeature.h
PRO_FEATURE_SUPPRESS_PRE	ProFeatureSuppressPreAction	ProFeature.h
PRO_FEATURE_SUPPRESS_POST	ProFeatureSuppressPostAction	ProFeature.h
PRO_FEATURE_REDEFINE_PRE	ProFeatureRedefinePreAction	ProFeature.h
PRO_FEATURE_REDEFINE_POST	ProFeatureRedefinePostAction	ProFeature.h
PRO_FEATURE_REGEN_PRE	ProFeatureRegenPreAction	ProFeature.h
PRO_FEATURE_REGEN_POST	ProFeatureRegenPostAction	ProFeature.h
PRO_FEATURE_REGEN_FAILURE	ProFeatureRegenFailureAction	ProFeature.h
PRO_FEATURE_NEEDS_REGEN_GET	ProFeatureNeedsRegenGet	ProFeature.h
PRO_FEATURE_REROUTE_PRE	ProFeatureReroutePreAction	ProFeature.h
PRO_FEATURE_REROUTE_POST	ProFeatureReroutePostAction	ProFeature.h
PRO_FEATURE_REPLACE_PRE	ProFeatureReplacePreAction	ProFeature.h
PRO_FEATURE_REPLACE_POST	ProFeatureReplacePostAction	ProFeature.h
PRO_GROUP_UNGROUP_PRE	ProGroupUngroupPreAction	ProGroup.h
PRO_GROUP_UNGROUP_POST	ProGroupUngroupPostAction	ProGroup.h
PRO_PARAM_CREATE_PRE	ProParameterCreatePreAction	ProParameter.h
PRO_PARAM_CREATE_POST	ProParameterCreatePostAction	ProParameter.h
PRO_PARAM_MODIFY_PRE	ProParameterModifyPreAction	ProParameter.h
PRO_PARAM_MODIFY_POST	ProParameterModifyPostAction	ProParameter.h

PRO_PARAM_DELETE_PRE	ProParameterDeletePreAction	ProParameter.h
PRO_PARAM_DELETE_POST	ProParameterDeletePostAction	ProParameter.h
PRO_SOLID_REGEN_PRE	ProSolidRegeneratePreAction	ProSolid.h
PRO_SOLID_REGEN_POST	ProSolidRegeneratePostAction	ProSolid.h
PRO_SOLID_UNIT_CONVERT_PRE	ProSolidUnitConvertPreAction	ProNotify.h
PRO_SOLID_UNIT_CONVERT_POST	ProSolidUnitConvertPostAction	ProNotify.h
PRO_SOLID_PRINC_SYS_UNITS_RENAMED_POST	ProSolidPrincSysUnitsRenamedPostAction	ProNotify.h
PRO_DWGTABLE_ROW_DELETE_PRE	ProDwgtableRowDeletePreAction	ProNotify.h
PRO_DWGTABLE_ROW_DELETE_POST	ProDwgtableRowDeletePostAction	ProNotify.h
PRO_DWGTABLE_DELETE_PRE	ProDwgtableDeletePreAction	ProNotify.h
PRO_DWGTABLE_DELETE_POST	ProDwgtableDeletePostAction	ProNotify.h

Context Change Events

Notifications called after events that change details in the current Pro/ENGINEER context. They allow your application to leverage the details of these changes as needed.

Session Context Events

Event type	Callback typedef	Include file
PRO_DIRECTORY_CHANGE_POST	ProDirectoryChangePostAction	ProNotify.h
PRO_WINDOW_CHANGE_POST	ProWindowChangePostAction	ProNotify.h
PRO_POPUPMENU_CREATE_POST	ProPopupMenuCreatePostAction	ProPopupMenu.h
PRO_POPUPMENU_DESTROY_PRE	ProPopupMenuDestroyPreAction	ProPopupMenu.h
PRO_WINDOW_VACATE_PRE	ProWindowVacatePreAction	ProWindows.h
PRO_WINDOW_OCCUPY_POST	ProWindowOccupyPostAction	ProWindows.h
PRO_GLOBAL_INTERF_CALC_POST	ProGlobalInterfCalcPostAction	ProNotify.h
PRO_ASMCOMP_ACTIVATE_PRE	ProAsmcompActivatePreAction	ProNotify.h
PRO_ASMCOMP_ACTIVATE_POST	ProAsmcompActivatePostAction	ProNotify.h

Graphics Events

Notifications before and after the repainting of the current Pro/ENGINEER window. This enables you to overlay your own graphics over the window and ensure that they get refreshed when the Pro/ENGINEER window is repainted, for any reason.

Note: These notifications will be called many times; for this reason, your callback routine should be as optimized as possible to avoid performance penalties. For other techniques that may be used to draw graphics, which will repaint with the Pro/ENGINEER window, refer to the section on Display Lists. Refer to the User Interface: Basic Graphics chapter for details.

Graphics Events

Event type	Callback typedef	Include file
PRO_MDL_DISPLAY_PRE	ProMdlDisplayPreAction	ProMdl.h
PRO_MDL_DISPLAY_POST	ProMdlDisplayPostAction	ProMdl.h

Pro/NC Output Events

Notification of the output from Pro/NC of an operation CL data file, or an NC sequence CL data file. This enables you to perform your own post-processing on these files. The callback functions are called with arguments that provide the name of the file created.

Pro/NC Output Events

Event type	Callback typedef	Include file
PRO_NCSEQ_CL_POST	ProNcseqClPostAction	ProNotify.h
PRO_OPER_CL_POST	ProMfgoperClPostAction	ProNotify.h

CL Command Events

Notifications that give you the ability to create auxiliary NC sequences with programmatically created CL commands.

CL Command Events

Event type	Callback typedef	Include file
PRO_NCL_COMMAND_EXPAND	ProClCommandExpandAction	ProClCmd.h
PRO_NCL_COMMAND_GET_LOC	ProClCommandGetLocAction	ProClCmd.h

Mold Layout Events

Notifications that are invoked before entering a corresponding Mold Layout dialog.

Mold Layout UI Events

Event type	Callback typedef	Include file
PRO_RMDT_CREATE_IMM_PRE	ProRmdtCreateImmPreAction	ProRmdt.h
PRO_RMDT_BOUND_BOX_PRE	ProRmdtBoundingBoxPreAction	ProRmdt.h
PRO_RMDT_CAV_LAYOUT_PRE	ProRmdtCavLayoutPreAction	ProRmdt.h
PRO_RMDT_CREATE_WP_PRE	ProRmdtCreateWpPreAction	ProRmdt.h
PRO_RMDT_MATERIAL_PRE	ProRmdtMaterialPreAction	ProRmdt.h
PRO_RMDT_MBASE_SELECT_PRE	ProRmdtMBaseSelectPreAction	ProRmdt.h

Weld Events

Notifications that give you the ability to customize the results generated by Pro/ENGINEER when gathering info for weld operations.

Weld Events

Event type	Callback typedef	Include file
PRO_DRAWING_WELD_SYMPATH_GET	ProDrawingWeldSympathGetAction	ProNotify.h
PRO_DRAWING_WELD_GROUPIDS_GET	ProDrawingWeldGroupidsGetAction	ProNotify.h
PRO_DRAWING_WELD_SYMTEXT_GET	ProDrawingWeldSymtextGetAction	ProNotify.h

66

Event-driven Programming: External Objects

This chapter describes the Pro/TOOLKIT functions that enable you to create and manipulate external objects.

Topic

Summary of External Objects	66 - 2
External Objects and Object Classes	66 - 3
External Object Data	66 - 6
External Object References	66 - 12
Callbacks for External Objects	66 - 14
Warning Mechanism for External Objects	66 - 16

Summary of External Objects

External objects are objects created by an application that is external to Pro/ENGINEER. Although these objects can be displayed and selected within a Pro/ENGINEER session, they can not be independently created by Pro/ENGINEER. Using Pro/TOOLKIT functions, you can define and manipulate external objects, which are then stored in a model database.

Note: External objects are limited to text and wireframe entities. In addition, external objects can be created for parts and assemblies only. That is, external objects can be stored in a part or assembly database only.

In a Pro/TOOLKIT application, an external object is defined by a *ProExtobj* object. This DHandle identifies an external object in the Pro/ENGINEER database, which contains the following information for the object:

- *Object class*—A class of external objects is a group that contains objects with similar characteristics. All external objects must belong to a class. Object class is contained in the *ProExtobjClass* object.
- *Object data*—The object data contains information about the display and selection of an external object. Object data is contained in the *ProWExtobjdata* object.
- *Object parameters*—External objects can own parameters. You can use the *ProParameter* API to get, set, and modify external object parameters.
- *Object references*—External objects can reference any Pro/ENGINEER object. This functionality is useful when changes to Pro/ENGINEER objects need to instigate changes in the external objects. The changes are communicated back to your Pro/TOOLKIT application via the callback functions.
- *Callback functions*—Pro/TOOLKIT enables you to specify callback functions for a class of external objects. These functions are called whenever the external object owner or reference is deleted, suppressed, or modified. In this manner, the appearance and behavior of your external objects can depend on the object owner or reference.

External Objects and Object Classes

This section describes the Pro/TOOLKIT functions that relate to the creation and manipulation of external objects and object classes. Note that this description does not address the display or selection of the external object. For more information on this topic, see External Object Data on page 66 - 6.

Creating External Object Classes

Functions Introduced:

- **ProExtobjClassCreate()**
- **ProExtobjClassDelete()**

Every external object must belong to a class. The concept of a “class” enables you to group together external objects that exhibit similar characteristics. In addition, classes permit multiple applications to create external objects without conflict.

The *ProExtobjClass* object contains the name and type of an external object class. PTC recommends that you supply a class name unique to your application. The type of the class is an integer that should vary among the different classes.

To register an external object class, pass a completed *ProExtobjClass* object to the function **ProExtobjClassCreate()**. To unregister a class, call the function **ProExtobjClassDelete()**.

Creating External Objects

Functions Introduced:

- **ProExtobjCreate()**
- **ProExtobjDelete()**
- **ProExtobjClassGet()**

After the object class is registered, you can create the external object by calling the function **ProExtobjCreate()**. This function requires as input the object class and owner of the external object. (Currently, the owner of the external object can be a part or an assembly only.) As output, this function gives a pointer to the handle of the newly created external object.

When the external object is created, it is assigned an integer identifier that is persistent from session to session. The external object is saved as part of the model database and will be available when the model is retrieved next.

To delete an external object, call the function **ProExtobjDelete()**. This function requires as input both the object to be deleted and the class to which it belongs. To determine the class of an external object, call the function **ProExtobjClassGet()**.

External Object Owners

Functions Introduced:

- **ProExtobjOwnerobjGet()**
- **ProExtobjOwnerobjSet()**

The owner of an external object is set during the call to **ProExtobjCreate()**. For example, the “owner” would be the part or assembly where the external object resides.

To determine the owner of an external object, call the function **ProExtobjOwnerobjGet()**. To change the owner, call the function **ProExtobjOwnerobjSet()**.

Recycling External Object Identifiers

Functions Introduced:

- **ProExtobjReusableSet()**
- **ProExtobjReusableGet()**
- **ProExtobjReusableClear()**

By default, the identifier of an external object is not “recycled.” When you delete an external object, its identifier is not freed for reuse by external objects that are subsequently created.

You can override this default behavior using the function **ProExtobjReusableSet()**. This function enables external object identifiers to be recycled. To determine whether external object identifiers are set to be recyclable, call the function **ProExtobjReusableGet()**. To reset to the default behavior (no recycling), call the function **ProExtobjReusableClear()**.

External Object Parameters

As with features and models, external objects can also have user-defined parameters. Although you can specify parameters for an external object, there is no method to retrieve these parameters interactively in Pro/ENGINEER. Therefore, external object parameters are a way to store information in the Pro/ENGINEER model that is not accessible to end-users.

You can convert a *ProExtobj* object to a *ProModelitem* object by casting. After this conversion, you can use the function **ProParameterCreate()** to create parameters for the *ProModelitem* object. See the 'Core: Parameters' chapter for more information.

External Types and Identifiers for External Objects

Functions Introduced:

- **ProExtobjExttypeSet()**
- **ProExtobjExttypeGet()**
- **ProExtobjExtidSet()**
- **ProExtobjExtidGet()**

ProExtobj is a DHandle that contains the type, identifier, and owner of an external object. This information identifies the external object in the Pro/ENGINEER database.

Some applications might require additional type and identifier information to be assigned to external objects. That is, the type and identifier may need to be independent of those assigned within Pro/ENGINEER.

The function **ProExtobjExttypeSet()** sets an external type for an external object. This function calls **ProParameterCreate()** internally and creates a parameter with the name EXTOBJ_EXTTYPE. The function **ProExtobjExttypeGet()** obtains the external type for the specified external object.

The function **ProExtobjExtidSet()** sets an external integer identifier for the specified external object. This function calls **ProParameterCreate()** internally and creates a parameter with name the EXTOBJ_EXTID. To get the external identifier for a given external object, call the function **ProExtobjExtidGet()**.

Visiting External Objects

Function Introduced:

- **ProExtobjVisit()**

Using the traversal functions for external objects, you can visit each external object in turn, and perform some action or filtration on it. The function **ProExtobjVisit()** specifies action and filter functions of type **ProExtobjVisitAction()** and **ProExtobjFilterAction()**, respectively.

External Object Data

Simply creating an external object does not allow the object to be displayed or selected in Pro/ENGINEER. For this, you must supply external object data that is used, stored, and retrieved by Pro/ENGINEER. The data is removed from the model database when the external object is deleted.

External object data is described by the opaque workspace handle *ProWExtobjdata*. The functions required to initialize and modify this object are specific to the type of data being created. That is, creating display data requires one set of functions, whereas creating selection data requires another.

Once you have created a *ProWExtobjdata* object, the manipulation of the external object data is independent of its contents: the functions required to add or remove data are the same for both display data and selection data.

The following sections describe the Pro/TOOLKIT functions that relate to external object data. The sections are as follows:

- Display Data for External Objects on page 66 - 6
- Selection Data for External Objects on page 66 - 10
- Manipulating External Object Data on page 66 - 11

Display Data for External Objects

Display data gives information to Pro/ENGINEER about how the external object is to appear in the model window. This data must include the color, scale, line type, and transformation of the external object. In addition, display data can include settings that override the user's ability to zoom and spin the external object.

Note that setting display data does not result in the external object being displayed. To see the object, you must repaint the model window using the function **ProWindowRepaint()**.

Allocating Display Data

Function Introduced:

- **ProDispdatAlloc()**

For display data, the workspace handle *ProWExtobjdata* is allocated using the function **ProDispdatAlloc()**. Because the other Pro/TOOLKIT display data functions require *ProWExtobjdata* as input, you must call **ProDispdatAlloc()** before calling the other functions in this section.

The input for **ProDispdatAlloc()** is the address of a *ProWExtobjdata* object that you declare in your application. You must set this *ProWExtobjdata* object to NULL before passing its address to **ProDispdatAlloc()**.

Creating the External Object Entity

Functions Introduced:

- **ProDispdatEntsSet()**
- **ProDispdatEntsGet()**

External objects are currently limited to text and wireframe entities. You can specify the entities to be displayed by creating an array of *ProCurvedata* objects that contain that necessary information. *ProCurvedata* is a union of specific entity structures, such as line, arrow, arc, circle, spline, and text. Note that when you specify the entities in the *ProCurvedata* array, the coordinate system used is the default model coordinate system.

After you have created the array of *ProCurvedata* objects, you can add entities to the display data by calling the function **ProDispdatEntsSet()**. Note that **ProDispdatEntsSet()** supports only PRO_ENT_LINE and PRO_ENT_ARC entities. However, you can draw polygons as multiple lines, and circles as arcs of extent 2π .

To obtain the entities that make up an external object, call the function **ProDispdatEntsGet()**.

Example 1: Creating an External Object on page 66 - 17 shows how to specify an external object that is composed of line segments.

Transformation of the External Object

Functions Introduced:

- **ProDispdatTrfSet()**
- **ProDispdatTrfGet()**
- **ProExtobjScreentrGet()**

To perform a coordinate transformation on an external object, you must set the transformation matrix within the associated display data. To do this, call the function **ProDispdatTrfSet()** and pass the transformation matrix as an input argument. To obtain the transformation matrix contained in a particular set of display data, call the function **ProDispdatTrfGet()**.

Example 1: Creating an External Object on page 66 - 17 implements a transformation from default coordinates to a coordinate system that is dependent on the orientation of a selected surface.

Note: Even if you do not want to transform your external object from the default coordinate system, you *must* specify a transformation matrix. In this case, pass the identity matrix to **ProDispdatTrfSet()**. If you omit this step, your external object will not be displayed.

To obtain the complete transformation of an object from external object coordinates (default coordinates) to screen coordinates, call the function **ProExtobjScreentrGet()**.

Note: In the assembly mode, **ProExtobjScreentrGet()** is applicable for external objects owned only by the top assembly model. Use the function **ProDispdatTrfGet()** to retrieve the transformation of external objects in sub-models in the assembly mode. In the part mode, **ProExtobjScreentrGet()** is applicable for all objects.

External Object Display Properties

Functions Introduced:

- **ProDispdatPropsSet()**
- **ProDispdatPropsGet()**

By default, when users spin or zoom in on a model, external objects are subjected to the same spin and zoom scale as the model. In addition, by default external objects are always displayed, even if the owner or reference objects are suppressed. Setting external object display properties within display data enables you to change these default behaviors.

The *ProExtobjDispprops* object is an enumerated type that contains the possible settings for display properties. To set any of these properties within display data, create a *ProExtobjDispprops* array that contains your settings and pass this array to the function **ProDispdatPropsSet()**. To determine the display settings for specified display data, call the function **ProDispdatPropsGet()**.

The settings contained in *ProExtobjDispprops* are as follows:

- **PRO_EXTOBJ_ZOOM_INVARIANT**—Sets the external object to be invariant with the zoom scale or magnification of the model. The object appears the same size at all times.
- **PRO_EXTOBJ_SPIN_INVARIANT**—Set the external object to be invariant with the spin or orientation of the model. The object has the same orientation at all times.
- **PRO_EXTOBJ_BLANKED**—Blank the display of the external object. This setting is useful if you want to suppress the external object when the reference or owner objects are suppressed.

External Object Color

Functions Introduced:

- **ProDispdatColorGet()**
- **ProDispdatColorSet()**

The enumerated type *ProColortype* specifies the colors available for external objects. To set the object color within display data, call the function **ProDispdatColorSet()**. To determine the color in the specified display data, use **ProDispdatColorGet()**.

Line Styles for External Objects

Functions Introduced:

- **ProDispdatLinestyleSet()**
- **ProDispdatLinestyleGet()**

The enumerated type *ProLinestyle* specifies the line styles available for external objects. To set the object line style within the display data, call the function **ProDispdatLinestyleSet()**. To determine the line style in the specified display data, use **ProDispdatLinestyleGet()**.

External Object Scale

Functions Introduced:

- **ProDispdatScaleSet()**
- **ProDispdatScaleGet()**

To vary the size of your external object without altering the entities themselves, you must specify an object scale factor as part of the display data. To set the scale factor, call the function **ProDispdatScaleSet()**. To determine the scale factor in the specified display data, use **ProDispdatScaleGet()**.

Example 1: Creating an External Object on page 66 - 17 shows how to set the scale of an object to be dependent on the size of the owner object.

Selection Data for External Objects

Functions Introduced:

- **ProSeldatAlloc()**
- **ProSeldatSelboxesSet()**
- **ProSeldatSelboxesGet()**

You can select external objects using the Pro/TOOLKIT selection function **ProSelect()**, with the selection option *ext_obj*. For this selection to be possible, however, you must designate a set of “hot spots,” or selection boxes for the object. These selection boxes indicate locations in which mouse selections will cause the external object to be selected. Selection boxes are specified as part of the external object selection data.

The function **ProSeldatAlloc()** allocates selection data in preparation for the specification of the selection boxes.

A selection box is defined by the pair of points contained in a *ProSelbox* object. The coordinates of the points are specified in the external object's coordinate system (the default coordinates). The line between the points forms the diagonal of the selection box; the edges of the box lie parallel to the coordinate axes of the external object. To set the selection boxes within the selection data, call the function **ProSeldatSelboxesSet()** and pass as input a pointer to a list of *ProSelbox* objects. This enables your external object to have more than one associated selection box.

Note: PTC recommends that the size and arrangement of the selection boxes be dependent on the size and shape of the external object. If the external object is compact and uniformly distributed in all coordinate directions, one selection box will probably suffice.

However, if the external object is distributed nonuniformly, or is interfering with other objects, you must designate more specific locations at which selection should occur.

To obtain the list of selection boxes in a given selection data, call the function **ProSeldatSelboxesGet()**.

The **ProSelect()** function returns an array of *ProSelection* objects. To obtain a *ProExtobj* object from a *ProSelection* object, call the function **ProSelectionModelitemGet()** and cast the output *ProModelitem* directly into *ProExtobj*. (*ProExtobj* and *ProModelitem* are DHandles with identical declarations.)

Manipulating External Object Data

Functions Introduced:

- **ProExtobjdataAdd()**
- **ProExtobjdataSet()**
- **ProExtobjdataGet()**
- **ProExtobjdataRemove()**
- **ProExtobjdataFree()**

The previous two sections describe how to create and modify external object data. In the case of both display and selection data, the data creation process results in the opaque workspace handle *ProWExtobjdata*. The functions in this section enable you to manipulate how the external object data relates to the object itself.

To add new data to an external object, pass the data handle *ProWExtobjdata* to the function **ProExtobjdataAdd()**. To set the contents of existing object data, call the function **ProExtobjdataSet()**.

The function **ProExtobjdataGet()** obtains the handle for the display or selection data associated with an external object. To specify which type of data you want to retrieve, pass to this function one of the values in the enumerated type *ProExtobjdataType*. The declaration is as follows:

```
typedef enum
{
    PRO_EXTOBJDAT_DISPLAY,
    PRO_EXTOBJDAT_SELBOX
} ProExtobjdataType;
```

To remove data from an external object, use the function **ProExtobjdataRemove()**. To free the memory occupied by external object data, call the function **ProExtobjdataFree()**.

External Object References

You can use external object references to make external objects dependent on model geometry. For example, consider an external object that is modeled as the outward-pointing normal of a surface. Defining the surface as a reference enables the external object to behave appropriately when the surface is modified, deleted, or suppressed.

In general, an external object can reference any of the geometry that belongs to its owner. In addition, if the owner belongs to an assembly, the external object can also reference the geometry of other assembly components, provided that you supply a valid component path.

Note: Setting up the references for an external object does not fully define the dependency between the object and the reference. You must also specify the callback function to be called when some action is taken on the reference.

Creating External Object References

Functions Introduced:

- **ProExtobjRefAlloc()**
- **ProExtobjRefFree()**
- **ProExtobjRefselectionSet()**
- **ProExtobjRefselectionGet()**
- **ProExtobjReftypeSet()**
- **ProExtobjReftypeGet()**
- **ProExtobjRefAdd()**
- **ProExtobjRefRemove()**

The *ProWExtobjRef* object is an opaque workspace handle that defines an external object reference. To allocate the memory for a new external object reference, call the function **ProExtobjRefAlloc()**. To free the memory occupied by an object reference, call the function **ProExtobjRefFree()**.

If you have the *ProSelection* object that corresponds to your intended reference geometry, you can set this *ProSelection* to be the reference by calling the function **ProExtobjRefselectionSet()**. To obtain the *ProSelection* object for a specified reference, use **ProExtobjRefselectionGet()**.

You might need to use “reference types” to differentiate among the references of an external object. To set a reference type, call the function **ProExtobjReftypeSet()**. To obtain the reference type of the specified reference, call the function **ProExtobjReftypeGet()**.

Once you have set the *ProSelection* and the reference type for an external object reference, you must add the reference to the external object using the function **ProExtobjRefAdd()**. To remove a reference from an external object, use **ProExtobjRefRemove()**.

Visiting External Object References

Function Introduced:

- **ProExtobjRefVisit()**

Using the traversal functions for external object references, you can visit each external object reference in turn, and perform some action or filtration on it. The function **ProExtobjRefVisit()** specifies action and filter functions of type *ProExtobjRefVisitAction()* and *ProExtobjRefFilterAction()*, respectively.

Callbacks for External Objects

Functions Introduced:

- **ProExtobjCBAct()**
- **ProExtobjCBEnable()**
- **ProExtobjCallbacksSet()**

External objects are associated with their owners and the references that you specify. Currently, the callbacks mechanism for external objects enables you to receive notification when the reference is deleted, modified, or suppressed. Your callback function can respond in a manner appropriate for the action taken on the reference.

The *ProExtobjCallbacks* object is a structure that specifies the callback functions for each action on the external object's owner or reference. Each callback function is specified by a function pointer of type *ProExtobjCBAct*. When you create an external object class, you should also fill in a *ProExtobjCallbacks* object for that class. To set the callbacks for the class, call the **ProExtobjCallbacksSet()** function.

Notes:

- Currently, the only supported callbacks for external objects are for deletion, modification, and suppression.
- You cannot use a callback for an external object that references a Pro/ENGINEER feature (and not some geometry of it).

The *ProExtobjCallbacks* data structure is defined as follows:

```
typedef struct
{
    int                enabled_cbs;
    ProExtobjCBAct     display_CB;          /* not yet implemented */
    ProExtobjCBAct     select_CB;           /* not yet implemented */
    ProExtobjCBAct     owner_modify_CB;     /* not yet implemented */
    ProExtobjCBAct     owner_suppress_CB;   /* not yet implemented */
}
```

```

ProExtobjCBAct  owner_delete_CB;    /* not yet implemented */
ProExtobjCBAct  ref_modify_CB;
ProExtobjCBAct  ref_suppress_CB;
ProExtobjCBAct  ref_delete_CB;
} ProExtobjCallbacks;

```

The first field, *enabled_cbs*, is a flag that enables and disables the callback functions. Set each of the other fields in the structure to the name of the callback function appropriate for each action. To enable or disable the callback functions for a particular action and object class, call the function **ProExtobjCBEnable()**.

As shown in the previous structure, the external objects callbacks are implemented only for cases where the reference is modified, suppressed, or deleted. For this reason, you must exercise caution when enabling callbacks using **ProExtobjCBEnable()**. One of the inputs of the function is an action bitmask that specifies which callback actions are to be enabled. The action bitmask is composed of members of the enumerated type *ProExtobjAction*. The values of the enumerated type are as follows:

```

typedef enum
{
    PRO_EO_ALT_DISPLAY      = (1 << 6),
                          /* alternate display --
                          not implemented */
    PRO_EO_ALT_SELECT       = (1 << 7),
                          /* alternate selection --
                          not implemented */
    PRO_EO_ACT_OWN_MODIF    = (1 << 9),
                          /* not implemented */
    PRO_EO_ACT_OWN_SUPPR    = (1 << 10),
                          /* not implemented */
    PRO_EO_ACT_OWN_DELETE   = (1 << 11),
                          /* not implemented */
    PRO_EO_ACT_REF_MODIF    = (1 << 13),
    PRO_EO_ACT_REF_SUPPR    = (1 << 14),
    PRO_EO_ACT_REF_DELETE   = (1 << 15)
} ProExtobjAction;

```

The action bitmask must not contain any callback actions that are not supported. Given the comments in the *ProExtobjCallbacks* structure, the only allowed callback actions are **PRO_EO_ACT_REF_MODIF**, **PRO_EO_ACT_REF_SUPPR**, and **PRO_EO_ACT_REF_DELETE**.

The following table describes the actions given in the *ProExtobjCallbacks* data structure.

Callback Type	When it is Triggered
display_CB	The external object is displayed. Currently, this is not implemented.
select_CB	The external object is selected. Currently, this is not implemented.
owner_modify_CB	The owner of the external object is modified. Currently, this is not implemented.
owner_suppress_CB	The owner of the external object is suppressed. Currently, this is not implemented.
owner_delete_CB	The owner of the external object is deleted. Currently, this is not implemented.
ref_modify_CB	The reference of the external object is modified.
ref_suppress_CB	The reference of the external object is suppressed.
ref_delete_CB	The reference of the external object is deleted.

Warning Mechanism for External Objects

Functions Introduced:

- **ProExtobjClassWarningEnable()**
- **ProExtobjClassWarningDisable()**
- **ProExtobjWarningEnable()**
- **ProExtobjWarningDisable()**

When users perform some action on the references of an external object, you might want to display a warning message to ask users to confirm the action. Pro/TOOLKIT includes functions that implement such warnings, either for all external objects in a class or for individual external objects. For example, if a user attempts to

delete a feature whose geometry is referenced by a class of external objects, the Pro/TOOLKIT warning mechanism, if enabled, would open a warning window that states the potential problem. The users would pick Yes to continue with the deletion, or No to abort the deletion.

To enable the warnings for a class of external objects, call the function **ProExtobjClassWarningEnable()**. Note that all external objects that belong to the class will inherit the enabled warning if they are created subsequent to the call to **ProExtobjClassWarningEnable()**.

The input arguments to **ProExtobjClassWarningEnable()** are the class object *ProExtobjClass* and an action bitmask composed of members of the enumerated type *ProExtobjAction* (described in the section Callbacks for External Objects). The action bitmask specifies for which actions the warning is to be displayed.

Currently, the only supported actions are PRO_EO_ACT_REF_SUPPR (suppression of the reference) and PRO_EO_ACT_REF_DELETE (deletion of the reference).

To have the warnings displayed for both reference suppression and deletion, the call to **ProExtobjClassWarningEnable()** would appear as follows:

```
ProExtobjClassWarningEnable (&User_arrow_class,
                             PRO_EO_ACT_REF_SUPPR|PRO_EO_ACT_REF_DELETE);
```

In this call, *User_arrow_class* is declared as a *ProExtobjClass* (an external object class).

To disable the warnings for a class of external objects, call **ProExtobjClassWarningDisable()**. Note that all external objects that belong to the class will inherit the disabled warning if they are created subsequent to the call to the function **ProExtobjClassWarningDisable()**.

To enable warnings for a single external object (not the entire class), call the function **ProExtobjWarningEnable()**. This function is similar to **ProExtobjClassWarningEnable()** except the first argument for **ProExtobjWarningEnable()** is a pointer to a *ProExtobj* object. To disable the warnings for a single external object, call the function **ProExtobjWarningDisable()**.

Example 1: Creating an External Object

The following example code shows how to create an external object at a location specified by the user. The external object is a green arrow that is aligned with the normal to a selected surface.

Note: For the sake of simplicity, the example does not implement selection data.

```

/*-----*\
Pro/TOOLKIT includes
\*-----*/
#include <ProToolkit.h>
#include <ProSelection.h>
#include <ProObjects.h>
#include <ProSolid.h>
#include <ProGeomitem.h>
#include <ptc_geom.h>
#include <ProModelitem.h>
#include <ProMdl.h>
#include <ProExtobj.h>
#include <ProExtobjDisp.h>
#include <ProExtobjCB.h>
#include <ProExtobjRef.h>
/*-----*\
Application includes
\*-----*/
#include "ProUtil.h"
#include "TestError.h"
#include "UtilMath.h"
#include "math.h"
/*-----*\
Application data
\*-----*/
static ProExtobjClass   User_arrow_class;
ProName msg_file;      /* message file */
/*=====*\
FUNCTION: UserExtobjCreate()
PURPOSE:  Example of external object creation
\*=====*/
int UserExtobjCreate()
{
    static int calls = 0;          /* number of calls to this
                                   function */
    static int extobjid = 1;       /* identifiers for the external
                                   objects */

    int UserMakeSelections (ProSurface *, ProModelitem *, Pro3dPnt,
        ProSelection **);          /* function to get the selections */
    ProError UserExtobjCBSet (ProExtobjClass *);
                                   /* function to set the callbacks */
    ProError UserExtobjRefAdd (ProExtobj *, ProSelection *);
                                   /* function to set the references */

    ProMdl      model;             /* current model */
    ProSurface   surf;             /* target surface to get the arrow */
    ProModelitem surf_mdllitem;    /* target surface ProModelitem */

```

```

ProSelection  *p_surf_sel;          /* target surface ProSelection */
ProModelitem  surf_owner_mditem;   /* ProModelitem for the surface
                                     owner */

int           status;               /* return value */
ProUvParam    surf_uv;              /* surface point in uv */
ProVector     surf_xyz;             /* surface point in xyz */
ProVector     deriv1[2], deriv2[3]; /* derivatives at the point on
                                     the surface */

ProVector     surf_norm;            /* surface normal */
ProExtobj     arrow_obj;            /* external object to be
                                     created */

ProWExtobjdata arrow_disp_data = NULL;
                                     /* display data for the
                                     external object */

double        scale;                /* scale factor for the external
                                     object */

Pro3dPnt      outline_points[2];    /* points delimiting the
                                     solid */

ProCurvedata curvdata[10];         /* array of curves to be
                                     drawn */

int           i, j;                 /* counters */
ProMatrix     transform;            /* arrow transformation */
/*-----*\
Unit arrow to be transformed and scaled
\*-----*/

#define ARROW_COLOR PRO_COLOR_SHEETMETAL /* green */
#define ARROW_LINestyle PRO_LINestyle_SOLID /* solid lines */
#define ARROW_SCALE 0.1 /* scale (fraction of model size) */
#define ARROW_NUM_SEG 9 /* number of segments in the arrow */
#define HW 0.2 /* head width */
#define HL 0.2 /* head length */
ProLinedata unit_arrow[ARROW_NUM_SEG] = {
    {PRO_ENT_LINE, {HL,0.0,0.0}, {1.0,0.0,0.0}}, /* "shaft" */
    {PRO_ENT_LINE, {0.0,0.0,0.0}, {HL,HW,0.0}},
    {PRO_ENT_LINE, {0.0,0.0,0.0}, {HL,0.0,HW}},
    {PRO_ENT_LINE, {0.0,0.0,0.0}, {HL,-HW,0.0}},
    {PRO_ENT_LINE, {0.0,0.0,0.0}, {HL,0.0,-HW}},
    {PRO_ENT_LINE, {HL,HW,0.0}, {HL,0.0,HW}},
    {PRO_ENT_LINE, {HL,0.0,HW}, {HL,-HW,0.0}},
    {PRO_ENT_LINE, {HL,-HW,0.0}, {HL,0.0,-HW}},
    {PRO_ENT_LINE, {HL,0.0,-HW}, {HL,HW,0.0}}
};
ProStringToWstring (msg_file, "msg_ugfund.txt");
/*-----*\
Get the current model.
\*-----*/
status = ProMdlCurrentGet (&model);
if (status != PRO_TK_NO_ERROR)
{
    ProMessageDisplay (msg_file, "USER %0s",
        "Error getting current model.");
}

```

```

        return (status);
    }
/*-----*\
    Create the external object class.
\*-----*/
    if (calls == 0)
    {
        ProStringToWstring (User_arrow_class.name, "Arrows");
        User_arrow_class.type = 0;

        status = ProExtobjClassCreate (&User_arrow_class);
/*-----*\
        Set up warnings for the entire class.  Users will be warned if
        the objects' references are either suppressed or deleted.
\*-----*/
        status = ProExtobjClassWarningEnable (&User_arrow_class,
            PRO_EO_ACT_REF_SUPPR|PRO_EO_ACT_REF_DELETE);
    }
    ++calls;
/*-----*\
    Use the model size to get the scale.
\*-----*/
    status = ProSolidOutlineGet ((ProSolid) model, outline_points);
    scale = ARROW_SCALE * ProUtilPointsDist (outline_points[1],
        outline_points[0]);
/*-----*\
    Get ProSurface and ProModelitem from the user selection.  Also
    get the xyz point selected.
\*-----*/
    status = UserMakeSelections (&surf, &surf_mdllitem,
        surf_xyz, &p_surf_sel);

    if (status != PRO_TK_NO_ERROR)
        return (status);
/*-----*\
    Get the ProModelitem for the surface owner.
\*-----*/
    status = ProMdlToModelitem (surf_mdllitem.owner, &surf_owner_mdllitem);
/*-----*\
    Determine the uv parameters from the xyz point.
\*-----*/
    status = ProSurfaceParamEval ((ProSolid) surf_mdllitem.owner, surf,
        surf_xyz, surf_uv);
/*-----*\
    Evaluate the surface data (normal, gradient,...) at the uv point.
\*-----*/
    status = ProSurfaceXyzdataEval (surf, surf_uv, surf_xyz,
        deriv1, deriv2, surf_norm);
/*-----*\
    Get the transformation for the arrow.
\*-----*/

```



```

    ProUtilVectorNormalize (surf_norm, surf_norm);
    ProUtilVectorNormalize (deriv1[0], deriv1[0]);
    ProUtilVectorNormalize (deriv1[1], deriv1[1]);
    status = ProUtilVectorsToTransf (surf_norm, deriv1[0], deriv1[1],
        surf_xyz, transform);
/*-----*\
    Create the arrow object with the same owner as the selected
    surface.
\*-----*/
    status = ProExtobjCreate (&User_arrow_class, &surf_owner_mditem,
        &arrow_obj);
/*-----*\
    Set the object identifiers to be reusable.
\*-----*/
    status = ProExtobjReusableSet (&arrow_obj);
/*-----*\
    Initialize the display data structure.
\*-----*/
    status = ProDispdatAlloc (&arrow_disp_data);
/*-----*\
    Set the display scale, color, line type, and display properties.
\*-----*/
    status = ProDispdatScaleSet (arrow_disp_data, scale);
    status = ProDispdatColorSet (arrow_disp_data, ARROW_COLOR);
    status = ProDispdatLinestyleSet (arrow_disp_data, ARROW_LINESTYLE);
/*-----*\
    Set the line segments (the "entities") in the display data.
\*-----*/
    for (i = 0; i < ARROW_NUM_SEG; ++i)
    {
        curvedata[i].line.type = unit_arrow[i].type;
        for (j = 0; j < 3; ++j)
        {
            curvedata[i].line.end1[j] = unit_arrow[i].end1[j];
            curvedata[i].line.end2[j] = unit_arrow[i].end2[j];
        }
    }
    status = ProDispdatEntsSet (arrow_disp_data, &curvedata[0],
        ARROW_NUM_SEG);
/*-----*\
    Set the arrow transformation.
\*-----*/
    status = ProDispdatTrfSet (arrow_disp_data, transform);
/*-----*\
    Add the display data to the object.
\*-----*/
    status = ProExtobjdataAdd (&arrow_obj, &User_arrow_class,
        arrow_disp_data);
/*-----*\
    Repaint the window so the object is displayed.
\*-----*/

```

```

    ProWindowRepaint (PRO_VALUE_UNUSED);
/*-----*\
    Set the references of the external object.
\*-----*/
    status = UserExtobjRefAdd (&arrow_obj, p_surf_sel);
/*-----*\
    Set the callback functions for the external objects.
\*-----*/
    status = UserExtobjCBSet (&User_arrow_class);

    return (PRO_TK_NO_ERROR);
}
/*=====*\
FUNCTION: UserExtobjCBSet()
PURPOSE:  Set the callback functions for an external object class.
\*=====*/
ProError UserExtobjCBSet (ProExtobjClass *obj_class)
{
    ProExtobjCallbacks *classCBs;
    ProError           UserArrowsCB();
    int                status;

    classCBs =
        (ProExtobjCallbacks *) calloc (1, sizeof(ProExtobjCallbacks));
/*-----*\
    See the file ProExtobjCB.h to see which callbacks are implemented.
\*-----*/
    classCBs->enabled_cbs = PRO_B_TRUE;
    classCBs->display_CB = NULL;
    classCBs->select_CB = NULL;
    classCBs->owner_modify_CB = NULL;
    classCBs->owner_suppress_CB = NULL;
    classCBs->owner_delete_CB = NULL;
    classCBs->ref_modify_CB = (ProExtobjCBAct) UserArrowsCB;
    classCBs->ref_suppress_CB = (ProExtobjCBAct) UserArrowsCB;
    classCBs->ref_delete_CB = (ProExtobjCBAct) UserArrowsCB;
    status = ProExtobjCallbacksSet (obj_class, classCBs);
/*-----*\
    Enable the callback functions. The second argument must not contain
    unsupported actions.
\*-----*/
    status = ProExtobjCBEnable (obj_class,
        PRO_EO_ACT_REF_MODIF|PRO_EO_ACT_REF_SUPPR|PRO_EO_ACT_REF_DELETE,
        PRO_B_TRUE);

    return (PRO_TK_NO_ERROR);
}
/*=====*\
FUNCTION: UserArrowsCB()
PURPOSE:  Callback function for arrows
\*=====*/

```

```

ProError UserArrowsCB (ProExtobj **obj_arr, ProExtobjClass *obj_class,
    ProAppData *appdata_arr, int n_obj)
{
    int status;
    fprintf (stderr, "Callback function called.\n");
    ProMessageDisplay (msg_file, "USER %0s",
        "Callback function called.");
    return (PRO_TK_NO_ERROR);
}

/*=====*\
FUNCTION: UserExtobjRefAdd()
PURPOSE:  Add a reference to an external object.
\*=====*\
ProError UserExtobjRefAdd (ProExtobj *p_obj, ProSelection *p_refsel)
{
    ProWExtobjRef  *p_ref;
    int            status;

    status = ProExtobjRefAlloc (p_ref);
    status = ProExtobjRefselectionSet (*p_ref, *p_refsel);
    status = ProExtobjRefAdd (p_obj, *p_ref);

    return (PRO_TK_NO_ERROR);
}

/*=====*\
FUNCTION: UserMakeSelections()
PURPOSE:  Obtain the ProSurface and ProModelitem for the target surface.
          Also obtain the selected xyz point.
\*=====*\
int UserMakeSelections (ProSurface *p_surf, ProModelitem *p_mdل_item,
    Pro3dPnt xyz, ProSelection **pp_surf_sel)
{
    int            status;        /* return status */
    ProName        msg_file;      /* message file */
    ProSelection    *p_sel;        /* selection */
    int            n_sel;          /* number of selections */

    ProStringToWstring (msg_file, "testmsg.txt");
    ProMessageDisplay (msg_file, "USER %0s", "Select a surface.");

    status = ProSelect ("surface", 1, NULL, NULL, NULL, NULL, &p_sel,
        &n_sel);
    if (status != PRO_TK_NO_ERROR)
    {
        ProMessageDisplay (msg_file, "USER %0s",
            "Error during surface selection.");

        return (status);
    }
}

/*-----*\
    Get the Modelitem from the ProSelection, then get the ProSurface.

```

```

\*-----*/
    status = ProSelectionModelitemGet (*p_sel, p_mdl_item);
    status = ProGeomitemToSurface ((ProGeomitem *) p_mdl_item, p_surf);
/*-----*\
    Get the selected xyz point.
\*-----*/
    status = ProSelectionPoint3dGet (*p_sel, xyz);
/*-----*\
    Set the output ProSelection pointer to p_sel.
\*-----*/
    *pp_surf_sel = p_sel;

    return (PRO_TK_NO_ERROR);
}

```

Event-driven Programming: Toolkit-Based Analysis

This chapter describes the functions that enable you to create analysis and analysis feature objects in a Pro/ENGINEER solid.

Topic	Page
Overview	67 - 2
Interactive Creation of Toolkit-Based Analysis	67 - 3
Interactive Creation of Toolkit-Based Analysis Feature	67 - 3
Storage of Toolkit-Based Analysis Feature in Pro/ENGINEER	67 - 4
Registering a Toolkit-Based Analysis with Pro/ENGINEER	67 - 5
Analysis Callbacks	67 - 5
Pro/TOOLKIT Analysis Information	67 - 8
Results Data	67 - 9
Analysis Attributes	67 - 27
Visiting Saved Toolkit-Based Analyses	67 - 27
Visiting Toolkit-Based Analyses Features	67 - 28
Using the Model without Pro/TOOLKIT	67 - 28

Overview

The Pro/ENGINEER module Behavioral Modeling Extension (BMX) allows the creation of two types of objects in a Pro/ENGINEER solid:

- Analysis
- Analysis Feature

Users create an analysis with the **Analysis** menu from the menu bar. Analyses show the results of certain standard types of measurement or calculation, for example, curvature of an edge or surface, or the center of gravity of a solid. Users can name an analysis and store it in the solid, along with the references to the geometry items it analyses. The analysis is then reevaluated automatically upon each model regeneration, and can be queried at any time using the **Analysis** command. Such an analysis is stored separately from the features and geometry items in the solid.

An analysis feature is a feature that uses an analysis to determine the values of its feature parameters and the shape of its geometry items. An analysis feature is a variety of datum feature and is created using the Pro/ENGINEER **Datum** command. An example of the use of an analysis feature is the creation of a coordinate system datum at the center of gravity of a solid, aligned with its axes of inertia. Another example is a pair of datum points at the closest points of two parts in an assembly.

Pro/TOOLKIT analysis functions allow definition of analyses and analysis features whose computations are performed by callback functions provided by the Pro/TOOLKIT application. This means that Pro/TOOLKIT can be used to make analysis computations, and determine feature geometry, in ways not native to Pro/ENGINEER. We refer to the analyses and analysis features defined by Pro/TOOLKIT as *toolkit-based*; this stresses the fact that the computations could be performed by a separate Pro/TOOLKIT application.

Pro/TOOLKIT users should practice using standard Pro/ENGINEER analyses and analysis features before studying toolkit-based analyses.

The functions and data structures specific to toolkit-based analysis features are declared in the header file *ProAnalysis.h*.

Interactive Creation of Toolkit-Based Analysis

If a Pro/TOOLKIT application registers a *Toolkit-Based Analysis* type, the **Analysis** menu on the Pro/ENGINEER toolbar contains an extra button labeled **Toolkit-Based Analysis**. Click this button to see the Toolkit-Based Analysis dialog box.

The selector at the top of the dialog box shows the types of external analysis registered by the Pro/TOOLKIT application.

When you have chosen the type, click the **Analysis** UI button; this calls the Pro/TOOLKIT callback, which prompts the user for the information needed by the analysis. When you have answered all the prompts, click **Compute**.

Compute performs the analysis and displays the resulting text, if any, in the text area of the dialog box. It may also display some graphics.

When you have clicked **Compute**, you may also click **Info**, which displays the output text in an information window.

The **Saved Analyses** and **Close** buttons behave as for standard analyses.

Event-driven
Programming:
Toolkit-Based Analysis

Interactive Creation of Toolkit-Based Analysis Feature

On the Pro/ENGINEER top menu bar, select the command **Insert > Model Datum > Analysis**. This displays the Pro/ENGINEER ANALYSIS dialog box, which leads the user through the definition of the elements of the analysis feature. If a Pro/TOOLKIT application is running and has registered at least one type of toolkit-based analysis, there will be an additional button labeled **Toolkit-Based Analysis** in the *Type* section.

Set the name of the analysis first, set the type to *Toolkit-Based Analysis*, and continue to the next step, which is called “Definition.” The Toolkit-Based Analysis dialog box appears. This behaves exactly as for an external analysis, until you click the **Close** button. It then returns to the ANALYSIS dialog box, to allow you to continue specifying the feature elements.

The remaining elements are *Result Params* and *Result Datums*. These behave exactly as for standard analysis features, except that the parameters and datums are defined by the Pro/TOOLKIT application. One or other may be absent if the toolkit-based analysis defines no parameters or no datums.

Storage of Toolkit-Based Analysis Feature in Pro/ENGINEER

A toolkit-based analysis feature is stored in Pro/ENGINEER in exactly the same way as any other feature. It appears in the model tree as a feature of type *Analysis*, and all the regular *Feature* commands can be used on it.

The references to existing geometry that the feature needs to calculate its own parameters and geometry are given to Pro/ENGINEER by the Pro/TOOLKIT application in the form of *ProSelection* structures. Pro/ENGINEER stores these references, using the standard method for storing feature references. This means that the Pro/TOOLKIT application does not need to store this information, and that the feature automatically has the correct behavior under the following Pro/ENGINEER commands:

- **Regenerate**—Pro/ENGINEER knows from the feature references what other features it depends on, and therefore whether the features needs to be included in a particular regeneration.
- **Feature Reroute**
- **Feature Pattern**

The Pro/TOOLKIT application may also give Pro/ENGINEER the values of any variables that control the geometry, and Pro/ENGINEER stores them as feature dimensions in the new feature. The Pro/TOOLKIT application can then read the current dimension values when recomputing the feature geometry. This means the feature correctly responds to dimension changes as a result of, for example, being driven by a relation.

The Pro/TOOLKIT application must store as external data any information about toolkit-based analysis features that is not stored as either geometry references or dimensions.

Registering a Toolkit-Based Analysis with Pro/ENGINEER

Function Introduced:

- **ProAnalysisTypeRegister()**

The function **ProAnalysisTypeRegister()** registers a toolkit-based analysis with Pro/ENGINEER by specifying its type name and the set of callback functions to be used when creating it and performing the computation. Call this function in **user_initialize()**.

If called correctly, both the **Analysis** menu from the menu bar and the ANALYSIS dialog box used in creating an analysis feature will include the button **Toolkit-Based Analysis**. Refer to the sections 'Interactive Creation of Toolkit-Based Analysis' and 'Interactive Creation of Toolkit-Based Analysis Feature' for more information on creating these objects.

Event-driven
Programming:
Toolkit-Based Analysis

Analysis Callbacks

When registering a *Toolkit-Based Analysis* type, callbacks must be provided for each of the following 13 types:

- ui
- dims
- infoalloc
- infofree
- compcheck
- compute
- display
- output
- savecheck
- infosave
- inforetrieve
- infocopy
- result

Each callback is passed an argument of the *ProAnalysis* type, which is an opaque handle and identifies the analysis information stored by Pro/ENGINEER.

The following table explains when the callbacks are called, and how each one should be used.

When Pro/ENGINEER creates a toolkit-based analysis or analysis feature:

Callback	Description
infoalloc	Allocate memory for the Pro/TOOLKIT application information about the toolkit-based analysis.
ui	Pro/TOOLKIT prompts the user for inputs that define the analysis—for example, select a surface datum point on which to position a csys.
compcheck	Tell Pro/ENGINEER whether the computation can be performed. If the Pro/TOOLKIT application cannot perform the computation (for example, because input data is unavailable), it returns an error and the regeneration fails.
compute	Perform the analysis computation and store the results in memory.
display	Display graphics showing the computation result.
output	Pass a set of text lines to Pro/ENGINEER for display in the ANALYSIS dialog box to show the result of the computation.
infocopy	Copy the application information from an existing analysis to a new one. Call <i>infocopy</i> during creation because of the way in which Pro/ENGINEER handles feature creation.
dims	Pro/TOOLKIT gives Pro/ENGINEER a list of double values needed to calculate the geometry. Pro/ENGINEER stores these as model dimensions.

Callback	Description
result	Pro/TOOLKIT gives Pro/ENGINEER a description of the feature parameters and geometry items that result from the computation of the analysis. Pro/ENGINEER may also call this callback when it needs to know only the number and names of parameters and datums; an example is when the user selects Feature Info . For more details, refer to the section Results Data.

When the Pro/ENGINEER user saves the analysis to the solid:

Callback	Description
savecheck	Tell Pro/ENGINEER whether the description of the analysis can be saved.
infosave	Give Pro/ENGINEER a list of geometry items referenced by the analysis. Pro/ENGINEER stores these using its own internal mechanism for storing references. The references appear in the model as feature references, and are used to determine the relationship of the feature to other features, and therefore when the feature needs to be regenerated. OR Store any other data as external data.

When the Pro/ENGINEER user retrieves a solid containing analyses:

Callback	Description
inforetrieve	Pro/ENGINEER provides an array of <i>ProSelection</i> objects representing the geometry references it stored with the analysis. (This means that the Pro/TOOLKIT application does not need to save these references between sessions—Pro/ENGINEER uses its own mechanism.)

When the Pro/ENGINEER user leaves the ANALYSIS dialog box without saving the new analysis, or erases a solid containing a toolkit-based analysis:

Callback	Description
infofree	The Pro/TOOLKIT application frees the memory used by its internal description of the analysis.

Many of the callbacks will be called during other commands in Pro/ENGINEER whenever the toolkit-based analysis or analysis feature is affected.

Pro/TOOLKIT Analysis Information

Functions Introduced:

- **ProAnalysisInfoGet()**
- **ProAnalysisInfoSet()**
- **ProAnalysisTypeGet()**

The Pro/TOOLKIT application must keep its own description of the analysis in memory in order to perform the computation. This memory is allocated, filled, and freed in the callbacks provided, as described in the section ‘Analysis Callbacks’.

The callback-calling sequence may vary depending upon exactly what the Pro/ENGINEER user does. This means Pro/TOOLKIT applications should not use global variables to pass the “current” analysis from one callback to another.

Instead of such variables, Pro/ENGINEER can store a pointer to the Pro/TOOLKIT data stored inside the *ProAnalysis* object. Use function **ProAnalysisTypeGet()** to return the type of a specified analysis. The *infoalloc* callback should end with a call to **ProAnalysisInfoSet()**, which gives Pro/ENGINEER the pointer to the new description. Every other callback that needs to use this data should call **ProAnalysisInfoGet()** to get a pointer to the application data for the analysis object, rather than assume this by context.

Pro/ENGINEER cannot access the user data, nor can it provide general storage of this data in the Pro/ENGINEER file when the solid is saved. As explained in the section “Analysis Callbacks” earlier, the callbacks *infosave* and *inforetrieve* respectively give to Pro/ENGINEER, and return after retrieval, geometry references contained in the information; double values can be stored as dimensions. Note that any other information the application needs to store should be saved and retrieved inside *infosave* and *inforetrieve* using Pro/TOOLKIT external data.

Results Data

This section describes in more detail the data given to Pro/ENGINEER by the Pro/TOOLKIT application as the output from the *results* callback.

The output consists of two arrays, one for the feature parameters, the other for the feature geometry. Each of these is a *ProArray* allocated by Pro/ENGINEER before calling the callback.

The structure for a feature parameter is:

```
typedef struct analysis_param
{
    ProName      name;
    ProBoolean    create;
    ProLine      description;
    ProParamvalue *values;
} ProAnalysisParameter;
```

The name is that of the feature parameter that will be created. The *create* flag shows the default setting of the **Create** option for the parameter in the ANALYSIS dialog box. The description appears alongside the parameter in the ANALYSIS dialog box. The array of values should have only a single item in it in the current release. The structure *ProParamvalue* is the same one used for accessing user parameters through the functions in *ProParameter.h* and *ProParamvalue.h*. The value type must be “double” in the current release.

The structure for a geometry item is:

```
typedef struct analysis_geom
```

```

{
    ProName name;
    ProBoolean create;
    ProAnalysisEntityType type;
    ProAnalysisEntity *shapes;
} ProAnalysisGeomitem;

```

The name is given to the resulting datum (NOT to the feature), but with a numerical suffix to ensure that the name is unique in the Pro/ENGINEER model. The *create* flag is exactly as for parameters.

The entity type is an enum which is a subset of the object types in *ProType*. The types supported by this release are:

PRO_ANALYSIS_CURVE	Curve
PRO_ANALYSIS_CSYS	Coord csys
PRO_ANALYSIS_POINT	Datum point
PRO_ANALYSIS_COMP_CRV	Composite curve
PRO_ANALYSIS_SURFACE	Surface

Array “shapes” contains any number of geometric entities of the same type. The union that represents an entity shape is:

```

typedef union
{
    ProAnalysisSrfData    *surface;
    ProQuiltdata          *quilt;
    ProCurvedata         *curve;
    ProCsysdata           csys;
} ProAnalysisEntity;

```

The fields in this union all have types that are generic geometry types in Pro/TOOLKIT, and are declared in the appropriate headers: *ProSurfacedata.h*, *ProQuiltdata.h*, *ProCurvedata.h*, *ProCsysdata.h*, *ProEdgedata.h*. The first three fields, although they are pointers, are not opaque: in spawn mode or asynchronous mode they point to memory in the Pro/TOOLKIT process, not in the Pro/ENGINEER process. It is recommended that you build these structures using the functions in the corresponding header files, for example, **ProCurvedataAlloc()**, **ProLinedataInit()**, and so on.

ProAnalysisSrfData Structure

The *ProAnalysisSrfData* structure is used to define a datum surface from a toolkit-based analysis feature. This structure consists of:

```
typedef struct ProAnalysisSrfData
{
    ProEdgedata    *edge_data;
    ProSurfacedata *pro_surface;
} ProAnalysisSrfData;
```

The *ProSurfacedata** structure contains the surface shape, parameters, and a populated *ProContourdata* structure referencing the boundary edges. The *ProEdgedata** member should be a *ProArray* of edge geometric data whose ids are referenced by the contour data in the surface data structure.

Function Introduced:

- **ProAnalysisSrfDataAlloc()**

The function **ProAnalysisSrfDataAlloc()** allocates the *ProAnalysisSrfData* data structure.

Example 1: Offset Coordinate System Datum

The following example code shows a Pro/TOOLKIT application that defines a toolkit-based analysis that builds a coordinate system datum at an offset from a surface point. The Z axis can be an inward or an outward normal to the surface, the X and Y axes are aligned with the UV mesh lines in the surface, and the origin can be offset in the Z direction. The offset value is stored as a feature dimension.

```
#include "ProToolkit.h"
#include "prodevelop.h"
#include "ProFeature.h"
#include "ProAnalysis.h"
#include "ProArray.h"
#include "ProDtmPnt.h"
#include "ProElement.h"
#include "ProExtdata.h"
#include "ProMessage.h"
#include "ProPoint.h"
#include "ProSection.h"
#include "Pro2dEntdef.h"
#include "ProSelection.h"
```

```

#include "ProSurface.h"
#include <PTApplsUnicodeUtils.h>
#include <UtilString.h>

#include "UtilMath.h"

static ProFileName msgfil;

/*-----*\
    Data structure to describe the information needed by an analysis
    feature that creates a surface sys. The csys is centered at a
    datum point on a surface, with its X and Y axes aligned with the
    U and V mesh lines. The Z axis can be inwards or outwards, by
    user choice, and the origin can be offset from the datum point
    by an offset value that is stored as a feature dimension.
\*-----*/
typedef struct surf_csys_data
{
    /* References */
    ProSelection point; /* The datum point referenced by the csys */

    /* Dimensions */
    double offset;

    /* Application-specific data */
    ProBoolean outwards;

    /* Results storage */
    ProCsysdata csysdata; /* The geometry of the csys */
} Surfcsysdata_t;

int UsrclassRegister(ProMdl model, ProExtdataClass *class);
/*=====*\
FUNCTION : UsrcsurfcsysUiAction()
PURPOSE : Callback for UI action for surface csys
\*=====*/
ProError UsrcsurfcsysUiAction(
    ProAnalysis analysis)
{
    Surfcsysdata_t *data;
    ProError status;
    ProSelection *sel;
    int n_sel;
    ProModelitem modelitem;
    ProPoint p;
    ProDtmptType ptype;
    ProError UsrcYesnoGet();
/*-----*\
    Ask the user to select a datum point on a surface. Check that it is on a
    surface, and reject it if it is not.
\*-----*/

```



```

ProMessageDisplay(msgfil,"USER Select a point on a surface");
while(1)
{
    if(ProSelect("point",1,NULL,NULL,NULL,NULL,&sel,&n_sel)
        != PRO_TK_NO_ERROR || n_sel < 0)
        return(PRO_TK_MSG_USER_QUIT);
    ProSelectionModelitemGet(sel[0], &modelitem);
    ProPointInit(modelitem.owner, modelitem.id, &p);
    if(!UsrPointPtypeGet(modelitem.owner, p, &ptype) ||
        ptype != PRO_DTMPNT_TYPE_ON_SURF)
    {
        ProMessageDisplay(msgfil,"USER That is not a point on surf. Select
            again");
        continue;
    }
    break;
}
}

/*-----*\
Get the pointer to the application data stored for this analysis
feature, and set the datum point in it.
\*-----*/
if(ProAnalysisInfoGet(analysis, (ProAppData*)&data) !=
    PRO_TK_NO_ERROR)
    return(PRO_TK_GENERAL_ERROR);
ProSelectionCopy(sel[0], &data->point);

/*-----*\
Ask the user whether the Z axis is inwards or outwards, and
set the answer in the application data.
\*-----*/
ProMessageDisplay(msgfil,"USER Is the Z axis to be outwards? ||| Yes");
if(UsrYesnoGet(msgfil,"YES",&data->outwards) != PRO_TK_NO_ERROR)
    return(PRO_TK_GENERAL_ERROR);

/*-----*\
Ask the user for the Z offset, and store that, too.
\*-----*/
ProMessageDisplay(msgfil,"USER Enter the Z offset distance|||0.0");
status = ProMessageDoubleRead(NULL, &data->offset);

if(status == PRO_TK_MSG_USER_QUIT)
    return(PRO_TK_GENERAL_ERROR);

if(status != PRO_TK_NO_ERROR)
    data->offset = 0.0;
return(PRO_TK_NO_ERROR);
}

/*=====*\
FUNCTION : UsrSurfcsysDimsAction()
PURPOSE : Callback for Dims action for surface csys
\*=====*/
ProError UsrSurfcsysDimsAction(ProAnalysis analysis, double **dims)
{

```

```

    Surfcsysdata_t *data;
/*-----*\
Get the application data for this analysis
/*-----*\
    if(ProAnalysisInfoGet(analysis, (ProAppData*)&data) !=
        PRO_TK_NO_ERROR)
        return(PRO_TK_GENERAL_ERROR);
/*-----*\
Add the csys offset value as a feature dimension
/*-----*\
    ProArrayObjectAdd((ProArray*)dims, -1, 1, &data->offset);
    return(PRO_TK_NO_ERROR);
}
/*=====*\
FUNCTION : UsrSurfcsysInfoallocAction()
PURPOSE  : Callback to allocate application data needed to describe
           the surface csys
/*=====*\
ProError UsrSurfcsysInfoallocAction(ProAnalysis analysis)
{
    Surfcsysdata_t *data;
/*-----*\
    Allocate a data structure
/*-----*\
    data = (Surfcsysdata_t*)calloc(1, sizeof(Surfcsysdata_t));
/*-----*\
    Put the data pointer into the analysis
/*-----*\
    ProAnalysisInfoSet(analysis, data);
    return(PRO_TK_NO_ERROR);
}
/*=====*\
FUNCTION : UsrSurfcsysInfofreeAction()
PURPOSE  : Callback to free the application data for surface csys
/*=====*\
ProError UsrSurfcsysInfofreeAction(ProAnalysis analysis)
{
    Surfcsysdata_t *data;
/*-----*\
    Get the data pointer from the analysis
/*-----*\
    if(ProAnalysisInfoGet(analysis, (ProAppData*)&data) !=PRO_TK_NO_ERROR)
        return(PRO_TK_GENERAL_ERROR);
/*-----*\
    Free the data
/*-----*\
    free(data);
/*-----*\
    Set the data pointer in the analysis to NULL
/*-----*\
    ProAnalysisInfoSet(analysis, NULL);

```

```

    return(PRO_TK_NO_ERROR);
}
/*=====*\
FUNCTION : UsrSurfcsysCompcheckAction()
PURPOSE  : Callback to tell Pro/E whether computation can be performed
           for surface csys analysis feature
\*=====*/
ProError UsrSurfcsysCompcheckAction(ProAnalysis analysis)
{
    Surfcsysdata_t *data;
/*-----*\
    Get the analysis data
\*-----*/
    if(ProAnalysisInfoGet(analysis, (ProAppData*)&data) !=PRO_TK_NO_ERROR)
        return(PRO_TK_GENERAL_ERROR);
/*-----*\
    Return NO_ERROR if the datum point selection is set OK
\*-----*/
    return(data->point!=NULL?PRO_TK_NO_ERROR:PRO_TK_GENERAL_ERROR);
}
/*=====*\
FUNCTION : UsrSurfcsysComputeAction()
PURPOSE  : Callback to perform the surface csys computation
\*=====*/
ProError UsrSurfcsysComputeAction(ProAnalysis analysis)
{
    Surfcsysdata_t *data;
    ProModelitem modelitem;
    ProPoint p;
    ProVector pos;
    ProSurface surface;
    ProUvParam uv;
    ProVector normal, der1[2];
    ProFeature feature;
/*-----*\
    Get the application data from the analysis
\*-----*/
    if(ProAnalysisInfoGet(analysis, (ProAppData*)&data) !=PRO_TK_NO_ERROR)
        return(PRO_TK_GENERAL_ERROR);
/*-----*\
    Get the location of the datum point
\*-----*/
    ProSelectionModelitemGet(data->point, &modelitem);
    ProPointInit(modelitem.owner, modelitem.id, &p);
    ProPointCoordGet(p, data->csysdata.origin);
/*-----*\
    Find the surface the datum point sits on, and find its first derivative
    vectors and the normal
\*-----*/
    UsrPointSrfGet(data->point, &surface);
    ProSurfaceParamEval(modelitem.owner, surface, data->csysdata.origin,

```

```

        uv);
    ProSurfaceXYZdataEval(surface, uv, NULL, der1, NULL, normal);
/*-----*\
Flip and offset if necessary
\*-----*/
    if(!data->outwards)
        ProUtilVectorScale(-1.0, normal, normal);
    ProUtilVectorsLincom(1.0, data->csysdata.origin, data->offset, normal,
        data->csysdata.origin);
/*-----*\
Copy this information to the application data as a csys geometry
\*-----*/
    memcpy(data->csysdata.x_vector, der1[0], sizeof(ProVector));
    ProUtilVectorNormalize(data->csysdata.x_vector,
        data->csysdata.x_vector);
    memcpy(data->csysdata.z_vector, normal, sizeof(ProVector));
    ProUtilVectorNormalize(data->csysdata.z_vector,
        data->csysdata.z_vector);
    ProUtilVectorCross(data->csysdata.z_vector,
        data->csysdata.x_vector, data->csysdata.y_vector);
    return(PRO_TK_NO_ERROR);
}
/*=====*\
FUNCTION : UsrSurfcsysDisplayAction()
PURPOSE  : Callback to display the results of the analysis computation
\*=====*/
ProError UsrSurfcsysDisplayAction(ProAnalysis analysis)
{
    Surfcsysdata_t *data;
    ProVector pos;
/*-----*\
Get the application data from the analysis
\*-----*/
    if(ProAnalysisInfoGet(analysis, (ProAppData*)&data) !=PRO_TK_NO_ERROR)
        return(PRO_TK_GENERAL_ERROR);
/*-----*\
Draw a line for the X vector
\*-----*/
    ProGraphicsPenPosition(data->csysdata.origin);
    pos[0]=data->csysdata.origin[0]+data->csysdata.x_vector[0];
    pos[1]=data->csysdata.origin[1]+data->csysdata.x_vector[1];
    pos[2]=data->csysdata.origin[2]+data->csysdata.x_vector[2];
    ProGraphicsLineDraw(pos);
/*-----*\
Draw a line for the Y vector
\*-----*/
    ProGraphicsPenPosition(data->csysdata.origin);
    pos[0]=data->csysdata.origin[0]+data->csysdata.y_vector[0];
    pos[1]=data->csysdata.origin[1]+data->csysdata.y_vector[1];
    pos[2]=data->csysdata.origin[2]+data->csysdata.y_vector[2];
    ProGraphicsLineDraw(pos);

```

```

/*-----*\
Draw a line for the Z vector
/*-----*\
ProGraphicsPenPosition(data->csysdata.origin);
pos[0]=data->csysdata.origin[0]+data->csysdata.z_vector[0];
pos[1]=data->csysdata.origin[1]+data->csysdata.z_vector[1];
pos[2]=data->csysdata.origin[2]+data->csysdata.z_vector[2];
ProGraphicsLineDraw(pos);
return(PRO_TK_NO_ERROR);
}
/*=====*\
FUNCTION : UsrSurfcsysOutputAction()
PURPOSE : Callback to write textual information about the result of
          the computation
/*=====*\
ProError UsrSurfcsysOutputAction(ProAnalysis analysis, ProLine **lines)
{
    Surfcsysdata_t *data;
    ProCharLine buff;
    ProLine ll[4];
/*-----*\
Get the application data from the analysis
/*-----*\
if(ProAnalysisInfoGet(analysis, (ProAppData*)&data) !=PRO_TK_NO_ERROR)
return(PRO_TK_GENERAL_ERROR);
/*-----*\
Write the origin of the csys
/*-----*\
ProTKSprintf(buff,"Csys is at location (%0.2f, %0.2f, %0.2f)",
              data->csysdata.origin[0],
              data->csysdata.origin[1],
              data->csysdata.origin[2]);
ProStringToWstring(ll[0], buff);
/*-----*\
X vector
/*-----*\
ProTKSprintf(buff,"          X vector (%0.2f, %0.2f, %0.2f)",
              data->csysdata.x_vector[0],
              data->csysdata.x_vector[1],
              data->csysdata.x_vector[2]);
ProStringToWstring(ll[1], buff);
/*-----*\
Y vector
/*-----*\
ProTKSprintf(buff,"          Y vector (%0.2f, %0.2f, %0.2f)",
              data->csysdata.y_vector[0],
              data->csysdata.y_vector[1],
              data->csysdata.y_vector[2]);
ProStringToWstring(ll[2], buff);
/*-----*\
Z vector

```

```

\*-----*/
    ProTKSprintf(buff, "          Z vector (%0.2f, %0.2f, %0.2f)",
        data->csysdata.z_vector[0],
        data->csysdata.z_vector[1],
        data->csysdata.z_vector[2]);
    ProStringToWstring(ll[3], buff);
/*-----*\
Add the four lines to the output array
\*-----*/
    ProArrayObjectAdd((ProArray*)lines, -1, 4, ll);
    return(PRO_TK_NO_ERROR);
}
/*=====*\
FUNCTION : UsrSurfcsysSavecheckAction()
PURPOSE  : Callback to tell Pro/E whether the analysis application
           data can be saved
\*=====*/
ProError UsrSurfcsysSavecheckAction(ProAnalysis analysis)
{
    return(PRO_TK_NO_ERROR);
}
/*=====*\
FUNCTION : UsrSurfcsysInfosaveAction()
PURPOSE  : Callback to tell Pro/E what element references are required
           by the analysis feature, and save the rest of the info
           to Ext Data
\*=====*/
ProError UsrSurfcsysInfosaveAction(
    ProAnalysis analysis,
    ProFeature *feature,
    ProSelection **references)
{
    Surfcsysdata_t *data;
    ProSelection reference;
/*-----*\
Get the application data from the analysis
\*-----*/
    if(ProAnalysisInfoGet(analysis, (ProAppData*)&data) !=PRO_TK_NO_ERROR)
        return(PRO_TK_GENERAL_ERROR);
/*-----*\
Output the reference to the datum point
\*-----*/
    ProSelectionCopy(data->point, &reference);
    ProArrayObjectAdd((ProArray*)references, -1, 1, &reference);
/*-----*\
Save the "outwards" option
\*-----*/
    if(!UsrOutwardsSave(feature, data->outwards))
        return(PRO_TK_GENERAL_ERROR);
    return(PRO_TK_NO_ERROR);
}

```

```

/*=====*\
FUNCTION : UsrSurfcsysInforetrieveAction()
PURPOSE  : Callback to get the references Pro/E has stored with the
           analysis feature, and retrieve the rest from Ext Data
\*=====*/
ProError UsrSurfcsysInforetrieveAction(
    ProAnalysis analysis,
    ProFeature *feature,
    ProSelection *references)
{
    Surfcsysdata_t *data;
    int n_dims, *dims;
    PRODIMENSION dim;
/*-----*\
    Get the application data from the analysis
\*-----*/
    if(ProAnalysisInfoGet(analysis, (ProAppData*)&data) !=PRO_TK_NO_ERROR)
        return(PRO_TK_GENERAL_ERROR);
/*-----*\
    Copy the first reference to the application as the datum point
\*-----*/
    ProSelectionCopy(references[0], &data->point);
/*-----*\
    Retrieve the "outwards" option from Ext Data
\*-----*/
    UsrOutwardsRetrieve(feature, &data->outwards);
/*-----*\
    If the feature has a dimension, take its value as the offset
\*-----*/
    n_dims = prodb_get_feat_dim_ids((Prohandle)feature->owner,
                                   feature->id, &dims);

    if(n_dims > 0)
    {
        prodim_get_dimension((Prohandle)feature->owner,
                            dims[0], PRO_DIM_PARAM, &dim);
        data->offset = dim.value;
    }
    return(PRO_TK_NO_ERROR);
}
/*=====*\
FUNCTION : UsrSurfcsysInfocopyAction()
PURPOSE  : Callback to copy application information from one analysis
           to another.
\*=====*/
ProError UsrSurfcsysInfocopyAction(ProAnalysis from, ProAnalysis to)
{
    Surfcsysdata_t *fdata, *tdata;
/*-----*\
    Get the application data from the analyses
\*-----*/
    if(ProAnalysisInfoGet(from, (ProAppData*)&fdata) != PRO_TK_NO_ERROR)

```

```

    return(PRO_TK_GENERAL_ERROR);
    if(ProAnalysisInfoGet(to, (ProAppData*)&tdata) != PRO_TK_NO_ERROR)
    return(PRO_TK_GENERAL_ERROR);
/*-----*\
Copy the application-specific data (everything except references)
\*-----*/
    tdata->outwards = fdata->outwards;
    tdata->offset = fdata->offset;
    return(PRO_TK_NO_ERROR);
}
/*=====*\
FUNCTION : UsrSurfcsysResultAction()
PURPOSE  : Callback to give Pro/E the feature parameters and geometry
           that the analysis feature must contain
\*=====*/
ProError UsrSurfcsysResultAction(
    ProAnalysis analysis,
    ProBoolean names_only,
    ProAnalysisParameter **parameters,
    ProAnalysisGeomitem **geometry)
{
    Surfcsysdata_t *data;
    ProAnalysisGeomitem geomitem;
    ProAnalysisParameter param;
/*-----*\
Get the application data from the analysis
\*-----*/
    if(ProAnalysisInfoGet(analysis, (ProAppData*)&data) !=PRO_TK_NO_ERROR)
    return(PRO_TK_GENERAL_ERROR);
/*-----*\
Specify a CSYS DATUM with the geometry stored in the application data
\*-----*/
    ProStringToWstring(geomitem.name, "SURF_CSYS");
    geomitem.create = PRO_B_TRUE;
    geomitem.type = PRO_ANALYSIS_CSYS;
    if(!names_only)
    {
        ProArrayAlloc(1, sizeof(ProAnalysisEntity), 1,
            (ProArray*)&geomitem.shapes);
        memcpy(geomitem.shapes[0].csys.origin, data->csysdata.origin,
            sizeof(ProVector));
        memcpy(geomitem.shapes[0].csys.x_vector, data->csysdata.x_vector,
            sizeof(ProVector));
        memcpy(geomitem.shapes[0].csys.y_vector, data->csysdata.y_vector,
            sizeof(ProVector));
        memcpy(geomitem.shapes[0].csys.z_vector, data->csysdata.z_vector,
            sizeof(ProVector));
    }
/*-----*\
Output the csys geometry description
\*-----*/

```



```

    ProArrayAlloc(0, sizeof(ProAnalysisGeomitem), 1, (ProArray*)geometry);
    ProArrayObjectAdd((ProArray*)geometry, -1, 1, &geomitem);
    return(PRO_TK_NO_ERROR);
}

int UsrExtAnalysisRegister ()
{
    ProAnalysisFuncsData data;
    ProStringToWstring(msgfil, "umsg.txt");
/*-----*\
Build the description of the surface csys analysis feature
data, and register it with Pro/E
\*-----*/
    memset(&data, '\0', sizeof(ProAnalysisFuncsData));
    ProStringToWstring(data.type, "Surface Csys");
    data.ui_action          = UsrSurfcsysUiAction;
    data.dims_action        = UsrSurfcsysDimsAction;
    data.infoalloc_action   = UsrSurfcsysInfoallocAction;
    data.infofree_action    = UsrSurfcsysInfofreeAction;
    data.compcheck_action   = UsrSurfcsysCompcheckAction;
    data.compute_action     = UsrSurfcsysComputeAction;
    data.display_action     = UsrSurfcsysDisplayAction;
    data.output_action      = UsrSurfcsysOutputAction;
    data.savecheck_action   = UsrSurfcsysSavecheckAction;
    data.infosave_action    = UsrSurfcsysInfosaveAction;
    data.inforetrieve_action = UsrSurfcsysInforetrieveAction;
    data.infocopy_action    = UsrSurfcsysInfocopyAction;
    data.result_action      = UsrSurfcsysResultAction;
    ProAnalysisTypeRegister(&data);
    return(0);
}

/*=====*\
FUNCTION : UsrPointPtypeGet()
PURPOSE  : Utility to get the placement type of a datum point
\*=====*/
int UsrPointPtypeGet(
    ProSolid owner,
    ProPoint p,
    ProDtmpntType *ptype)
{
    ProElempathItem items[5];
    ProElempath epath;
    ProValue value;
    ProValueData vdata;
    int id;
    ProGeomitem geomitem;
    ProFeature feature;
/*-----*\
Get the feature that owns the point
\*-----*/
    ProPointIdGet(p, &id);
    ProModelitemInit(owner, id, PRO_POINT, &geomitem);

```

```

    ProGeomitemFeatureGet(&geomitem, &feature);
/*-----*\
Get the placement type from the element tree
\*-----*/
    items[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
    items[0].path_item.elem_id = PRO_E_DPOINT_POINTS_ARRAY;

    items[1].type = PRO_ELEM_PATH_ITEM_TYPE_INDEX;
    items[1].path_item.elem_index = 0;

    items[2].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
    items[2].path_item.elem_id = PRO_E_DPOINT_PLA_CONSTRAINTS;

    items[3].type = PRO_ELEM_PATH_ITEM_TYPE_INDEX;
    items[3].path_item.elem_index = 0;

    items[4].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
    items[4].path_item.elem_id = PRO_E_DPOINT_PLA_CONSTR_TYPE;

    ProElempathAlloc(&epath);
    ProElempathDataSet(epath, items, 5);
    if(ProFeatureElemValueGet(&feature, epath, &value) != PRO_TK_NO_ERROR)
        return(0);
    if(ProValueDataGet(value, &vdata) != PRO_TK_NO_ERROR)
        return(0);
    *ptype = (ProDtmpntType)vdata.v.i;
    return(1);
}
/*-----*\
FUNCTION : UsrPointSrfGet()
PURPOSE : Utility to get the surface a datum point is placed on.
\*-----*/
int UsrPointSrfGet(
    ProSelection point,
    ProSurface *surface)
{
    ProElempathItem items[5];
    ProElempath epath;
    ProValue value;
    ProValueData vdata;
    int id;
    ProGeomitem geomitem;
    ProFeature feature;
    ProSelection sel;
/*-----*\
Get the feature that owns the datum point
\*-----*/
    ProSelectionModelitemGet(point, &geomitem);
    ProGeomitemFeatureGet(&geomitem, &feature);
/*-----*\
Get the surface from the element tree

```

```

\*-----*/
items[0].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
items[0].path_item.elem_id = PRO_E_DPOINT_POINTS_ARRAY;

items[1].type = PRO_ELEM_PATH_ITEM_TYPE_INDEX;
items[1].path_item.elem_index = 0;

items[2].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
items[2].path_item.elem_id = PRO_E_DPOINT_PLA_CONSTRAINTS;

items[3].type = PRO_ELEM_PATH_ITEM_TYPE_INDEX;
items[3].path_item.elem_index = 0;

items[4].type = PRO_ELEM_PATH_ITEM_TYPE_ID;
items[4].path_item.elem_id = PRO_E_DPOINT_PLA_CONSTR_REF;

ProElempathAlloc(&epath);
ProElempathDataSet(epath, items, 5);

if(ProFeatureElemValueGet(&feature, epath, &value) != PRO_TK_NO_ERROR)
return(0);

if(ProValueDataGet(value, &vdata) != PRO_TK_NO_ERROR)
return(0);

sel = (ProSelection)vdata.v.r;
ProSelectionModelitemGet(sel, &geomitem);
ProSurfaceInit(geomitem.owner, geomitem.id, surface);
return(1);
}
\*=====*\
FUNCTION : UsrYesnoGet()
PURPOSE  : Prompt for a yes/no answer
\*=====*\
ProError UsrYesnoGet(
    wchar_t *msgfil,
    char *def,          /* INPUT : default answer */
    ProBoolean *yes)/* Output: TRUE = yes, FALSE no */
{
    wchar_t wreply[4];
    char reply[4];
    ProError status;
    while(1)
    {
\*-----*\
        Read string from keyboard
\*-----*\
        status = ProMessageStringRead(4, wreply);
        if(status == PRO_TK_MSG_USER_QUIT)
            return(PRO_TK_MSG_USER_QUIT);
\*-----*\
    }
}

```

```

Convert reply (or default) to uppercase C string
/*-----*/
    if(status == PRO_TK_NO_ERROR)
        ProWstringToString(reply,wreply);
    else
        strcpy(reply, def);
        ProUtilStringUpper(reply,reply);
/*-----*/
Output response
/*-----*/
    if(reply[0] == 'Y')
    {
        *yes = PRO_B_TRUE;
        return(PRO_TK_NO_ERROR);
    }
    if(reply[0] == 'N')
    {
        *yes = PRO_B_FALSE;
        return(PRO_TK_NO_ERROR);
    }
/*-----*/
If reply was neither yes or no, ask again.
/*-----*/
    ProMessageDisplay(msgfil,"USER Please answer Y or N : ");
}
}

#define CLASS_NAME "SURFCSYS"

/*=====*\
FUNCTION : UsrOutwardsSave()
PURPOSE  : Save the "outwards" flag to external data
/*=====*\
int UsrOutwardsSave(
    ProFeature *feature,
    ProBoolean outwards)
{
    ProExtdataClass class;
    ProCharName slotname;
    if(!UsrClassRegister(feature->owner, &class))
        return(0);
    ProTKSprintf(slotname,"OUTWARDS_%d",feature->id);
    UsrIntegerSave(&class, slotname, (int)outwards);
    return(1);
}
/*=====*\
FUNCTION : UsrOutwardsRetrieve()
PURPOSE  : Retrieve the outwards flag from external data
/*=====*\
int UsrOutwardsRetrieve(
    ProFeature *feature,

```

```

        ProBoolean *outwards)
{
    ProExtdataClass class;
    ProCharName slotname;
    int i;

    if(!UsrClassExists(feature->owner, CLASS_NAME, &class))
        return(0);

    ProTKSprintf(slotname, "OUTWARDS_%d", feature->id);
    if(!UsrIntegerRetrieve(&class, slotname, &i))
        return(0);

    *outwards = i==0 ? PRO_B_FALSE : PRO_B_TRUE;

    return(1);
}
/*=====*\
FUNCTION : UsrClassRegister()
PURPOSE  : Register a data class, if it does not already exist
\*=====*/
int UsrClassRegister(
    ProMdl model,
    ProExtdataClass *class)
{
    ProName cname;

    if(!UsrClassExists(model, CLASS_NAME, class))
    {
        ProStringToWstring(cname, CLASS_NAME);
        if(ProExtdataClassRegister(model, cname, class) !=
            PROEXTDATA_TK_NO_ERROR)
            return(0);
    }
    return(1);
}
/*=====*\
FUNCTION : UsrIntegerSave()
PURPOSE  : Utility to save an integer slot
\*=====*/
int UsrIntegerSave(
    ProExtdataClass *class,
    char *slotname,
    int i)
{
    ProExtdataSlot slot;
    ProName sname;

    /*-----*\
    If the specified slot does not exist, create it
    \*-----*/

```

```

    ProStringToWstring(sname, slotname);
    if(ProExtdataSlotCreate(class, sname, &slot) !=
PROEXTDATA_TK_NO_ERROR)
    {
        slot.p_class = class;
        ProStringToWstring(slot.slot_name, slotname);
    }
/*-----*\
Write the integer to the data slot
\*-----*/
    ProExtdataSlotWrite(&slot, KEY_BY_NAME, PRO_INTEGER_TYPE, 0, &i);
    return(1);
}
/*=====*\
FUNCTION : UsrIntegerRetrieve()
PURPOSE  : Retrieve an integer
\*=====*/
int UsrIntegerRetrieve(
    ProExtdataClass *class,
    char *slotname,
    int *i)
{
    ProExtdataSlot slot;
    int *data;
    int size, type;
/*-----*\
Read the integer from the data slot
\*-----*/
    slot.p_class = class;
    ProStringToWstring(slot.slot_name, slotname);
    if(ProExtdataSlotRead(&slot, KEY_BY_NAME, &type, &size, (void**)&data)
        != PROEXTDATA_TK_NO_ERROR)
        return(0);
    *i = *data;
    return(1);
}
/*=====*\
FUNCTION : UsrClassExists
PURPOSE  : Check if the specified class exists
\*=====*/
int UsrClassExists(
    ProMdl model,
    char *name,
    ProExtdataClass *class)
{
    ProExtdataErr status;
    int n_classes, c;
    ProName *classes;
    status = ProExtdataClassNamesList(model, &n_classes, &classes);
    if(status != PROEXTDATA_TK_NO_ERROR)
        return(0);

```

```

for(c=0;c<n_classes;c++)
{
    if(!ProUtilStrwscmp(name, classes[c]))
    {
        class->p_model = model;
        ProStringToWstring(class->class_name, name);
        ProArrayFree((ProArray *)&classes);
        return(1);
    }
}
ProArrayFree((ProArray *)&classes);
return(0);
}

```

Analysis Attributes

Functions Introduced:

- **ProAnalysisAttrSet()**
- **ProAnalysisAttrIsSet()**

These functions allow you to get and set certain attributes on a toolkit-based analysis. The only attribute defined in the current release is PROANALYSIS_COMPUTE_OFF. If this is set, the *compute* and *result* callbacks will not be called during regeneration of the model. If the toolkit-based analysis belongs to a feature, the geometry of the feature will be frozen until the PROANALYSIS_COMPUTE_OFF is unset again. If the feature geometry includes a surface curve, the 3D location of the curve will be recalculated during regeneration in accordance with the existing UV curves definition but using the new geometry of the surface. This means the curve remains in the surface even if the surface moves while COMPUTE_OFF is TRUE.

Use the PROANALYSIS_COMPUTE_OFF attribute to temporarily turn off the toolkit-based analysis to save time when making other changes to the model.

Visiting Saved Toolkit-Based Analyses

Functions Introduced:

- **ProSolidAnalysisVisit()**
- **ProAnalysisNameGet()**

The function **ProSolidAnalysisVisit()** visits a saved toolkit-based analysis in a part or assembly. It does not visit standard saved analyses, nor toolkit-based analyses in features.

The function **ProAnalysisNameGet()** provides the name under which a toolkit-based analysis is saved.

Visiting Toolkit-Based Analyses Features

Functions Introduced:

- **ProFeatureAnalysisGet()**

To visit analyses in analysis features, use **ProSolidFeatureVisit()**, filter for features whose type is **PRO_FEAT_ANALYSIS**, and call **ProFeatureAnalysisGet()** on each feature.

Using the Model without Pro/TOOLKIT

If a model contains external analyses or external analysis features of a type defined by a Pro/TOOLKIT application, and that model is retrieved into Pro/ENGINEER while the Pro/TOOLKIT application is not running, Pro/ENGINEER will not attempt to recompute those analyses during regeneration. Saved toolkit-based analyses will retain their old values, and toolkit-based analysis features will have their geometry frozen.

If a Pro/TOOLKIT application is terminated during a Pro/ENGINEER session, any toolkit-based analysis types it registered will be automatically deregistered, and any analyses that use those types will be frozen.

68

Event-driven Programming: Foreign Datum Surfaces and Curves

This chapter describes the Pro/TOOLKIT functions that enable you to create foreign datum surfaces and curves.

Topic	Page
Foreign Datum Surfaces	68 - 2
Foreign Datum Curves	68 - 3

Foreign Datum Surfaces

Functions Introduced:

- **prosrf_bind_srf_class()**
- **user_init_srf()**
- **user_eval_srf()**
- **user_rmov_srf()**

Pro/TOOLKIT enables you to define your own surface types, known to Pro/ENGINEER as foreign datum surfaces. Pro/ENGINEER supports the use of foreign datum surfaces as the boundaries of solid parts in the same way it supports surfaces that are native to Pro/ENGINEER.

Each foreign datum surface has a class and a name:

- *Class*—This is a string of up to 10 characters (such as “Parabolic”). It is permanently associated with each surface upon surface creation, but is not unique to one surface. There is a set of user-supplied functions associated with each class of foreign surfaces, defined as initialization, evaluation, and termination functions. These functions must be invoked before a class can be recognized by Pro/ENGINEER. All surfaces of a class share the same functions.
- *Name*—This is a string of up to 40 characters (such as “right wing, section b2”). The name identifies a particular instance of a surface in a class. It is permanently associated with each surface upon surface creation, and is unique to each surface in the part. It is used to identify the surface, display the surface name, and show information about the surface.

Importing a Foreign Datum Surface

This section describes the way that a Pro/ENGINEER user would import a foreign datum surface.

Foreign surfaces are a type of datum surface that are displayed by drawing the outline of the surface extents. You select a foreign surface by selecting the surface name, just as you would for a datum surface.

Before importing a surface, you must have a coordinate system available in the part to which to reference the surface.

To Import a Foreign Datum Surface

1. Choose **Insert->Foreign Surface**.
Note: This button is present only when one or more surface classes are registered.
2. Choose **Import**. If no classes have been initialized, the **Import** option of the datum surface menu will be disabled.
3. Select the class of the surface from the SURF CLASS menu. This menu lists all classes that have been properly initialized.
4. Enter the foreign datum surface name.
5. Select the name of the coordinate system from the namelist menu that the surface will reference.

You can replace part surfaces with foreign datum surfaces using the **Replace** option in the offset tool. Once the foreign surface is incorporated into your part geometry, all other geometric operations, such as **Cut** or **Round**, are available for the surface.

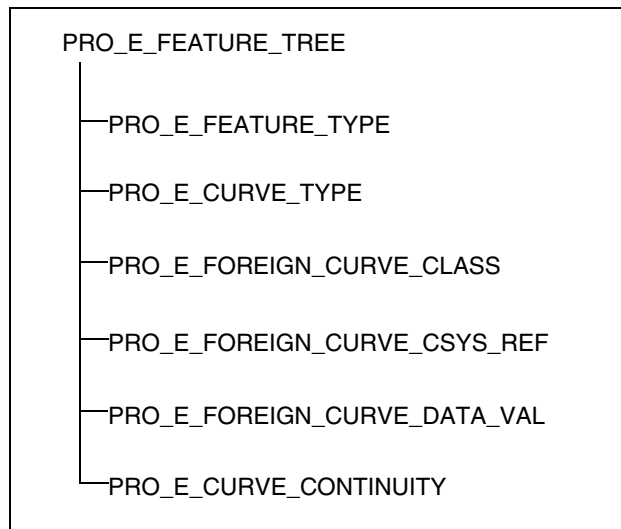
Note: PTC recommends that you use DLL mode to get decent performance, because many calls are made by Pro/ENGINEER to the evaluation function.

Foreign Datum Curves

In Pro/TOOLKIT, you create foreign datum curves using the feature creation techniques described in the chapter 'Element Trees: Principles of Feature Creation'. The header file *ProForeignCurve.h* contains the element tree structure and a table that maps each element to an element identifier, value type, and valid values.

The following figure shows the element tree structure for foreign datum curve creation. Note that all elements are required.

Figure 68-1: Element Tree for Foreign Datum Curve



As the element tree implies, foreign datum curve creation requires that you provide the feature type, curve type, curve class, reference coordinate system, data used in the analytical representation of the curve, and curve continuity. Pro/ENGINEER uses this information, together with an evaluation function, to create an internal representation of the curve.

Providing an Evaluation Function

Function Introduced:

- **ProForeignCurveEvalFunction()**

In addition to building the element tree for your datum curve feature, you must provide its analytical representation to Pro/ENGINEER. This representation is made available to Pro/ENGINEER in a special function called an *evaluator*, or *evaluation function*.

The evaluation function must contain parameterized equations for the X, Y, and Z coordinates of points that define the curve. If $C(X,Y,Z)$ is a function representing the curve in three-dimensional space, you can represent the parameterized equations for each coordinate as follows:

$$X = f(t)$$

$$Y = g(t)$$

$$Z = h(t)$$

In these equations, the parameter t ranges from 0 to 1 over the extent of the curve. For example, a parametric representation of a circle of radius R lying in the XY-plane, whose center coincides with the origin, is as follows:

```
X = R*cos(2*PI*t);
Y = R*sin(2*PI*t);
Z = 0;
```

In these equations, $PI = 3.14159$.

Pro/TOOLKIT provides the prototype for the evaluation function. The syntax is as follows:

```
typedef ProError (*ProForeignCurveEvalFunction)
(
    ProName      class,          /* input */
    wchar_t      *data_string,    /* input */
    ProSelection  csys,          /* input */
    double        curve_param,    /* input */
    ProVector     xyz_point,      /* output */
    ProVector     deriv1,         /* output */
    ProVector     deriv2         /* output */
);
```

The function arguments are as follows:

- *class*—Identifies the type of curves generated by the evaluation function.
- *data_string*—The flag that controls specific attributes of the curve.
- *csys*—The reference coordinate system with respect to which the curve geometry is defined. Pass it to the evaluation function as a *ProSelection* object.
- *curve_param*—The parameter value at which the X, Y, and Z coordinates, as well as the first and second derivatives, will be evaluated.

- *xyz_point*—The X, Y, and Z coordinates at the value of *curve_param*.
- *deriv1*—The values of the first derivatives of X, Y, and Z with respect to the parameter, at the value of *curve_param*.
- *deriv2*—The values of the second derivatives of X, Y, and Z with respect to the parameter, at the value of *curve_param*.

All arguments are passed to the evaluation function by Pro/ENGINEER, based on the values you provide for the elements in the element tree.

A single evaluation function can be used to create a number of curve variations within a given class. The parameterized curve equations typically contain constants whose values control the shape, size, location, and orientation of the curve. You can write the evaluation function such that, depending on the value of the *data_string* argument, different values of those constants will be used to calculate the location of points on the curve.

Curve Continuity

Curve continuity, in a sense, defines the smoothness of intersections between the ends of the foreign curve and other geometry in the model. It also defines the continuity of three-dimensional geometry created from the curve, such as a swept surface. First-order continuity implies that the first derivatives of two adjoining curve segments are equal at the point at which the curves join. Second-order continuity is similarly defined. Depending on the curve continuity you want, the evaluator function needs to contain first and second derivatives of the parameterized curve equations.

You specify the curve continuity using the PRO_E_CURVE_CONTINUITY element in the element tree. The valid values, contained in the enumerated type *ProForeignCrvCont*, are as follows:

- PRO_FOREIGN_CURVE_CALC_XYZ
- PRO_FOREIGN_CURVE_CALC_XYZ_1_DER
- PRO_FOREIGN_CURVE_CALC_XYZ_1_AND_2_DER

These values correspond to zeroth-, first-, and second-order continuity, respectively. If you use the value `PRO_FOREIGN_CURVE_CALC_XYZ`, Pro/ENGINEER passes NULL for *deriv1* and *deriv2* to the evaluation function. Similarly, if you use the value `PRO_FOREIGN_CURVE_CALC_XYZ_1_DER`, Pro/ENGINEER passes NULL for *deriv2* to the evaluation function. Therefore, you should check for NULL values of *deriv1* and *deriv2* in your evaluation function before trying to assign derivative values to them.

Pro/ENGINEER calls your evaluation function multiple times for a series of values of the curve parameter, ranging from 0 to 1. The function outputs the following information:

- X, Y, and Z coordinates of the curve at the specified parameter value
- Values of the first and second derivatives, as needed for the desired curve continuity

These values are then used by Pro/ENGINEER to construct the curve.

Binding the Evaluation Function to a Class

Function Introduced:

- **ProForeignCurveClassEvalSet()**

The evaluation function must be bound to a class. This is done with a call to the function **ProForeignCurveClassEvalSet()**. The function takes as arguments the class name and a pointer to the evaluation function. If you call **ProForeignCurveClassEvalSet()** and pass NULL for the evaluation function pointer, it unbinds a previously bound evaluation function from the class.

Example 1: Creating a Sinusoidal Foreign Datum Curve

The following code example shows how to use the Pro/TOOLKIT functions to create a sinusoidal foreign datum curve.

```
/*=====*\
FUNCTION: ProTestForeignDatumCurve()
PURPOSE: Foreign datum curve evaluation function
\*=====*/
void ProTestForeignDatumCurve()
{
    wchar_t      data[PRO_NAME_SIZE];
    ProName      class;
    int          status, sel_num;
    static wchar_t msgfil[PRO_LINE_SIZE];
```

```

ProSelection    *p_sel_list, csys_ref, sel_model;
ProMdl          model;
ProModelitem    model_item;
ProElement      elem_tree, elem_ftype, elem_crv_type,
                elem_crv_class;
ProElement      elem_crv_csys_ref, elem_crv_data_string,
                elem_crv_continuity;
ProValueData    value_data;
ProValue        value;
xyz_point[1] =  amplitude*(sin(wave_num*2.0*PI*curve_param));
xyz_point[2] = 0.0;

if (deriv1 != NULL)
{
    deriv1[0] = wave_num*2.0*PI;
    deriv1[1] = (wave_num*2.0*PI)*amplitude *
                cos (wave_num*2.0*PI*curve_param);
    deriv1[2] = 0.0;
}

if (deriv2 != NULL)
{
    deriv2[0] = 0.0;
    deriv2[1] = -(wave_num*wave_num*4.0*PI*PI) *
                amplitude*sin(wave_num*2.0*PI*curve_param);
    deriv2[2] = 0.0;
}
return (0);
}

void ProTestForeignDatumCurve()
{
    wchar_t      data[PRO_NAME_SIZE];
    ProName      class;
    int          status, sel_num;
    static       wchar_t msgfil[PRO_LINE_SIZE];
    ProSelection *p_sel_list, csys_ref, sel_model;
    ProMdl        model;
    ProModelitem  model_item;
    ProElement    elem_tree, elem_ftype, elem_crv_type, elem_crv_class;
    ProElement    elem_crv_csys_ref, elem_crv_data_string,
                elem_crv_continuity;
    ProValueData  value_data;
    ProValue      value;
    ProFeature    crv_feature;
    ProErrorlist  errors;
/*-----*\
    Set the class type and curve data.
\*-----*/
    ProStringToWstring (class, "SIN_CURVE");
/*-----*\
    Bind the foreign curve evaluation function to the curve class.

```



```

\*-----*/
    if ((status = ProForeignCurveClassEvalSet (class,
        (ProForeignCurveEvalFunction)TestCurveFunction)) !=
        PRO_TK_NO_ERROR)
    {
        printf ("ProForeignCurveClassEvalSet returned %d\n", status);
    }
\*-----*/
    Create the selection object for the current model to use in
ProFeatureCreate().
\*-----*/
    status = ProMdlCurrentGet (&model);
    status = ProMdlToModelitem (model, &model_item);
    status = ProSelectionAlloc (NULL, &model_item, &sel_model);

    ProMessageDisplay (msgfil, "USER Enter data set for refining curve
        (short, long, shallow, deep)");
    ProMessageStringRead (PRO_NAME_SIZE, data);

    ProStringToWstring (msgfil, "foreign_dtm_curve.txt");
    ProMessageDisplay (msgfil, "USER Select a reference coordinate
        system");
\*-----*/
    Obtain the ProSelection object for the reference coordinate system.
\*-----*/
    status = ProSelect ("csys", 1, NULL, NULL, NULL, NULL,
        &p_sel_list, &sel_num);
    status = ProSelectionAlloc (NULL, NULL, &csys_ref);
    status = ProSelectionCopy (p_sel_list[0], &csys_ref);
\*-----*/
    Create the element tree.
\*-----*/
    status = ProElementAlloc (PRO_E_FEATURE_TREE, &elem_tree);
\*-----*/
    Allocate the feature type element.
\*-----*/
    status = ProElementAlloc (PRO_E_FEATURE_TYPE, &elem_ftype);
\*-----*/
    Set the value of the feature type element.
\*-----*/
    value_data.type = PRO_VALUE_TYPE_INT;
    value_data.v.i = PRO_FEAT_CURVE;
    status = ProValueAlloc (&value);
    status = ProValueDataSet (value, &value_data);
    status = ProElementValueSet (elem_ftype, value);
\*-----*/
    Add the feature type element as a child of the root of the tree.
\*-----*/
    status = ProElemtreeElementAdd (elem_tree, NULL, elem_ftype);
\*-----*/
    Allocate the curve type element.
\*-----*/

```

```

    status = ProElementAlloc (PRO_E_CURVE_TYPE, &elem_crv_type);
/*-----*\
    Set the value of the curve type element.
\*-----*/
    value_data.type = PRO_VALUE_TYPE_INT;
    value_data.v.i = PRO_CURVE_TYPE_FOREIGN;
    status = ProValueAlloc (&value);
    status = ProValueDataSet (value, &value_data);
    status = ProElementValueSet (elem_crv_type, value);
/*-----*\
    Add the curve type element as a child of the root of the tree.
\*-----*/
    status = ProElemtreeElementAdd (elem_tree, NULL, elem_crv_type);
/*-----*\
    Allocate the foreign curve class element.
\*-----*/
    status = ProElementAlloc (PRO_E_FOREIGN_CURVE_CLASS,
        &elem_crv_class);
/*-----*\
    Set the value of the foreign curve class element.
\*-----*/
    value_data.type = PRO_VALUE_TYPE_WSTRING;
    value_data.v.w = class;
    status = ProValueAlloc (&value);
    status = ProValueDataSet (value, &value_data);
    status = ProElementValueSet (elem_crv_class, value);
/*-----*\
    Add the foreign curve class element as a child of the root of
    the tree.
\*-----*/
    status = ProElemtreeElementAdd (elem_tree, NULL, elem_crv_class);
/*-----*\
    Allocate the foreign curve csys reference element.
\*-----*/
    status = ProElementAlloc (PRO_E_FOREIGN_CURVE_CSYS_REF,
        &elem_crv_csys_ref);
/*-----*\
    Set the value of the foreign curve csys reference element.
\*-----*/
    value_data.type = PRO_VALUE_TYPE_SELECTION;
    value_data.v.r = csys_ref;
    status = ProValueAlloc (&value);
    status = ProValueDataSet (value, &value_data);
    status = ProElementValueSet (elem_crv_csys_ref, value);
/*-----*\
    Add the foreign curve csys reference element as a child of the
    root of the tree.
\*-----*/
    status = ProElemtreeElementAdd (elem_tree, NULL,
        elem_crv_csys_ref);

```

```

/*-----*\
    Allocate the foreign curve data value element.
\*-----*/
    status = ProElementAlloc (PRO_E_FOREIGN_CURVE_DATA_VAL,
        &elem_crv_data_string);
/*-----*\
    Set the value of the foreign curve data value element.
\*-----*/
    value_data.type = PRO_VALUE_TYPE_WSTRING;
    value_data.v.w = data;
    status = ProValueAlloc (&value);
    status = ProValueDataSet (value, &value_data);
    status = ProElementValueSet (elem_crv_data_string, value);
/*-----*\
    Add the foreign curve data value element as a child of the root
    of the tree.
\*-----*/
    status = ProElemtreeElementAdd (elem_tree, NULL,
        elem_crv_data_string);
/*-----*\
    Allocate the foreign curve continuity element.
\*-----*/
    status = ProElementAlloc (PRO_E_FOREIGN_CURVE_CONTINUITY,
        &elem_crv_continuity);
/*-----*\
    Set the value of the foreign curve continuity element.
\*-----*/
    value_data.type = PRO_VALUE_TYPE_INT;
    value_data.v.i = PRO_FOREIGN_CURVE_CALC_XYZ;
    status = ProValueAlloc (&value);
    status = ProValueDataSet (value, &value_data);
    status = ProElementValueSet (elem_crv_continuity, value);
/*-----*\
    Add the foreign curve continuity element as a child of the root
    of the tree.
\*-----*/
    status = ProElemtreeElementAdd (elem_tree, NULL,
        elem_crv_continuity);
/*-----*\
    Create the foreign datum curve.
\*-----*/
    status = ProFeatureCreate (sel_model, elem_tree, NULL, 0,
        &crv_feature, &errors);
/*-----*\
    Free the allocated selection objects.
\*-----*/
    ProSelectionFree (&csys_ref);
    ProSelectionFree (&sel_model);
}

```


Task Based Application Libraries

Applications created using the different Pro/ENGINEER API products are interoperable. These products use Pro/ENGINEER as the medium of interactions, eliminating the task of writing native -platform specific interactions between different programming languages.

Topic	Page
ProArgument and Argument Management	69 - 2
Creating Pro/TOOLKIT DLL Task Libraries	69 - 3
Launching Synchronous J-Link Applications	69 - 17

ProArgument and Argument Management

Use the data structure ProArgument to pass application data to and from tasks in other applications. The declaration for this structure is:

```
typedef struct pro_argument
{
    ProName      label;
    ProValueData value;
} ProArgument;
```

The *ProValueData* structure supports the following argument types:

- Integer
- Double
- String (char*)
- String (wchar_t*)
- Boolean
- ProSelection
- ProMatrix

Do not use the value type PRO_VALUE_TYPE_POINTER (provided with this structure in order to support feature element tree values) when passing arguments between applications.

Functions Introduced:

- **ProArgumentByLabelGet()**
- **ProValuedataStringSet()**
- **ProValuedataWstringSet()**
- **ProValuedataTransformGet()**
- **ProValuedataTransformSet()**
- **ProArgumentProarrayFree()**

Use the function **ProArgumentByLabelGet()** to locate an argument within a ProArray of ProArgument structures passed between applications.

Use the function **ProValuedataStringSet()** to allocate and copy memory into the ProValuedata structure for a char* argument. Using this function ensures that **ProArgumentProarrayFree()** releases all memory in an arguments array.

Use the function **ProValuedataWstringSet()** to allocate and copy memory into the ProValuedata structure for a wchar_t* argument. Use this function to ensure that **ProArgumentProarrayFree()** releases all memory in an arguments array.

Use the function **ProValuedataTransformSet()** to allocate and copy memory into the ProValuedata structure for a ProMatrix argument. Use this function to ensure that **ProArgumentProarrayFree()** releases all memory in an arguments array.

Use the function **ProValuedataTransformGet()** to copy a ProMatrix value into a local variable. The matrix data is not directly accessible from the double** member of the ProValuedata structure.

Use the function **ProArgumentProarrayFree()** to completely free an array of ProArgument structures.

Creating Pro/TOOLKIT DLL Task Libraries

Functions that are intended to act as Pro/TOOLKIT task library functions must have the same signature as **ProTkdllFunction()**:

```
typedef ProError (*ProTkdllFunction) (
    ProArgument* inputs, ProArgument** outputs );
```

Use the preprocessor macro **PRO_TK_DLL_EXPORT** with any function that must be accessible to external applications. This macro provides platform-specific instructions to the compiler to make the function visible to other external applications. This macro must be placed in the function prototype, if it exists, and in the function definition if the prototype does not exist.

Some platforms require externally visible symbols to be declared on the application link line.

Memory Management in Task Library Functions

To avoid memory leaks or overwrites, the DLL functions must use proper memory management. The DLL function must:

- Ensure that the contents of the input ProArgument array are not freed. This is taken care of by the calling application and the Pro/TOOLKIT communications code.
- Use the **ProValuedata*Set()** functions to assign values to the output ProArgument array. This allows the calling application to free the output array with **ProArgumentProarrayFree()**.

Example 1: An Exported Toolkit Task Function

The following example function demonstrates how you can set up and implement a Pro/TOOLKIT task function accessible from other Pro/ENGINEER auxiliary applications. It uses the required signature and macro for task functions. It parses the input argument array to search for parameters required by the task.

```
#include <ProToolkit.h>
#include <ProMdl.h>
#include <ProArray.h>
#include <ProModelitem.h>
#include <ProImportfeat.h>
#include <ProToolkitDll.h> /* for ProArgument */
#include <ProWstring.h>
#include <UtilString.h>
#include <ProUtil.h>

/*-----*\
Prototypes
\*-----*/
ProIntfData* PTestPrismCreate (double x_value, double y_value, double
                               z_value, ProSolid solid);
ProIntfData* PTestSphereCreate (double, ProMdl);

typedef enum
{
  USER_IMPFEATOPT_PRISM,
  USER_IMPFEATOPT_SPHERE
} UserImpfeatopt;

/*=====*\
FUNCTION: UserCreateImpfeatTask
PURPOSE: Example task function in a Pro/TOOLKIT DLL
\*=====*/
PRO_TK_DLL_EXPORT ProError UserCreateImpfeatTask (ProArgument* inputs,
                                                    ProArgument** outputs)
{
```



```

ProError status;
ProSolid solid;
ProArgument arg;
ProValueData data;
UserImpfeatopt type;
ProMdlType model_type;
ProIntfData *interface_data;
ProImportfeatAttr attr;
ProCsys csys = NULL;
ProIntfDataSource data_source;
ProFeature feature;
ProSelection sel;

attr.join_surfaces = PRO_B_TRUE;
attr.attempt_make_solid = PRO_B_TRUE;
attr.cut_or_add = PRO_B_FALSE;

status = ProMdlCurrentGet ((ProMdl*)&solid);
if (status != PRO_TK_NO_ERROR)
return PRO_TK_BAD_CONTEXT;

status = ProMdlTypeGet (solid, &model_type);
if (model_type != PRO_MDL_PART)
return PRO_TK_INVALID_TYPE;

/*-----*\
Look up required input arguments (by name)
\*-----*/
/*status = ProArgumentByLabelGet (inputs, L"IMPGEOMTYPE", &data);*/
status = UserArgumentByLabelGet (inputs, L"IMPGEOMTYPE", &data);
if (status != PRO_TK_NO_ERROR)
return PRO_TK_EMPTY;

type = (UserImpfeatopt)data.v.i;

switch (type)
{
case USER_IMPFEATOPT_PRISM:
{
double length, width, height;

/* Characteristic inputs: L, W, H */
/* status = ProArgumentByLabelGet (inputs, L"LENGTH", &data); */
status = UserArgumentByLabelGet (inputs, L"LENGTH", &data);
if (status != PRO_TK_NO_ERROR)
return PRO_TK_BAD_INPUTS;

length = data.v.d;

/* status = ProArgumentByLabelGet (inputs, L"WIDTH", &data); */
status = UserArgumentByLabelGet (inputs, L"WIDTH", &data);

```

```

if (status != PRO_TK_NO_ERROR)
return PRO_TK_BAD_INPUTS;

width = data.v.d;

/* status = ProArgumentByLabelGet (inputs, L"HEIGHT", &data); */
status = UserArgumentByLabelGet (inputs, L"HEIGHT", &data);
if (status != PRO_TK_NO_ERROR)
return PRO_TK_BAD_INPUTS;

height = data.v.d;

/*-----*\
Create interface data describing a rectangular prism with given dimensions
\*-----*/
interface_data = PTTestPrismCreate (length, width, height, solid);

break;
}

case USER_IMPFEATOPT_SPHERE:
{
double radius;

/* status = ProArgumentByLabelGet (inputs, L"RADIUS", &data); */
status = UserArgumentByLabelGet (inputs, L"RADIUS", &data);
if (status != PRO_TK_NO_ERROR)
return PRO_TK_BAD_INPUTS;

radius = data.v.d;

/*-----*\
Create interface data describing a spherical boundary with given dimension
\*-----*/
interface_data = PTTestSphereCreate (radius, solid);

break;
}
}

/*-----*\
Look up coordinate system (optional argument to this task)
\*-----*/
/* status = ProArgumentByLabelGet (inputs, L"CSYS", &data); */
status = UserArgumentByLabelGet (inputs, L"CSYS", &data);
if (status == PRO_TK_NO_ERROR)
{
ProSelection sel;
ProModelitem item;

sel = data.v.r;

```

```

ProSelectionModelitemGet (sel, &item);
ProGeomitemToCsys (&item, &csys);
}

/*-----*\
Look for cut flag
\*-----*/
/* status = ProArgumentByLabelGet (inputs, L"CUT", &data); */
status = UserArgumentByLabelGet (inputs, L"CUT", &data);
if (status == PRO_TK_NO_ERROR)
{
if (data.v.b)
attr.cut_or_add = 1;
}

/*-----*\
Create import feature with parameters
\*-----*/
status = ProIntfDataSourceInit (PRO_INTF_NEUTRAL, interface_data,
                                &data_source);

status = ProImportfeatCreate ( solid, &data_source, csys, &attr,
                                &feature);

if (status != PRO_TK_NO_ERROR)
return PRO_TK_GENERAL_ERROR;

/* status = ProArgumentByLabelGet (inputs, L"NAME", &data); */
status = UserArgumentByLabelGet (inputs, L"NAME", &data);
if (status == PRO_TK_NO_ERROR)
{
ProModelitemNameSet (&feature, data.v.w);
}

/*-----*\
Note that function does not free input arguments (memory belongs
to calling application)
\*-----*/

/*-----*\
Send output arguments containing selection of created feature
\*-----*/
status = ProArrayAlloc (0, sizeof (ProArgument), 1, (ProArray*)outputs);
if (status == PRO_TK_NO_ERROR)
{
ProSelectionAlloc (NULL, &feature, &sel);

ProStringToWstring (arg.label, "NEW_FEATURE");
arg.value.type = PRO_VALUE_TYPE_SELECTION;
arg.value.v.r = sel;

```

```

ProArrayObjectAdd( (ProArray*)outputs, -1, 1, &arg );

return PRO_TK_NO_ERROR;
}
else
return PRO_TK_GENERAL_ERROR;
}

```

Launching Pro/TOOLKIT DLL Functions

Functions Introduced:

- **ProToolkitDllLoad()**
- **ProToolkitTaskExecute()**
- **ProToolkitDllUnload()**
- **ProToolkitDllIdGet()**
- **ProToolkitDllHandleGet()**

Use the function **ProToolkitDllLoad()** to register and start a Pro/TOOLKIT DLL. The input parameters of this function are similar to the fields of a registry file and are as follows:

- *app_name*—The name of the application to initialize.
- *exec_file*—The DLL file to load, including its full path.
- *text_dir*—The path to the application's message and UI text files.
- *user_display*—Set this parameter to PRO_B_TRUE if you want the interactive user to be able to see the application registered in the Pro/ENGINEER User Interface and to see error messages if the application fails.

The function outputs a handle to the loaded DLL in the form of a *ProToolkitDllHandle*. If the function fails to load the DLL, the function outputs information describing the failure and the application's **user_initialize()** function is called.

Use the function **ProToolkitTaskExecute()** to call a properly designated function of the Pro/TOOLKIT DLL library. You can pass arbitrary combinations of input arguments to the library function.

Use the function **ProToolkitDllUnload()** to shutdown a Pro/TOOLKIT DLL previously loaded by **ProToolkitDllLoad()**. The application's **user_terminate()** function is called.

Use the function **ProToolkitDllIdGet()** to get a string representation of the DLL application. The string representation can be sent to other applications, which can use **ProToolkitDllHandleGet()** to obtain the Pro/TOOLKIT DLL handle using this string representation. Pass NULL to the first argument of **ProToolkitDllIdGet()** to get the identifier for the calling application.

Custom Pro/TOOLKIT DLL Tasks for Distributed Pro/BATCH

You can create a customized Distributed Pro/BATCH task by providing a Pro/TOOLKIT DLL to be run by the Distributed Pro/BATCH service. The steps required to create and deploy a custom DLL task for Distributed Pro/BATCH are as follows:

- Compose the Pro/TOOLKIT DLL containing a function that meets the criteria for a custom DLL task function. Refer to the “Creating Pro/TOOLKIT DLL Task Libraries” section for the function signature. Use the standard Distributed Pro/BATCH arguments while coding the function. Refer to the “Coding a Custom DLL Task Function” section for more information on the arguments.
- Create a DLL registry file for Distributed Pro/BATCH. Refer to the “Registry File for Custom DLL Tasks” section for more information.
- Create a TTD (Task Type Definition) file specifying the Pro/TOOLKIT DLL and the custom task function to be executed by the custom task. Refer to the “TTD File Format for Custom DLL Tasks” section for more information.

Coding a Custom DLL Task Function

Distributed Pro/BATCH provides the following input arguments to the custom DLL task function:

Note: All the input arguments are of the widest string data type.

- **DBS_WORKING_DIRECTORY**—Specifies the full path of the working directory for the Distributed Pro/BATCH service. It stores all the .log, .inf, and output files generated during the execution of a custom task.
- **DBS_CURRENT_OBJECT_NAME**—Specifies the name of the object that will be acted upon by the custom DLL task function.

- **DBS_CURRENT_OBJECT_TYPE**—Specifies the type of the object such as a 3D model, assembly, or drawing. The value of this argument will be a widestring containing an integer value of the type that matches with the values in the enumerated type *ProMdlType*.

Distributed Pro/BATCH supplies any additional input arguments and their values, if they are listed in a TTD file, as

<USER_DATA></USER_DATA> tags within the
<TKFUNC></TKFUNC> node. Each user data tag is supplied as an independent argument.

The custom DLL task function can influence the state of Distributed Pro/BATCH and the status of the task through one of the following output arguments. If an argument is not supplied, the current or default value is used.

Note: All the output arguments are of the widestring data type.

- **DBS_CURRENT_OBJECT_NAME**—Specifies the name of the generated object. This output informs Distributed Pro/BATCH that a model with this name should be used for other TTD entries that follow.
- **DBS_CURRENT_OBJECT_TYPE**—Specifies the type of the generated object such as a 3D model, assembly, or drawing. The value of this argument will be a widestring containing an integer value of the type that matches with the values in the enumerated type *ProMdlType*. This output informs Distributed Pro/BATCH that a model of this type should be used for other TTD entries that follow.
- **DBS_IGNORE_FILE**—By default, the Distributed Pro/BATCH service returns any file generated in the working directory, except for report type files such as .log, .inf and .txt. The names of the files specified by this output argument will not be included in the output returned to the client. You can return more than one argument of this type.
- **DBS_OUTPUT_FILE**—Some file types such as .log, .inf and .txt are not transferred to the Distributed Pro/BATCH client by default. File names specified by this output argument indicate that a particular file such as trail.txt should be passed back to the client. This file is always returned even if Distributed Pro/BATCH ignores it. This argument is not required for files returned by default. You can return more than one argument of this type.

- **DBS_OUTPUT_DIRECTORY**—Specifies the output directory containing all files except ones such as trail files or log files that are ignored by Distributed Pro/BATCH. This optional argument is required only if the contents of the output directory are different from the input working directory.
- **DBS_MESSAGE**—Specifies the user-visible message that will be printed in the log file generated by the custom DLL task function. This message is especially useful if the function returns an error.

In case of chained TTD tasks (described in the “TTD File Format for Custom DLL Tasks” section), the output arguments generated by the first TTD containing the custom DLL task function are supplied as input arguments to the second TTD within the same file.

The custom DLL task function should return one of the following types of errors:

- **PRO_TK_NO_ERROR**—Specifies that the task function executed successfully.
- **PRO_TK_BAD_INPUTS**—Specifies that the task function was called with incorrect input arguments.

Registry File for Custom DLL Tasks

Registering a custom Pro/TOOLKIT DLL means providing information to the Distributed Pro/BATCH service about the files that form the DLL.

An example of the registry file is as follows:

```
name pt_userguide.dll
exec_file <TK_LOADPOINT>/<MACHINE>/obj/pt_userguide.dll
text_dir <TK_LOADPOINT>/protk_appls/pt_userguide/text
```

where <TK_LOADPOINT> refers to the directory that forms the loadpoint of Pro/TOOLKIT under the Pro/ENGINEER installation.

The fields of the above registry file are described below:

- *name*—Specifies the name of the Pro/TOOLKIT DLL to be used as the `dllname` attribute in the TTD file.
- *exec_file*—Specifies the full path to the DLL binary file on the service machine.
- *text_dir*—Specifies the directory containing language-specific directories that include the menu and message files used by the DLL.

The registry file must be named `dbatchs.dat`. The DLL registry file must be visible to the Distributed Pro/BATCH service on the service machine. To make it visible, place the `dbatchs.dat` file in the same location as the Distributed Pro/BATCH service executable `dbatchs.exe` available in the `<proe_loadpoint>/<machine_type>/obj` directory. More than one custom DLL can be listed in a given `dbatchs.dat` file.

TTD File Format for Custom DLL Tasks

TTD files serve as templates for all the common tasks that can be performed using Distributed Pro/BATCH. Refer to the Distributed Pro/BATCH Help for complete information about the nodes included in a TTD file.

For a custom DLL task, specify the name of the Pro/TOOLKIT DLL as the `dllname` attribute and the function

ProToolkitTaskExecute() that causes Pro/ENGINEER to execute the task function as the `func` attribute within the `<TKFUNC></TKFUNC>` node in a TTD file. Specify the name of the custom DLL task function in the `<DLL_FUNCTION></DLL_FUNCTION>` tag within the `<TKFUNC></TKFUNC>` node. You can also include the required `<USER_DATA></USER_DATA>` tags within the `<TKFUNC></TKFUNC>` node.

Additionally, you can specify an existing Pro/TOOLKIT function along with its input arguments and enumerated types within a `<TKFUNC></TKFUNC>` node. A TTD file containing multiple `<TKFUNC></TKFUNC>` nodes is called a **chained TTD**.

Example 2: Chained TTD Task to convert a Pro/ENGINEER model into a simplified representation in the VRML format

Following is an example of a TTD file for a custom DLL task function.

```
<TTD version="1.0" created_by="PTC">
  <DESCRIPTION>Load and call external DLL to change the
  current model to simprep</DESCRIPTION>
  <DETAILS>Load an external Pro/TK DLL into the
  Pro/ENGINEER session</DETAILS>
  <SERVICE name="dbatchs" />
  <TKFUNC func="ProToolkitTaskExecute"
    dllname="pt_userguide.dll">
    <DLL_FUNCTION>TestSimprepActivateTask</DLL_FUNCTION>
    <USER_DATA name="SIMP_REP">NO_B</USER_DATA>
  </TKFUNC>
  <!-- Export to VRML format -->
  <TKFUNC func="ProExportVRML">
```



```

    </TKFUNC>
</TTD>

```

In the above `simprep.ttd` file, the function **ProToolkitTaskExecute()** executes the custom task function **TestSimrepActivateTask** defined in `pt_userguide.dll`. The function **TestSimrepActivateTask** converts a Pro/ENGINEER model into a simplified representation. The name of the simplified representation to be created must be specified in the `<USER_DATA></USER_DATA>` tag.

The activated simplified representation is converted into the VRML format by the second `<TKFUNC></TKFUNC>` node entry in the chained TTD task. This part of the execution is directly handled by Distributed Pro/BATCH.

The C language code for the custom task function **TestSimrepActivateTask** used in the `simprep.ttd` file is as follows:

```

#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
/*----- Pro/Toolkit Includes -----*/

#include <ProToolkit.h>
#include <ProMenu.h>
#include <ProMdl.h>
#include <ProIntf3Dexport.h>
#include <ProShrinkwrap.h>
#include <ProFaminstance.h>
#include <ProSolid.h>
#include <ProToolkitDll.h>
#include <ProSimprep.h>
#include <ProDrawing.h>
#include <ProUtil.h>
#include <ProModelitem.h>
#include <ProWstring.h>
#include <ProTKRunTime.h>
/*----- Application Includes -----*/
#include <TestError.h>

#define MSGFIL L##"msg_uginterface.txt"

static ProError UserDBSObjectTypeGet(ProArgument*
    input_arguments, ProName label, int* object_type);

static ProError UserDBSObjectNameGet (ProArgument*
    input_arguments, ProName label, ProName name);

/*=====*\

```

```

FUNCTION: UserSimprepActivateTask()
PURPOSE:  Activates a simplified representation of the
           current Distributed Pro/BATCH model. This
           simplified rep will be considered the current
           model for any subsequent tasks in the TTD.
\*=====*/

PRO_TK_DLL_EXPORT ProError
UserSimprepActivateTask(ProArgument* input_args,
                        ProArgument**output_args)
{
ProName object_name;
ProFileName input_file;
ProMdl model_handle;
int mdl_type;
ProSimprep simp_rep;
ProFileName file_name, file;
ProName rep_name;
ProError status;

mdl_type = PRO_MDL_UNUSED;

/*-----*\
Get the object name and its type loaded through Dbatch.
\*-----*/

status = UserDBSObjectNameGet (input_args,
                               L"DBS_CURRENT_OBJECT_NAME", object_name);

status = UserDBSObjectTypeGet
        (input_args, L"DBS_CURRENT_OBJECT_TYPE", &mdl_type);
if (mdl_type == PRO_MDL_UNUSED)
{
    status = PRO_TK_BAD_INPUTS;
    return status;
}

/*-----*\
Initialise the model handle.
\*-----*/

status = ProMdlInit(object_name, (ProMdlType)mdl_type,
                    &model_handle);

/*-----*\
Get the USER_DATA from TTD passed by user.
\*-----*/
status = UserDBSObjectNameGet (input_args,
                               L"SIMP_REP", rep_name);

/*-----*\
Initialise the simplified representation.

```

```

/*-----*/
status = ProSimpInit(rep_name,-1,
                    (ProSolid)model_handle, &simp_rep);

/*-----*\
  Activate the specified simplified representation
  \*-----*/
status =
    ProSimpActivate((ProSolid)model_handle,&simp_rep);

status = ProMdlEraseNotDisplayed();

return status;
}

/*=====*\
FUNCTION: UserDBSObjectNameGet()
PURPOSE:  Get the object name loaded through Dbatch.
\*=====*/

ProError UserDBSObjectNameGet (ProArgument*
                             input_arguments, ProName label, ProName name)
{

ProValueData data;
int length;
ProError status;

/*-----*\
  Get the value of argument
  \*-----*/

status = ProArgumentByLabelGet(input_arguments, label,
                              &data);

if (status != PRO_TK_NO_ERROR)
    return status;

/*-----*\
  Data will always be of wstring type
  \*-----*/

if (data.type != PRO_VALUE_TYPE_WSTRING)
{
    status = PRO_TK_INVALID_TYPE;
    return status;
}
ProWstringLengthGet (data.v.w, &length);

if (length > PRO_NAME_SIZE -1)
{
    status = PRO_TK_LINE_TOO_LONG;

```

```

return status;
}

/*-----*\
Copy the data in output value
\*-----*/

ProWstringCopy (data.v.w, name, PRO_VALUE_UNUSED);
status = PRO_TK_NO_ERROR;
return status;

}

/*=====*\
FUNCTION: UserDBSObjectTypeGet()
PURPOSE:  Get the object type loaded through Dbatch.
\*=====*/
ProError UserDBSObjectTypeGet (ProArgument*
input_arguments, ProName label, int* object_type)
{
ProValueData data;
char  model_type[PRO_NAME_SIZE];
int length;
ProError status;

/*-----*\
Get the value of argument
\*-----*/
status = ProArgumentByLabelGet (input_arguments, label,
&data);

if (status != PRO_TK_NO_ERROR)
return status;

/*-----*\
Data will always be of wstring type
\*-----*/

if (data.type != PRO_VALUE_TYPE_WSTRING)
{
status = PRO_TK_INVALID_TYPE;
return status;
}

ProWstringLengthGet (data.v.w, &length);

if (length > PRO_NAME_SIZE -1)
{
status = PRO_TK_LINE_TOO_LONG;
return status;
}

```

```

/*-----*\
Copy the data in output value
\*-----*/

ProWstringToString (model_type,data.v.w);

*object_type = atoi(model_type);

status = PRO_TK_NO_ERROR;

return status;
}
#undef MSGFIL

```

Launching Synchronous J-Link Applications

The functions described in this section allow Pro/TOOLKIT API users to launch a synchronous J-Link application and call methods within it with user-specified arguments.

The ability to launch and control a J-Link application by a Pro/TOOLKIT API enables:

- Code reuse through the ability to create J-Link libraries that can be called from Pro/TOOLKIT or other APIs.
- Creation of platform independent library routines.

Functions Introduced:

- **ProJlinkApplicationStart()**
- **ProJlinkTaskExecute()**
- **ProJlinkApplicationStop()**

Use the function **ProJlinkApplicationStart()** to register and start a J-Link application. The input parameters of this function are similar to the fields of a registry file and are as follows:

- *app_name*—Assigns a unique name to this J-Link application.
- *java_app_class*—Specifies the fully qualified class and package name of the Java class that contains the J-Link application's start and stop method.
- *java_app_start*—Specifies the start method of the program.
- *java_app_stop*—Specifies the stop method of the program.

- *java_app_add_classpath*—Specifies the locations of packages and classes that can be loaded when running a Java program. Can be NULL, if not needed.
- *text_dir*—Specifies the application text path for menus and messages. Can be NULL if the application does not use menus or messages.
- *user_display*—Specifies whether to display the application in the **Auxiliary Applications** dialog box in Pro/ENGINEER.

The function provides the *ProJlinkAppHandle* handle to the JLink application. If the start method throws an exception, the description of the exception is stored in the argument *startup_exception*.

The function **ProJlinkTaskExecute()** calls a registered task in a J-Link application. The input parameters are:

- *handle*—Specifies the handle to the J-Link application.
- *task_id*—Specifies the task to be executed. The J-Link application should register the task using the J-Link method **pfcSession.BaseSession.RegisterTask()**. Refer to J-Link documentation for more information on the RegisterTask() method.
- *input_args*—Specifies the input arguments to be passed to the task.

The output of this function is an array of arguments of type ProArgument. These arguments are returned by the J-Link task method.

If the method throws an exception, the description of the exception is stored in the output argument *exception*.

The function **ProJlinkApplicationStop()** stops the J-Link application specified by the *ProJlinkAppHandle* application handle. The function activates the application's stop method.

If the stop method throws an exception, the description of the exception is stored in the output argument *exception*.



Summary of Technical Changes

This appendix contains a list of new and enhanced functions for Pro/TOOLKIT under Pro/ENGINEER Wildfire 5.0. See the APIWizard online browser for complete descriptions of the functions.

Each release of Pro/TOOLKIT includes a README file in the loadpoint directory. Check the README file for the most current information.

Topic	Page
Critical Technical Changes	A - 2
New Functions	A - 6
Superseded Functions	A - 26
Miscellaneous Technical Changes	A - 29

Note: Reference information on all functions is available in the Pro/TOOLKIT APIWizard online browser. Use the APIWizard **Search** function to find the information on a function. See the Online Documentation — Pro/TOOLKIT APIWizard section for information on the APIWizard.

Critical Technical Changes

This section describes the changes in Pro/ENGINEER Wildfire 5.0 and Pro/TOOLKIT that might require alteration of existing Pro/TOOLKIT applications.

Compiling and Linking on Windows

All Pro/TOOLKIT applications on 32-bit and 64-bit Windows platforms must be built using the Microsoft Visual Studio .NET 2008 compiler. A new library `ws2_32.lib` has been added. The CC flags no longer need the `-D_USE_32BIT_TIME_T` flag on 32-bit Windows platforms. These changes have been incorporated in the sample makefiles.

3D Import and 2D PDF Export Using Profiles

3D file formats can now be imported using a profile. A 3D profile is an XML file with the extension `.dip` (Dex In Profile) that contains all the options required for the 3D import operation. New functions **ProIntfimportModelWithProfileCreate()** and **ProImportfeatWithProfileCreate()** have been added in *ProIntfimport.h* and *ProImportfeat.h*, respectively. Refer to the “Interface: Data Exchange” chapter for the complete list of 3D formats that can be imported using a profile.

2D PDF export now supports profiles. A 2D PDF profile is an XML file with the extension `.dop` (Dex Out Profile). It contains all the options required for a 2D PDF export operation. The options settings used during a 2D PDF export can be saved as the profile. A new function **ProPDFWithProfileExport()** has been added in *ProPDF.h*.

Creating Section Constraints

Pro/TOOLKIT for Pro/ENGINEER Wildfire 5.0 M050 supports the Intent Manager mode for a section. Pro/TOOLKIT applications that include calls to function **ProSectionConstraintCreate()** built in Pro/TOOLKIT for Pro/ENGINEER Wildfire 5.0 M040 or earlier, will return `PRO_TK_BAD_CONTEXT` when used with Pro/ENGINEER Wildfire 5.0 M050 and later. From Pro/TOOLKIT for Pro/ENGINEER Wildfire 5.0 M050 onwards, in order to use the function **ProSectionConstraintCreate()** on any section, first set the Intent Manager mode for that section to ON using the function **ProSectionIntentManagerModeSet()**.

Mechanica Updates

Displacement Constraints

The data structure `ProMechDisplacementConstr` for displacement constraints in *ProMechConstraint.h* has been deprecated. Functions previously available to access Mechanica displacement constraints have also been deprecated. Use the new data structure `ProMechDisplacementRegularConstr` and new functions that access this data structure instead.

Pin, Ball, and Planar Constraints

Pin, Ball, and Planar constraints are now supported in the Pro/ENGINEER Mechanica environment. The following existing enumerated types have been deprecated:

- `PRO_MECH_DISPLACEMENT_CYL_ANGULAR`
- `PRO_MECH_DISPLACEMENT_CYL_AXIAL`
- `PRO_MECH_DISPLACEMENT_CYL_RADIAL_ONLY`

The above listed types can only be set as the new enumerated type `PRO_MECH_DISPLACEMENT_PIN`. New functions to access the new constraint types have also been added to *ProMechConstraint.h*.

Mechanica Lite

Pro/TOOLKIT does not provide access to **Mechanica Lite**. The function **ProMechanicaEnter()** returns a new error type `PRO_TK_CANT_ACCESS` while accessing this mode.

Shell Pair Types

In the Pro/ENGINEER Mechanica environment in both FEM and Native modes, a shell pair with multi-constant thickness can now be distinguished from the shell pair with variable thickness. The shell pair with multi-constant thickness is represented by a new enumerated type `PRO_MECH_SHELL_PAIR_MULTI_CONSTANT` under `ProMechShlpairType` in *ProMechShellPair.h*. However, the variable thickness type has been obsoleted from the Native mode of Pro/ENGINEER Mechanica. Hence, in Native mode, if `PRO_MECH_SHELL_PAIR_VARIABLE` is used as one of the input arguments for the function **ProMechshlpairrefsTypeSet()**, the function returns an error type `PRO_TK_BAD_INPUTS`.

Modifications to Sketched Features

Pro/ENGINEER Wildfire 5.0 provides separate sketching tools for creating construction points, construction centerlines, construction coordinate system, geometry points, geometry centerlines, and a geometry coordinate system. The geometry entities are used to convey feature level information outside the Sketcher environment, while the construction entities are used as references within the Sketcher environment.

Prior to this release, the Pro/ENGINEER construction entities also conveyed feature-level information. Hence, earlier Pro/TOOLKIT applications using functions associated with feature-level information may fail. To address this issue, convert the affected construction entities to geometry entities using the function **ProSectionEntityConstructionSet()** with the input argument construction set to PRO_B_FALSE.

Note: From Pro/ENGINEER Wildfire 5.0 onward, the sketched datum point is obsolete. Its functions are consolidated within the sketched feature as a geometry point, and the geometry entities within the sketched feature generate the corresponding datum entities.

New Pattern Types

Pro/TOOLKIT now supports pattern types such as fill, axis, direction, curve, and point. A new element tree for the creation of the new and existing types of pattern features has been added in *ProPattern.h*. The old element tree is also available in the same header file. Patterns created using the old tree are represented in the new tree format. However, it is strongly recommended that you use the new element tree for all pattern creations.

No-Resolve Mode

Pro/ENGINEER Wildfire 5.0 introduces the No-Resolve mode, wherein if a model and feature regeneration fails, you can continue with the regeneration and select the option to create failed features and children of failed features. Two new bit flags `PRO_REGEN_NO_RESOLVE_MODE` and `PRO_REGEN_RESOLVE_MODE` have been added for the No-Resolve and Resolve mode, respectively, in *ProSolid.h*. Use the appropriate bit flag in the relevant functions as per the mode in which you may want to work. If no bit flag is explicitly set, then your Pro/TOOLKIT application operates in the mode set as per the value of the configuration option `regen_failure_handling` in the Pro/ENGINEER session. If the

configuration option is also not explicitly set, then by default, Pro/ENGINEER and your Pro/TOOLKIT application operate in No-Resolve mode. Existing Pro/TOOLKIT functions for feature inquiry, manipulation, and regeneration will support the failed features and children of failed features created in the No-Resolve mode.

If you want to continue with the Pro/ENGINEER Wildfire 4.0 behavior in Resolve mode, set the configuration option `regen_failure_handling` to `resolve_mode`. Setting the configuration option to switch to Resolve mode ensures the old behavior as long as you do not retrieve the models saved under the No-Resolve mode. To consistently preserve the old behavior, use Resolve mode from the beginning and throughout your Pro/ENGINEER session. Temporarily setting the bit flag `PRO_REGEN_RESOLVE_MODE` in the relevant functions does not ensure the old behavior.

Obsolete HP-GL Pen Plotters

Since the Versatec family of HP-GL pen plotters has been obsoleted, the *ProBoolean* option cutter from the `ProPrintPrinterOptions` structure in *ProPrint.h* has been removed.

System Parameters for Geometric Tolerance Attributes

System parameters can now be automatically generated for geometric tolerance attributes upon creation of the geometric tolerance. These parameters are accessible via Pro/TOOLKIT parameter functions, such as **ProParameter*Get()** and **ProParameter*Set()**, and the UDF placement functions related to variable parameters, such as **ProUdfvarparam*()** and **ProUdfdataVarparamAdd()**.

Ready-to-use Pro/TOOLKIT Application Using Visual Studio

Starting with Pro/ENGINEER Wildfire 5.0, PTC provides a ready-to-use Visual Studio solution file under the Pro/TOOLKIT load point on the Windows platform. With this solution, you can build and test Pro/TOOLKIT applications using the sample makefiles. Refer to the “Fundamentals” chapter or the *Getting Started Guide for Pro/TOOLKIT Wildfire 5.0* for more information.

New Functions

The following section describes the new Pro/TOOLKIT functions.

2D Export

New Function	Description
ProPDFWithProfileExport()	Exports a file to PDF format using an XML profile with an extension .dop (Dex Out Profile).

3D Export

New Function	Description
ProOutputInclusionWithOptionsSet()	Sets the parameters and configuration flags used while exporting the model to a faceted format such as PRO_INTF_EXPORT_CATIA_CGR.
ProIntfSliceFileWithOptionsExport()	Exports to "tesellated" formats such as STL, Render, Inventor, CatiaFacets, and Optegra Visualizer based on the values of a <i>ProArray</i> of parameters and two configuration flags.

3D Import

New Function	Description
ProIntfimportModelWithProfileCreate()	Imports objects of other formats using an XML profile with an extension <code>.dip</code> (Dex In Profile).

Animation

New Function	Description
ProAnimmovieSpinflagSet()	Sets the <i>ProBoolean</i> flag that allows spin in a batch animation process.

Annotations

New Function	Description
Arrow Type	
ProDimlocationArrowtypesGet() ProDtlattachArrowtypeGet()	Accesses the arrow types used by 3D annotations for dimensions, symbols, and surface finish.
Note	
ProNoteURLWstringGet() ProNoteURLWstringSet()	Returns and sets the Uniform Resource Locator (URL) associated with the specified note as a widestring.

Annotation Planes

New Function	Description
ProAnnotationplaneForcetoplaneflagSet() ProAnnotationplaneForcetoplaneflagGet()	Sets the <i>ProBoolean</i> argument <i>force_to_plane</i> for an annotation plane.

Cross Sections

New Function	Description
ProXsecGeometryCollect()	Provides read access to the geometry of cross-section components.
ProXsecGeometryArrayFree() ProXsecGeometryFree()	Clears the memory assigned to the geometry data of cross sections.

Data Sharing Features

New Function	Description
Saved Query	
ProDatasharingfeatureIsQuerydriven()	Identifies and updates a saved, query-driven Data Sharing feature.

Designated Area Features

New Function	Description
Designated Area Property	
ProDesignatedareaColorGet() ProDesignatedareaColorSet() ProDesignatedareaLinestyleGet()	Accesses the property of a Designated Area feature and its status.

New Function	Description
ProDesignatedareaLinestyleSet() ProDesignatedareaStatusGet()	Accesses the property of a Designated Area feature and its status.

Dimensions

New Function	Description
Value Displayed	
ProDimensionOverridevalueGet() ProDimensionOverridevalueSet() ProDimensionValuedisplayGet() ProDimensionValuedisplaySet()	Accesses the display for the dimension value and the overridden value of the dimension.
Rounded Values	
ProDimensionDisplayedValueGet() ProDimensionDisplayedToleranceGet()	Accesses the rounded values displayed for the specified dimension.
ProDimensionDisplayRoundedValueSet() ProDimensionIsDisplayRoundedValue()	Sets the attribute for the specified dimension to display rounded values.
Tolerance Property	
ProDimensionTolerancedecimalsGet() ProDimensionTolerancedecimalsSet() ProDimensionTolerancedenominatorGet() ProDimensionTolerancedenominatorSet()	Accesses the decimal places and denominator value for the dimension tolerance.

Drawings

New Function	Description
ProDrawingMerge()	Merges two drawings.
Setup Option	
ProDrawingSetupOptionGet() ProDrawingSetupOptionSet()	Accesses the option for the drawing setup file.
Context	

New Function	Description
ProRibbonContextGet() ProRibbonContextSet()	Accesses the current context for the specified drawing.
View	
ProDrawingViewTypeGet()	Returns the type of a specified drawing view.
ProDrawingViewParentGet()	Returns the parent view of a specified drawing view.
ProDrawingViewChildrenGet()	Returns the child views of a drawing view and the total number of children.
ProDrawingViewOriginGet() ProDrawingViewOriginSet()	Accesses the location of the origin of a specified drawing view.
ProDrawingViewAlignmentGet() ProDrawingViewAlignmentSet()	Accesses the alignment of drawing view with respect to another view.
ProDrawingViewPerspectiveScaleGet()	Returns the perspective scale applied to a drawing view.
ProDrawingViewFlagGet() ProDrawingViewFlagSet()	Accesses the projection arrow flag for a projected or detailed drawing view.
ProDrawingViewColorSourceGet()	Returns the color source of a drawing view representation.
ProDrawingViewIsErased()	Identifies whether the drawing view is erased.
ProDrawingViewErase()	Erases a specified drawing view.
ProDrawingViewResume()	Resumes the specified drawing view.
ProDrawingViewProjectionSet()	Assigns the specified drawing view as a projection.
ProDrawingViewNameSet()	Assigns a name to a specified drawing view.

New Function	Description
ProDrawingViewDetailCreate() ProDrawingViewDetailReferenceGet() ProDrawingViewDetailReferenceSet() ProDrawingViewDetailCurvedataGet() ProDrawingViewDetailCurvedataSet() ProDrawingViewDetailBoundaryGet() ProDrawingViewDetailBoundarySet()	Creates and accesses the properties of a detailed drawing view.
ProDrawingViewAuxiliaryCreate() ProDrawingViewAuxiliarySet() ProDrawingViewAuxiliaryInfoGet()	Creates and accesses the properties of an auxiliary drawing view.
ProDrawingViewRevolveCreate() ProDrawingViewRevolveInfoGet()	Creates and accesses the properties of a revolved drawing view.
ProDrawingViewOrientationFromNameSet() ProDrawingViewOrientationFromReferenceSet() ProDrawingViewOrientationFromAngleSet() ProDrawingViewVisibleareaTypeGet() ProDrawingViewFullVisibleAreaSet() ProDrawingViewHalfVisibleAreaGet() ProDrawingViewHalfVisibleAreaSet() ProDrawingViewPartialVisibleAreaSet() ProDrawingViewPartialVisibleAreaGet() ProDrawingViewBrokenVisibleAreaGet()	Accesses the types of orientation for a specified drawing view.
ProDrawingViewBrokenVisibleAreaSet() ProDrawingViewBrokenNumberGet()	Accesses the types of visible area for a specified drawing view.
ProDrawingViewSectionTypeGet() ProDrawingView2DsectionGet() ProDrawingView2DsectionSet() ProDrawingView2DsectionNumberGet() ProDrawingView3DsectionGet() ProDrawingView3DsectionSet() ProDrawingViewSinglepartSectionGet() ProDrawingViewSinglepartSectionSet()	Accesses the section types for a specified drawing view.
ProDrawingViewSimplifiedGet() ProDrawingViewSimplifiedSet()	Accesses the simplified representation for a specified drawing view.

New Function	Description
ProDrawingViewExplodedGet() ProDrawingViewExplodedSet()	Accesses the exploded state for a specified drawing view.
Table	
ProDwgtableColumnWidthSet()	Assigns the width of a specified column in a drawing table.
ProDwgtableRowHeightSet()	Assigns the height of a specified row in a drawing table.
ProDwgtableRowDeletePreAction() ProDwgtableRowDeletePostAction()	Provides notification before or after a row is deleted from a given table. The application can cancel the deletion action from a Pre-callback.
ProDwgtableCelltextWrap()	Wraps the text in a drawing table cell.
ProDwgtableSave()	Saves the specified drawing table to disk in either TBL, TXT, or CSV format.
ProDwgtableSegmentOriginSet()	Assigns the origin for a specified segment of a drawing table.
Tree	
ProDrawingtreeRefresh() ProDrawingtreeExpand() ProDrawingtreeCollapse()	Rebuilds, expands, and collapses the drawing tree in the Pro/ENGINEER window that contains the drawing.
Sheet	
ProDrawingSheetCopy()	Creates a copy of the specified sheet in the given drawing.
ProDrawingSheetNameGet()	Retrieves the name of the specified drawing sheet.
Symbol Instance	

New Function	Description
ProDtlsymInstnoteDataGet()	Retrieves the data of a note in the symbol instance.
ProDtlsymInstentityDataGet()	Retrieves the data of an entity in the symbol instance.

Family Tables

New Function	Description
ProFaminstanceFamtableItemIsDefault()	Determines if the value of the specified item for the specified instance is the default value. The default value is the value of the specified item in the generic model.

Feature

New Function	Description
ProFeatureWithoptionsCreate() ProFeatureWithoptionsDelete() ProFeatureWithoptionsSuppress() ProFeatureWithoptionsResume() ProFeatureWithoptionsRedefine() ProFeatureWithoptionsReorder() ProFeatureInsertmodeWithoptionsActivate() ProInsertmodeWithoptionsCancel()	Creates, deletes, or manipulates the specified features in a solid. This action is based on the bitmask containing one or more regeneration control bit flags of the type PRO_REGEN_* defined in <i>ProSolid.h</i> .
ProFeatureStatusflagsGet()	Retrieves the bitmask containing one or more status PRO_FEAT_STAT_* bit flags for a specified feature.

Geometry: Quilts

New Function	Description
ProQuiltVolumeEval()	Calculates the volume of a closed quilt.
ProQuiltIsBackupgeometry()	Identifies if the specified quilt belongs to the invisible Copy Geometry backup feature.

Import Feature

New Function	Description
ProImportfeatWithProfileCreate()	Creates an import feature in the Pro/ENGINEER solid model using an XML profile with an extension .dip (Dex In Profile).

Layers

New Function	Description
ProLayerItemsPopulate()	Allocates and fills an array of <i>ProLayerItem</i> objects that contain the type and identifier of the items assigned to the specified layer.
ProLayeritemarrayFree()	Clears the memory assigned to the <i>ProLayerItem</i> objects.

Manufacturing Process Manager

New Function	Description
ProMfgprocitemFilterAction() ProMfgprocitemVisitAction() ProMfgProctableVisit() ProMfgprocitemFeatureGet() ProMfgprocitemAnnotationGet() ProMfgprocitemCreate() ProMfgprocitemFromTemplateCreate() ProMfgprocitemRedefine() ProMfgprocitemElemtreeGet() ProMfgprocitemElemtreeFree() ProMfgprocitemSubtypeGet() ProMfgprocitemDefaultfixturesetupstepGet() ProMfgprocitemDelete() ProMfgprocitemReorderlimitsGet() ProMfgprocitemReorder() ProMfgprocitemNextitemGet() ProMfgprocitemPreviousitemGet() ProMfgproctableFeaturesCreate() ProMfgproctableFeaturesDelete() ProMfgProctableWrite() ProMfgProctableSynchronize() ProMfgprocitemHolesetdepthGet() ProMfgprocitemHolesetdepthSet() ProMfgprocitemHolesetdepthtypeGet() ProMfgprocitemTemplateparamGet() ProMfgprocitemPropertyGet() ProMfgprocitemPropertySet() ProMfgproctableDisplayUpdate() ProMfgproctableSelecteditemsGet() ProMfgproctableSelecteditemsSet() ProMfgproctableMenuAdd() ProMfgproctableItemAccessFunction() ProMfgproctableItemActionFunction() ProMfgproctablePushbuttonAdd() ProMfgProctableEnable()	Accesses the contents of the Manufacturing Process Manager, its user interface, and its events.

Summary of
Technical Changes

New Function	Description
ProMfgProctableDisable() ProMfgProctablemodelGet() ProMfgproctableExtractionPostAction()	Accesses the contents of the Manufacturing Process Manager, its user interface, and its events.

Mechanica Items

New Function	Description
Pin, Ball, and Planar Constraints	
ProMechdisplacementregularconstrTypeSet() ProMechdisplacementregularconstrTypeGet() ProMechdisplacementregularconstrValueSet() ProMechdisplacementregularconstrValueGet() ProMechdisplacementregularconstrAlloc() ProMechdisplacementregularconstrFree() ProMechdisplacementregularconstrProarrayFree() ProMechdisplacementdatapinconstrSet() ProMechdisplacementdatapinconstrGet() ProMechdisplacementpinconstrAlloc() ProMechdisplacementpinconstrFree() ProMechdisplacementdataPinangularconstrTypeSet() ProMechdisplacementdataPinangularconstrTypeGet() ProMechdisplacementdataPinaxialconstrTypeSet() ProMechdisplacementdataPinaxialconstrTypeGet()	Provides read and write access to the pin, ball, and planar constraints.
Angular Translation	
ProMechdisplacementdataTranslationinterpretinradiansflagGet() ProMechdisplacementdataTranslationinterpretinradiansflagSet()	Reads and sets the flag for the angular translations to be interpreted in radians for a non-Cartesian coordinate system.
Interface	

New Function	Description
ProMechcontactinterfacedataAnglebetweensurfacesExprGet() ProMechcontactinterfacedataAnglebetweensurfacesExprSet()	Provides access to the angle between surfaces for selection filter tolerance while creating runtime contacts between surfaces in a component-component type interface. The angle is specified or retrieved as an expression.
ProMechcontactinterfacedataSeparationdistanceExprGet() ProMechcontactinterfacedataSeparationdistanceExprSet()	Provides access to the separation distance for selection filter tolerance for creating contacts between surfaces in a component-component type interface. The separation distance is specified or retrieved as an expression.
Shell Pair	
ProMechshlpairrefsExtendAdjacentSurfacesSet() ProMechshlpairrefsExtendAdjacentSurfacesGet()	Provides read and write access to the Extend Adjacent Surfaces flag of the shell pair.
AutoGEM Mesh Control	
ProMechmcautogemHardpointSet() ProMechmcautogemHardcurveSet()	Controls the creation of AutoGEM mesh control elements by adding hard points and hard curves.
Beam	
ProMechbeamdataCompcrvrelGet() ProMechbeamdataCompcrvrelSet()	Provides access to the beam release application point for composite curves.
Total Bearing Load At Point (TBLAP)	
ProMechforcedataAxialForceIgnoreflagGet() ProMechforcedataAxialForceIgnoreflagSet() ProMechforcedataAxialMomentIgnoreflagGet() ProMechforcedataAxialMomentIgnoreflagSet()	Reads and sets the flag for ignoring the axial force and axial moment loads in case of Total Bearing Load At Point (TBLAP).
Perimeter Welds	

New Function	Description
ProMechweldedgeMaterialidGet() ProMechweldedgeMaterialidSet()	Provides read and write access to the type of material used in shell elements created on the Perimeter Weld surfaces.
End Welds	
ProMechweldEnddataGet() ProMechweldEnddataSet() ProMechweldendAlloc() ProMechweldendTypeGet() ProMechweldendTypeSet() ProMechweldendExtendAdjacentSurfacesGet() ProMechweldendExtendAdjacentSurfacesSet() ProMechweldendFree()	Provides the read and write access to the end welds defined between different surfaces.
Weld Feature	
ProMechweldfeatureAlloc() ProMechweldFeaturedataGet() ProMechweldfeatureOverrideflagGet() ProMechweldfeatureOverrideflagSet() ProMechweldfeatureFree() ProMechweldfeatureThicknessGet() ProMechweldfeatureThicknessSet() ProMechweldfeatureMaterialidGet() ProMechweldfeatureMaterialidSet() ProMechweldFeaturedataSet()	Provides read and write access to the Weld Feature.

Mechanism Design: Kinematic Drag and Snapshots

New Function	Description
Kinematic Drag	
ProKinDragStart() ProKinDragStop()	Starts and stops a Drag session in the Pro/ENGINEER user interface.
ProKinDragEnvironmentSet()	Sets the environment for subsequent dragging and reconnecting operations.

New Function	Description
ProKinDragReconnect()	Reconnects to the previous Drag session.
ProKinDragSessionInquire()	Identifies if the Drag session is active.
ProKinDragPerformMove()	Performs the Drag operation on a specified geometric object.
Snapshots	
ProKinDragSnapshotsNamesGet() ProSnapshotApply() ProSnapshotCreate() ProSnapshotUpdate() ProSnapshotRename() ProSnapshotDelete() ProSnapshotTrfsGet() ProSnapshotTrfsUpdate() ProSnapshotAllowedInDrawingSet() ProSnapshotAllowedInDrawingGet() ProSnapshotConstraintsGet() ProSnapshotConstraintAdd() ProSnapshotConstraintDelete() ProSnapshotConstraintUpdate() ProSnapshotConstraintEvaluate()	Accesses the snapshots belonging to the main assembly and its subassemblies.

Model

New Function	Description
ProMdlActiveGet()	Retrieves the model handle <i>ProMdl</i> for an active Pro/ENGINEER object.

Parameter

New Function	Description
Parameter Table	
ProParameterTableExport()	Exports a version of the Pro/ENGINEER parameter table to disk in either TXT or CSV format.

Patterns

New Function	Description
ProPatternMembersGet()	Returns the member features in a pattern.
ProPatternMemberVisit()	Visits the features in a pattern.

Sections

New Function	Description
Entity	
ProSectionEntityGetSelected()	Provides the selection references of the selected section entity.
ProSectionEntityIsConstruction() ProSectionEntityConstructionSet()	Identifies a construction entity and sets its property.
ProSectionEntityIntersectionGet()	Provides the intersection points between the two entities of a section.

New Function	Description
ProSectionEntityParamEval()	Provides the corresponding normalized parameter of a valid section entity, given the XY point.
ProSectionEntityCorner() ProSectionEntityDivide()	Allows you to trim, extend, and divide a section entity.
ProSectionEntityUseOffset() ProSectionEntityUseOffsetChain() ProSectionEntityUseOffsetLoop()	Creates offset section entities from edges and 3D curve segment from normal, chain and loop selection.
Dimension	
ProSecdimStrengthen()	Converts a weak dimension to a strong dimension.
ProSecdimLockSet()	Enables you to lock or unlock a given section dimension.
ProSecdimIsLocked()	Identifies if a given section dimension is locked.
Constraints	
ProSectionConstraintCreate() ProSectionConstraintDelete()	Enables the Pro/TOOLKIT application to create and delete section constraints.
ProSectionIntentManagerModeGet() ProSectionIntentManagerModeSet()	Identifies and sets the Intent Manager property of the specified section to ON or OFF.
ProSectionActiveGet() ProSectionActiveSet()	Creates a copy of a section in the same Sketcher session and sets a section as the current active section.

Simplified Representations

New Function	Description
ProBoundaryBoxSimpRepRetrieve() ProDefaultEnvelopeSimpRepRetrieve()	Retrieves into session the boundary box and default envelope representations of a assembly.
ProAsmcompSubstitutionTypeGet()	Returns the substitution type performed on the simplified representation of an assembly component.
ProAsmcompSubstituteGet()	Returns the path to the substituted component representation in the form of <i>ProAsmcompPath</i> and the handle to the substituted component representation in the form of <i>ProAsmcomp</i> .

Solid

New Function	Description
Regeneration	
ProSolidRegenerationStatusGet()	Identifies the regeneration status of the model (equivalent to the Green/Yellow/Red status shown in the Pro/ENGINEER user interface).
ProSolidRegenerationIsNoResolveMode()	Identifies whether the regeneration mode in the active Pro/ENGINEER session is No-Resolve mode.
Failed Features	

New Function	Description
ProSolidFailedfeaturesList()	Retrieves the list of identifiers of failed and/or child of failed and/or child of external failed features in the specified part or assembly.
Feature Status	
ProSolidFeatstatusflagsGet()	Retrieves the array of feature IDs in the specified solid and the corresponding array of bitmasks. The bitmasks represent one or more feature status bit flags of the type <code>PRO_FEAT_STAT_*</code> defined in <i>ProFeature.h</i> .
ProSolidFeatstatusWithoptionsSet()	Assigns the regeneration order and status bit flags for the specified features in a solid. This action is based on the bitmask containing one or more regeneration control bit flags of the type <code>PRO_REGEN_*</code> defined in <i>ProSolid.h</i> .
Mathematical Expression	
ProMathExpressionEvaluate()	Evaluates the given mathematical expression in the context of a given solid. It evaluates expressions of the type <code>ProMechExpression</code> returned by most of the ProMech*Get() functions.

Quick Print

New Function	Description
Regeneration Status	
ProQuickprintoptionsPrintFlatToScreenAnnotsSet()	Sets the <i>ProBoolean</i> flag to quick print flat-to-screen annotations.

User Interface: Dialog Boxes

New Function	Description
ProFileOpenRegister() ProFileOpenAccessFunction() ProFileOpenOperationAction()	Registers a new file type in the Open dialog box in the Pro/ENGINEER user interface.
ProFileSaveRegister() ProFileSaveAccessFunction() ProFileSaveOperationAction()	Registers a new file type in the Save a Copy dialog box in the Pro/ENGINEER user interface.
ProUIDialogFocusSet()	Sets the focus for the specified component in a dialog box created by a Pro/TOOLKIT application.
ProUITableSelectedrownamesGet() ProUITableSelectedrownamesSet()	Reads and sets the selected rows in a table in a dialog box.
ProUIDialogTitlebarimageSet()	Sets the icon to be displayed in the title bar of a dialog box created by a Pro/TOOLKIT application. This icon is also displayed in the task switch bar invoked using Alt+tab.

User Interface: Navigation Area

New Function	Description
ProNavigatorpaneBrowserAdd() ProNavigatorpaneBrowsericonSet() ProNavigatorpaneBrowserURLSet() ProNavigatorpanePHolderAdd() ProNavigatorpanePHolderDevicenameGet() ProNavigatorpanePHolderLayoutGet() ProNavigatorpanePHolderComponentnameGet()	Enables Pro/TOOLKIT applications to add custom panes to the navigation area in the Pro/ENGINEER user interface. A custom pane can contain either custom dialog box components or Web pages.

Summary of
Technical Changes

User Interface: Main Window

New Function	Description
ProWindowVacatePreAction()	Provides notification before a Pro/ENGINEER window is closed or cleared. This function is available by calling ProNotificationSet() with the value of the notify type as PRO_WINDOW_VACATE_PRE.
ProWindowOccupyPostAction()	Provides notification after a model has been loaded into the Pro/ENGINEER window. This function is available by calling ProNotificationSet() with the value of the notify type as PRO_WINDOW_OCCUPY_POST.

User Interface: Accessory Window

New Function	Description
ProAccessorywindowWithTreeCreate()	Opens an accessory window containing the specified object. This function controls the display of the Model Tree for the specified object in the accessory window.

User Interface: Properties of Layout

New Function	Description
ProUILayoutIsMapped()	Specifies if the given layout component is mapped.
ProUILayoutMappedSet()	Keeps the size of the layout unchanged when working with the functions ProUILayoutShow() and ProUILayoutHide() .
ProUILayoutMappedUnset()	Sets the layout to its nominal size when working with the functions ProUILayoutShow() and ProUILayoutHide() .

Windchill Server Connectivity

New Function	Description
Server Operations	
ProServerObjectIsModified()	Verifies the current status of the object in the workspace as well as in the local workspace cache.

Superseded Functions

The following table lists superseded functions in this release.

Superseded Function	New Function
3D Export	
ProOutputInclusionFacetparamsSet() ProIntfSliceFileExport()	ProOutputInclusionWithOptionsSet() ProIntfSliceFileWithOptionsExport()
Cross Section	
ProXsecGeometryRetrieve()	ProXsecGeometryCollect()
Drawing Context	
ProDrawingContextGet() ProDrawingContextSet()	ProRibbonContextGet() ProRibbonContextSet()
Displacement Constraint	
ProMechdisplacementconstrAlloc() ProMechdisplacementconstrTypeGet() ProMechdisplacementconstrTypeSet() ProMechdisplacementconstrValueGet() ProMechdisplacementconstrValueSet() ProMechdisplacementconstrFree() ProMechdisplacementconstrProarrayFree()	ProMechdisplacementregularconstrAlloc() ProMechdisplacementregularconstrTypeGet() ProMechdisplacementregularconstrTypeSet() ProMechdisplacementregularconstrValueGet()

Superseded Function	New Function
	ProMechdisplacementregularconst rValueSet() ProMechdisplacementregularconst rFree() ProMechdisplacementregularconst rProarrayFree()
Feature	
ProFeatureCreate() ProFeatureDelete() ProFeatureSuppress() ProFeatureResume() ProFeatureRedefine() ProFeatureReorder() ProFeatureInsertModeActivate() ProInsertModeCancel()	ProFeatureWithoptionsCreate() ProFeatureWithoptionsDelete() ProFeatureWithoptionsSuppress() ProFeatureWithoptionsResume() ProFeatureWithoptionsRedefine() ProFeatureWithoptionsReorder() ProFeatureInsertmodeWithoption sActivate() ProInsertmodeWithoptionsCancel()
Interface	
ProMechcontactinterfacedataAnglebetweens urfacesGet() ProMechcontactinterfacedataAnglebetweens urfacesSet()	ProMechcontactinterfacedataAngl ebetweensurfacesExprGet() ProMechcontactinterfacedataAngl ebetweensurfacesExprSet()
ProMechcontactinterfacedataSeparationdista nceGet() ProMechcontactinterfacedataSeparationdista nceSet()	ProMechcontactinterfacedataSepa rationdistanceExprGet() ProMechcontactinterfacedataSepa rationdistanceExprSet()
Layers	
ProLayerItemsGet()	ProLayerItemsPopulate()
Note	
ProNoteURLGet() ProNoteURLSet()	ProNoteURLWstringGet() ProNoteURLWstringSet()
Server Operations	
ProServerObjectStatusGet()	ProServerObjectIsModified()
Solid	
ProSolidFailedFeatsList()	ProSolidFailedfeaturesList()

Miscellaneous Technical Changes

The following changes in Pro/ENGINEER Wildfire 5.0 can affect functional behavior in Pro/TOOLKIT. PTC does not anticipate that these changes cause critical issues with existing Pro/TOOLKIT applications.

2D Export Formats

Pro/TOOLKIT function **Pro2dExport()** now supports the existing 2D export format PRO_CGM_FILE and the following new 2D export formats added to the enumerated type ProImportExportFile in *ProUtil.h*:

Export Format	Type Constant
Stheno	PRO_STHENO_FILE
TIFF	PRO_SNAP_TIFF_FILE

The Stheno format is supported only on 32-bit and 64-bit Windows platforms.

3D Contact Feature Element Tree

Access to the new 3D Contact feature in Mechanism has been provided via the element tree in *ProContact3dFeat.h*.

Improvements to the feature element tree for the 3D Contact feature are planned for the next release of Pro/ENGINEER. Owing to these enhancements, the existing element tree will not be available for use. You will need to rebuild your existing Pro/TOOLKIT applications according to the new element tree.

Annotations and Reference Planes

Annotations can now be placed on the reference plane while defining the annotation orientation. Use the new function **ProAnnotationplaneForcetoplaneflagSet()** to set the *ProBoolean* argument *force_to_plane* to PRO_B_TRUE for an annotation plane. If this argument is set to PRO_B_TRUE, then the annotations that reference the annotation plane are placed on that plane. If the annotation orientation is not frozen and if the reference plane is moved, then the annotations will also move along with the plane.

Belt Feature Element Tree

Access to the new Belt feature in Mechanism has been provided via the element tree in *ProBeltFeat.h*.

Color Schemes

Pro/ENGINEER Wildfire 5.0 introduces a new default color scheme with a light to dark blue gradient background. This color scheme is represented by an existing enumerated type `PRO_COLORMAP_ALT_DEFAULT` under `ProColormapAlternatescheme` in *ProColor.h*. The previous default color scheme with a light to dark grey background is now represented by an enumerated type `PRO_COLORMAP_CLASSIC_WF`. Two optional color schemes with dark and medium backgrounds have been added. These optional color schemes are represented by the enumerated types `PRO_COLORMAP_OPTIONAL1` and `PRO_COLORMAP_OPTIONAL2`, respectively.

Cross Sections

Pro/TOOLKIT provides improved capabilities to read the properties of the geometry of cross-section components. The data structure `ProXsecGeom` in *ProXsec.h* has been deprecated. The function **`ProXsecGeometryRetrieve()`** accessing the data structure has also been deprecated. Use the new data structure `ProXsecGeometry` and new functions that access this data structure instead.

The function **`ProXsecTypeGet()`** now supports the cutting object type `PRO_XSEC_PATTERN` along with the existing types `PRO_XSEC_PLANAR` and `PRO_XSEC_OFFSET`.

Datum Coordinate System Element Tree

The element `PRO_E_CSYS_FOLLOW_SRF_OPT` to specify whether the datum coordinate system follows a sheetmetal surface has been added in the element tree for the Datum Coordinate System feature in *ProDtmCsys.h*. This element will be available only in the sheetmetal mode.

Datum Target Annotation Features

Pro/ENGINEER Wildfire 5.0 now supports the creation of a type of Annotation Feature called Datum Target Annotation Feature (DTAF) to define datum targets and associated set datums. A type of Annotation Element called Datum Target Annotation Element (DTAE) can also be created only in DTAFs to specify datum target geometry. A DTAF can contain only two types of Annotation Elements, only one Set Datum Target Annotation Element (SDTAE) and one or more DTAEs. However, Pro/TOOLKIT does not support the creation of DTAFs and DTAEs. The following table lists the functions that return new error types while accessing DTAFs and DTAEs:

Pro/TOOLKIT Function	New Error Type
ProAnnotationelemAnnotationSet()	PRO_TK_CANT_MODIFY
ProAnnotationfeatElementAdd()	PRO_TK_CANT_MODIFY
ProAnnotationfeatElementArrayAdd()	PRO_TK_CANT_MODIFY
ProAnnotationfeatElementDelete()	PRO_TK_NO_PERMISSION

Designated Area Element Tree

Access to the new Designated Area feature is provided via the element tree in *ProDesignatedArea.h*. Use this tree to create, access, and redefine designated area features.

Dimensions in Draft Groups

Dimensions visible on drawings can now be included in draft groups. Various types of dimensions, such as linear, angular, radial, diameter, chamfer, and coordinate can be added to draft groups in a similar manner as notes. All Pro/TOOLKIT **ProDtlgroup*** functions now accept and return dimensions in draft groups. However, Pro/Develop functions such as **prodtl_get_item()**, **prodtl_create()** and **prodtl_modify()** will not support dimensions in draft groups.

Enhancements to ProModelitemNameGet() and Set()

In addition to notes of the type PRO_NOTE, the functions **ProModelitemNameGet()** and **ProModelitemNameSet()** can now be used to read and set the name of the following annotation types:

- Driving or driven dimension of the type PRO_DIMENSION
- Reference dimension of the type PRO_REF_DIMENSION
- Symbol instance of the type PRO_SYMBOL_INSTANCE
- Surface finish of the type PRO_SURF_FIN
- Geometric tolerance of the type PRO_GTOL
- Set datum tag of the type PRO_SET_DATUM_TAG (applicable only for **ProModelitemNameGet()**)

Hole Element Tree

The element PRO_E_HLE_MAKE_LIGHTWT has been added in the element tree for the Hole feature in *ProHole.h*. This element helps you identify the type of hole to be created, whether it will be a regular hole feature or a lightweight hole.

Inheritance Feature Support

Several new Get and existing Set functions now support the reading and modification of the properties of Inheritance feature subitems. Refer to the following table and to the “Assembly: Data Sharing Features” chapter in the Pro/TOOLKIT User's Guide for more information.

Reading of Properties	Modification of Properties
ProDimensionTolerancedecimalGet() ProDimensionTolerancedenominatorGet() ProDimensionOverridevalueGet() ProDimensionValuedisplayGet()	ProGtolPrefixSet() ProGtolSuffixSet() ProGtolTopTextSet() ProGtolRightTextSet() ProGtolTextstyleSet() ProGtoltextTextstyleSet() ProNoteTextSet() ProNoteURLSet() ProNoteElbowlengthSet() ProNoteTextstyleSet() ProTextStyle*Set()

Intent Datums

You can now use Intent Datums such as Intent Axis, Intent Point, Intent Plane, and Intent Coordinate System as follows:

- To specify generic direction in features
- In assembly component placement and constraints
- As references for sketcher dimensions

You can use Intent Planes as sketch orientation or placement references, and you can use Intent Point and Intent Axis to create sections using projections.

The following table lists the Pro/ENGINEER features and the specific intent datums that they can use:

Pro/ENGINEER Feature	Intent Datum	Purpose
Hole feature	Intent Point, Intent Axis, and Intent Plane	For hole placement in parts
Extrude feature	Intent Point, Intent Axis, and Intent Plane	For depth reference

Pro/ENGINEER Feature	Intent Datum	Purpose
Revolve feature	Intent Axis, Intent Plane, and Intent Point	For placement reference

Issue with Legacy Toolkit Applications

Spawn mode and asynchronous mode applications that include calls to the function **ProMfgCreate()** built in Pro/TOOLKIT for Pro/ENGINEER Wildfire 3.0 and earlier releases fail when run with Pro/ENGINEER Wildfire 4.0 and later. This issue has been identified and fixed in Pro/ENGINEER Wildfire 4.0 release M130 and later and in Pro/ENGINEER Wildfire 5.0 release M020 and later.

Materials and Material Properties

Pro/ENGINEER Wildfire 5.0 introduces an option to create elastoplastic (nonlinear) materials and their properties. The new material sub type `PRO_MATERIAL_SUB_TYPE_ELASTOPLASTIC` and its property types are defined in *ProMaterial.h*.

Mechanica Updates

Datum Reference Features

Geometric items such as datum reference features can now be referenced in the Pro/ENGINEER Mechanica environment. Datum reference features store the design intent objects that are families of associated points, curves, edges, or surfaces. The following new types of datum reference features have been added to *ProMechGeomref.h*:

- `PRO_MECH_INT_PNT`
- `PRO_MECH_INT_CURVE`
- `PRO_MECH_INT_EDGE`
- `PRO_MECH_INT_SURFACE`

Weld Feature

Weld Feature, a new type of weld connection, has been introduced in Pro/ENGINEER Mechanical. This feature is represented by a new enumerated type `PRO_MECH_WELD_FEAT` under `ProMechWeldType` in *ProMechWeld.h* and is used to select Fillet and Groove type of weld connections for inclusion in a midsurface compressed model. The geometric references for this weld connection is represented by a new enumerated type `PRO_MECH_FEAT` under `ProMechGeomrefType` in *ProMechGeomref.h*.

Total Heat Load for Referenced Volume

The heat load type `PRO_MECH_HEAT_TOTAL` can now be used along with the geometric reference type `PRO_MECH_VOLUME` while applying total heat load to a referenced volume. Earlier, only `PRO_MECH_HEAT_PER_UNIT` could be applied even in case of referenced volumes.

Total Bearing Load at Point

A new type of load distribution, Total Bearing Load at Point (TBLAP), has been introduced in Pro/ENGINEER Mechanical. It is represented by a new enumerated type `PRO_MECH_FORCE_TBLAP` under `ProMechForceType` in *ProMechLoad.h* and is used to specify TBLAP for cylindrical surfaces and curves.

New Types for ProFileList()

Starting with Pro/ENGINEER Wildfire 5.0, the function **ProFilesList()** has been enhanced to return instance objects when accessing Windchill workspaces or folders. A PDM location (for workspace or commonspace) must be passed as the directory path. The following new types have been added under the enumerated type `ProFileListOpt` that can be passed as the `listing_option` argument to the function **ProFilesList()**:

- `PRO_FILE_LIST_ALL_INST`—Same as the `LIST_ALL` type. It returns instances only for PDM locations.
- `PRO_FILE_LIST_LATEST_INST`—Same as the `LIST_LATEST` type. It returns instances only for PDM locations.

- **PRO_FILE_LIST_ALL_SORTED_INST**—Same as the **LIST_ALL_SORTED** type. It returns instances only for PDM locations.
- **PRO_FILE_LIST_LATEST_SORTED_INST**—Same as the **LIST_LATEST_SORTED** type. It returns instances only for PDM locations.

NC Sequence Pattern Support

Pro/TOOLKIT now supports creating an NC sequence pattern. A new compound element **PRO_E_PAT_APPLICATIONS** has been added in *ProPattern.h* for the creation of an NC sequence pattern.

Obsolete Data Exchange Formats

The following data exchange formats and the corresponding type constants are no longer supported:

Data Exchange Format	Type Constant
CADAM	PRO_CADAM_FILE PRO_CADAM_DIRECT_FILE PRO_CADAM_CPTR_FILE
PDGS	PRO_INTF_EXPORT_PDGS PRO_INTF_IMPORT_PDGS
CATIA	PRO_INTF_EXPORT_CATIA PRO_INTF_IMPORT_CATIA PRO_INTF_CATIA
IDEAS 10	PRO_INTF_IMPORT_IDEAS_DESIGN PRO_INTF_IMPORT_IDEAS_PACKAGE PRO_INTF_IDEAS_DESIGN PRO_INTF_IDEAS_PACKAGE

PDF/A Format

New export option `PRO_PDFOPT_PDF_SAVE` has been added to the enumerated type `ProPDFOptionType` in *ProPDF.h* to specify the PDF format while exporting 2D drawings and solid models. Set this option to `PRO_PDF_ARCHIVE_1` to export 2D drawings to the PDF/A format instead of the standard PDF format. The `PRO_PDF_ARCHIVE_1` type is applicable only for 2D drawings, that is, when you set the option `PRO_PDFOPT_EXPORT_MODE` to `PRO_PDF_2D_DRAWING`.

PRO_FEAT_UNREGENERATED Status

Starting with Pro/ENGINEER Wildfire 5.0 M050, the functions **ProSolidFeatstatusflagsGet()** and **ProFeatureStatusflagsGet()** return the `PRO_FEAT_UNREGENERATED` status value, in addition to the earlier `PRO_FEAT_ACTIVE` status value, for an unregenerated feature. The functions **ProSolidFeatstatusGet()** and **ProFeatureStatusGet()** return `PRO_FEAT_UNREGENERATED`, instead of the `PRO_FEAT_ACTIVE` status value, for an unregenerated feature.

Punch Form Feature Element Tree

Access to the new Punch Form feature has been provided via the element tree in *ProSmtForm.h*.

Quilt Form Feature Element Tree

Access to the new Quilt Form feature has been provided via the element tree in *ProSmtPunchQuilt.h*.

Revision Field in the Registry File

From Pro/ENGINEER Wildfire 5.0 onward, the revision field in the Pro/TOOLKIT registry file is ignored when running a Pro/TOOLKIT application. Always build your Pro/TOOLKIT applications with a supported build of Pro/ENGINEER for the specified release of Pro/TOOLKIT. For more information on Pro/TOOLKIT compatibility with Pro/ENGINEER, go to <http://www.ptc.com/partners/hardware/current/support.htm> and select a Pro/TOOLKIT option from the **By Product** list.

Round Element Tree

A new element `PRO_E_RNDCH_AUTO_CONTINUE` has been added in the element tree for the Round feature in *ProRound.h* to identify whether surfaces will be extended to meet the designated round surface.

A new option `PRO_ROUND_CURV_CONTINUOUS` to specify curvature continuous rounds has been added to the enumerated type `ProRoundConicType` in *ProRound.h*.

Section Dimensions

New dimension types have been added to the enumerated type `ProSecdimType` in *ProSecdimType.h*.

Set Datums in Inheritance Features

Set datum axes and planes contained as references in an Inheritance feature are now accessible to geometric tolerances. Use the following Pro/TOOLKIT functions to access the set datums in Inheritance features:

- `ProGtoldatumrefAlloc()`
- `ProGtoldatumrefFree()`
- `ProGtoldatumrefGet()`
- `ProGtoldataGtoldatumrefSet()`

Simplified Representations

New simplified representation types have been added to the enumerated types `ProSimprepActionType` and `ProSimprepType` in *ProSimprep.h*.

SolidWorks and Inventor 3D Import Formats

Pro/TOOLKIT functions **`ProIntfimportModelCreate()`** and **`ProIntfimportModelWithProfileCreate()`** now support the following new 3D import formats:

Import Format	Type Constant
SolidWorks Part (.sldprt) file	<code>PRO_INTF_IMPORT_SW_PART</code>

Import Format	Type Constant
SolidWorks Assembly (.sldasm) file	PRO_INTF_IMPORT_SW_ASSEM
Inventor Part (.ipt) file	PRO_INTF_IMPORT_INVENTOR_PART
Inventor Assembly (.iam) file	PRO_INTF_IMPORT_INVENTOR_ASSEM

The 3D import formats listed above are supported only on 32-bit and 64-bit Windows platforms.

Shrinkwrap Feature Element Tree

New elements have been added to the element tree for the Shrinkwrap feature in *ProDataShareFeat.h*.

Spring Feature Element Tree

New elements have been added in the element tree for the Spring feature in *ProDamperFeat.h* to specify the attachment references.

Surface Region Element Tree

Pro/TOOLKIT now provides the capability to create a Surface Region feature programmatically in the Pro/ENGINEER Mechanica environment. Access to the new feature is available via the element tree in *ProSurfReg.h*.

Support for the Detailed Drawing User Interface

The user interface in Detailed Drawings has been restructured. This may affect the behavior of existing Pro/TOOLKIT applications that add commands to specific Pro/ENGINEER menus or toolbars. If these menus or toolbars or both have been redesigned in Pro/ENGINEER Wildfire 5.0, the commands added by the Pro/TOOLKIT application appear under a new **Pro/TOOLKIT** menu in Drawing mode. Refer to the “User Interface: Menus, Commands, and Popupmenus” chapter for more information.

True Type Font Support

The *ProBoolean* option `use_ttf` has been added to the `ProPrintPrinterOptions` structure in *ProPrint.h* for plotting text in True Type Font (TTF). Set this option to `PRO_B_TRUE` to enable your printer to plot TTF text, instead of drawing the text with lines. Existing Pro/TOOLKIT functions such as **ProPrintExecute()**, **ProPrintPCFOptionsGet()**, and **ProPrintPrinterOptionsGet()** support this new option while plotting.

Variable Pull Direction Draft Feature Element Tree

Access to the new Variable Pull Direction Draft feature has been provided via the element tree in *ProVPDD.h*.

Weld Symbol Annotation Elements

Pro/ENGINEER Wildfire 5.0 supports the creation of weld symbols in Weld features as 3D Symbol Annotation Elements. Pro/TOOLKIT allows you to access the Weld Symbol Annotation Elements using existing **ProAnnotation*()** functions.

B

Unicode Encoding

This appendix describes how the new Unicode support used internally by Pro/ENGINEER from Wildfire 4.0 onward affects Pro/TOOLKIT and its applications.

Topic	Page
Introduction to Unicode Encoding	B - 2
Unicode Encoding and Pro/TOOLKIT	B - 3
Necessity of Unicode Compliance	B - 4
Platform Native Encoding and Unicode	B - 4
Legacy Encoding and Unicode	B - 10
Programming Considerations	B - 11
External Interface Handling	B - 15
Support for Legacy Encoding in Pro/TOOLKIT	B - 21

Introduction to Unicode Encoding

UNICODE is an acronym for "Universal Character Encoded System". It is a unique character encoding scheme allowing characters from European, Greek, Arabic, Hebrew, Chinese, Japanese, Korean, Thai, Urdu, Hindi, and other world languages to be encoded in a single character set. This enables applications to simultaneously support text in multiple languages in their data files. Unicode encoding covers most of the letters, punctuation marks, and technical symbols commonly used in the English language that are not covered by the legacy encoding.

Unicode defines two mapping methods:

- UCS (Universal Character Set) encoding
- UTF (Unicode Transformation Format) encoding

For more information on Unicode Encoding, visit <http://unicode.org>.

Pro/ENGINEER Wildfire 4.0 onward, all string data in Pro/ENGINEER (previously stored in the legacy encoding format) is now stored in the Unicode encoding. Pro/ENGINEER Wildfire 4.0 uses the UCS-2 encoding on Windows platforms and UCS-4 encoding in UNIX environments for widestring data. It reads and writes character data using the multibyte UTF-8 encoding on all platforms. UTF-8 is an 8-bit, variable-length character encoding format that uses one to four bytes per character.

Some important terminology about string encoding related to Pro/TOOLKIT that is used throughout this appendix is described as follows:

- “Unicode encoding” refers to the string and widestring encodings used by Pro/ENGINEER Wildfire 4.0 and later.
- “Legacy encoding” refers to the encoding used by Pro/ENGINEER Wildfire 3.0 and earlier. Depending on the language, this encoding is typically some version of an EUC encoding.
- “Native encoding” refers to the encoding used by the operating system in the language in which the system is running. This encoding is the same as legacy encoding in most cases.
- “Multibyte string” refers to a character array representing a string in the C language. Because of the limited size of the character (a single byte), combinations of multiple bytes are used to represent characters outside the ASCII range.

- “7-bit ASCII” refers to the character range 0x0 through 0x127. This range is shared between Unicode and non-Unicode encodings used by Pro/ENGINEER. Thus, any data of this type is unchanged after transcoding.
- “8-bit ASCII” refers to the character range 0x128 through 0x255. In many European native encodings, this range is used to represent European accented vowels and other letters. In Unicode, this range is not directly used. Therefore, 8-bit ASCII native strings are not equivalent in Unicode.
- “Byte Order Mark” (BOM) refers to a string of three bytes U+FEFF (represented in C language strings by “\357\273\277”), and is placed on the top of a text file to indicate that the text is Unicode encoded. Unicode has designated the character U+FEFF as the BOM and reserved U+FFFE as an illegal character for UTF-8 encoding. Most of the text files generated by Pro/ENGINEER are written with the BOM and Unicode encoding. Pro/ENGINEER can accept a Unicode encoded text file with a BOM, or a legacy encoded text file without a BOM as the input.
- “Transcoding” refers to the act of changing a string or wstring encoding from one encoding to another, for example, from platform native to Unicode or vice-versa. For some transcoding operations, there is a possibility of data loss, since characters from one encoding may not be supported in the target encoding.

Unicode Encoding and Pro/TOOLKIT

Pro/TOOLKIT applications running with Pro/ENGINEER Wildfire 4.0 and later must, by default, receive and send strings and wstrings to Pro/ENGINEER in Unicode encoding. This is a change to the encoding previously received by applications in Wildfire 3.0 and earlier. Because the workstation operating system will not be running in Unicode and other languages, functions and libraries accessed by the Pro/TOOLKIT application may not be Unicode aware, the Pro/TOOLKIT application must deal with the change of encoding in one of the following ways:

- Make changes to the application to expect and accept Unicode strings and wstrings when dealing with Pro/ENGINEER data. At the external interfaces from the application to the operating system or third-party APIs, perform necessary transcoding operations to ensure that those other systems receive an expected encoding.

- Turn off the `unicode_encoding` registry file option, so that Pro/ENGINEER can automatically transcode data at the Pro/TOOLKIT interface. While this may result in a short term delay in making changes to the application code, there are restrictions and risk associated with using the legacy encoding mode. Refer to section Support for Legacy Encoding in Pro/TOOLKIT for more information on this mode.

PTC recommends that all applications be evaluated for Unicode compliance regardless of their purpose or intended data. However, applications that would particularly be affected by Unicode encoding are as follows:

- Any Pro/TOOLKIT application expected to work with Pro/ENGINEER in any language other than English.
- Any Pro/TOOLKIT application expecting Pro/ENGINEER data in any language other than English (where strings from that data are transferred to and from Pro/ENGINEER or any other source).

Necessity of Unicode Compliance

It is strongly recommended that you make your existing Pro/TOOLKIT applications Unicode-compliant for the following reasons:

- Applications that are not Unicode-compliant will be unable to reliably handle Pro/ENGINEER data saved in the Unicode format with strings (notes, annotations, table, and so on) in multiple languages other than English. For example, a Pro/ENGINEER drawing can now contain both German and Japanese notes. The Pro/TOOLKIT application will not be able to read or modify those notes correctly without being Unicode compliant. This could result in data loss or corruption.
- Applications that do not consider the Unicode nature of Pro/ENGINEER data may try to pass that data directly to the system or third-party APIs that do not recognize it correctly. This could cause data corruption or crashes.
- Applications that do not transcode non-Unicode data into Unicode before using the data as strings inside Pro/ENGINEER models will generate corrupt and incorrect models.

The following sections describe the functions provided by Pro/TOOLKIT for conversion between platform-specific native or legacy encoding and Unicode encoding.

Platform Native Encoding and Unicode

The functions described in this section support the conversion of platform-specific native-encoded strings and wstrings to the Unicode-encoded format, and vice versa. These transcoding functions have the following common characteristics:

- Input arguments to the functions are the string to be transcoded and an integer indicating the number of characters or wide characters to be transcoded. The integer argument is optional, you can pass `PRO_VALUE_UNUSED` to transcode the entire input string.
- A buffer capable of holding the output string and an integer indicating its size are also the input arguments. The buffer should always have sufficient memory to contain the output transcoded string or wstring, including the NULL-terminated character.
- While transcoding from Unicode to native encoding, the functions allow you to use a character as one of the input arguments to replace any out-of-locale characters encountered. An output flag indicates if this out-of-locale replacement took place.

Unicode Encoding

Functions Introduced:

- **ProNativestringToString()**
- **ProNativestringToWstring()**
- **ProStringToNativestring()**
- **ProStringToNativewstring()**
- **ProNativewstringToString()**
- **ProNativewstringToWstring()**
- **ProWstringToNativewstring()**
- **ProWstringToNativestring()**

The function **ProNativestringToString()** converts a native-encoded multibyte string to a Pro/ENGINEER compatible Unicode-encoded multibyte string. This function is used when a native-encoded multibyte string from an external interface is to be sent to Pro/ENGINEER.

The function **ProNativestringToWstring()** converts a native-encoded multibyte string to a Pro/ENGINEER compatible Unicode-encoded widestring. This function is used when a native-encoded multibyte string from an external interface is to be sent to Pro/ENGINEER as a widestring.

The function **ProStringToNativestring()** converts a Pro/ENGINEER Unicode-encoded multibyte string to a native multibyte string. This function is used when a Unicode-encoded multibyte string obtained from Pro/ENGINEER needs to be sent to an application expecting a native-encoded multibyte string.

The function **ProStringToNativewstring()** converts a Pro/ENGINEER Unicode-encoded multibyte string to a native-encoded widestring. This function is used when a multibyte string obtained from Pro/ENGINEER is to be sent to an application expecting a native-encoded widestring.

The function **ProNativewstringToString()** converts a native-encoded widestring to a Pro/ENGINEER compatible Unicode-encoded multibyte string. This function is used when the output from an external interface returning a native-encoded widestring is sent to be to Pro/ENGINEER as a multibyte string.

The function **ProNativewstringToWstring()** converts a native-encoded widestring to a Pro/ENGINEER compatible Unicode-encoded widestring. This function is used when a native-encoded widestring from an external interface is to be sent to Pro/ENGINEER.

The function **ProWstringToNativestring()** converts a Pro/ENGINEER Unicode-encoded widestring to a native multibyte string. This function is used when a widestring from Pro/ENGINEER is to be sent to an external interface expecting a native-encoded multibyte string.

The function **ProWstringToNativewstring()** converts a Pro/ENGINEER Unicode-encoded widestring to a native-encoded widestring. This function is used when a Unicode-encoded widestring from Pro/ENGINEER needs to be sent to an external interface expecting a native-encoded widestring.

Example 1: Convert a Unicode-encoded string to a native and legacy string

```
#include <ProToolkit.h>
#include <ProMdlChk.h>
#include <ProParameter.h>
#include <ProSolid.h>
```

```

#include <ProDrawing.h>
#include <ProFamtable.h>
#include <ProSelection.h>
#include <ProScope.h>
#include <ProTKRunTime.h>
#include <ProMdl.h>

static ProError status;
static ProFileName message_file;

/*=====*\
FUNCTION : UserUnicodeToNativeAndLegacy()
PURPOSE  : Change unicode string to a native and legacy string.
NOTE     : In order for this function to work load a model in Pro/ENGINEER
           containing a parameter of type PRO_PARAM_STRING having name
           TEST_UNICODE.
\*=====*/
int UserUnicodeToNativeAndLegacy()
{
    ProMdl model ;                /* current model */
    ProModelitem modelitem;
    ProParamvalue param_value;
    ProParameter param;
    ProName mdl_name, param_name;
    char line[500];
    FILE *fptr_native, *fptr_legacy;
    ProBoolean was_used;

    fptr_native = fopen("unicode_to_native.txt", "w");

    fptr_legacy = fopen("unicode_to_legacy.txt", "w");

    ProStringToWstring(message_file, "pt_ug_unicode.txt");

/*-----*\
    Check for a model. Check for the correct parameter type, if incorrect
    display error. Retrieve the parameter value and write to a file.
\*-----*/
    status = ProMdlCurrentGet(&model);

    status = ProMdlNameGet(model, mdl_name);

    status = ProMdlToModelitem(model, &modelitem);

    param_value.type = PRO_PARAM_STRING;

    ProStringToWstring(param_name, "TEST_UNICODE");

    status = ProParameterInit(&modelitem, param_name, &param);

    /* Display error to the user that the model does not contain the

```

```

        string parameter with name TEST_UNICODE. */

    if(status == PRO_TK_E_NOT_FOUND)
    {

status = ProMessageDisplay(message_file, "UG ParamMissingError");
return(0);

    }

    ProParameterValueGet(&param, &param_value);

    /* Display error to the user that the TEST_UNICODE
       parameter is not of type string. */

    if(param_value.type != PRO_PARAM_STRING)
    {
status = ProMessageDisplay (message_file, "UG ParamTypeError");
return(0);
    }

    status = ProWstringToNativestring(line,500, param_value.value.s_val,
                                     PRO_VALUE_UNUSED, '^', &was_used);

fprintf(fp_ptr_native,
        "After Unicode Wide String to Native Conversion, the string is
: %s", line);

    fclose(fp_ptr_native);

    status = ProWstringToLegacystring(line,500,param_value.value.s_val,
                                     PRO_VALUE_UNUSED, '^', &was_used);

    fprintf(fp_ptr_legacy,
        "After Unicode Wide String to Legacy Conversion, the string is
: %s", line);

    fclose(fp_ptr_legacy);

    return(1);
}

```

Example 2: Write a Unicode-encoded widestring to Pro/ENGINEER after reading a native string from a file

```

#include <ProToolkit.h>
#include <ProMdlChk.h>
#include <ProParameter.h>
#include <ProSolid.h>
#include <ProDrawing.h>
#include <ProFamtable.h>

```

```

#include <ProSelection.h>
#include <ProScope.h>
#include <ProTKRunTime.h>
#include <ProMdl.h>

static ProError status;
static ProFileName message_file;

/*=====*\
FUNCTION : UserReadNativeToUnicode()
PURPOSE  : Write Unicode wide string to Pro/ENGINEER after reading a
native string from a file.
NOTE     : This function reads a string from text file
native_param_value.txt
           and creates a PRO_PARAM_STRING type parameter in the model
           with name TEST_NATIVE_PARAM as sets its value to the string
read.
\*=====*/

int UserReadNativeToUnicode()
{
    ProMdl model ;                /* current model */
    ProModelitem modelitem;
    ProParamvalue param_value;
    ProParameter param;
    ProName mdl_name, param_name, wval;
    ProPath text_path;
    char path[PRO_PATH_SIZE], path_file[PRO_PATH_SIZE],
param_value_str[500];
    wchar_t wline[500];
    FILE *fptr_native;

    status = ProToolkitApplTextPathGet(text_path);

    ProWstringToString(path, text_path);

    sprintf(path_file, "%s/text/usascii/%s", path,
"native_param_value.txt");

    fptr_native = fopen(path_file, "r");

    fgets(param_value_str, 500, fptr_native);

    param_value.type = PRO_PARAM_STRING;

    ProStringToWstring(wval, param_value_str);

    status = ProNativewstringToWstring(param_value.value.s_val, 500, wval,
PRO_VALUE_UNUSED);

/*-----*\

```

```

    Check for a model. Create a string type parameter with value set to
    the string read above.
    \*-----*/
    status = ProMdlCurrentGet(&model);

    status = ProMdlNameGet(model, mdl_name);

    status = ProMdlToModelitem(model, &modelitem);

    ProStringToWstring(param_name, "TEST_NATIVE_PARAM");

    status = ProParameterCreate(&modelitem, param_name, &param_value,
    &param);

    return(1);
}

```

Legacy Encoding and Unicode

The functions described in this section support the conversion of legacy encoded strings and wstrings to the Unicode encoding format, and viceversa. These functions have the common characteristics of the transcoding functions described in the previous section.

Functions Introduced:

- **ProLegacystringToString()**
- **ProStringToLegacystring()**
- **ProLegacystringToWstring()**
- **ProWstringToLegacystring()**

The function **ProLegacystringToString()** converts a legacy-encoded multibyte string to the Pro/ENGINEER compatible Unicode-encoded format. This function is used when a multibyte string from a file written by or for any release of Pro/ENGINEER prior to Wildfire 4.0 is to be sent to the current version of Pro/ENGINEER.

The function **ProStringToLegacystring()** converts a Pro/ENGINEER Unicode-encoded multibyte string to the legacy encoding format used by Pro/ENGINEER prior to Wildfire 4.0. This function is used when a string from Pro/ENGINEER is to be read by an earlier version of Pro/ENGINEER or a Pro/TOOLKIT application running in the earlier version of Pro/ENGINEER.

The function **ProLegacystringToWstring()** converts a legacy-encoded multibyte string to a Pro/ENGINEER compatible Unicode-encoded widestring. This function is used when a multibyte string from a file written by or for any release of Pro/ENGINEER prior to Wildfire 4.0 is to be sent as a widestring to the current version of Pro/ENGINEER.

The function **ProWstringToLegacystring()** converts a Pro/ENGINEER Unicode-encoded widestring to a legacy-encoded string used by Pro/ENGINEER prior to Wildfire 4.0. This function is used when a widestring from Pro/ENGINEER is to be read as a multibyte string by an earlier version of Pro/ENGINEER or a Pro/TOOLKIT application running in the earlier version of Pro/ENGINEER.

Programming Considerations

The strategy to follow when considering where and how to transcode strings between native and legacy encoding depends upon the type of application:

- If an application primarily interacts only with Pro/ENGINEER, it would be efficient for the application to have its strings and widestrings in memory in the Unicode encoding. Transcoding takes place only in those rare instances where the application interacts with the operating system or a third-party system. This would imply that the application “runs in Unicode”. This scenario is illustrated graphically in the figure “Application running in Unicode encoding”.

When the application runs in Unicode, the majority of the code logic should be unaffected by the changeover from legacy to Unicode mode. Only external interfaces will require attention to possibly transcode certain strings leaving or entering the application from outside. Recommendations for transcoding at the external interface level is covered in the section “External Interface Handling”.

Note: Most Pro/TOOLKIT sample applications are coded to run in Unicode.

- If an application primarily interacts with a third-party library, has an established non-Unicode compliant Graphical User Interface (GUI), and only peripherally touches Pro/ENGINEER at limited access points, then this application can be said to be “running in native encoding”. This scenario is illustrated graphically in the figure “Application running in native encoding”.

When running in native encoding, the application must transcode any and all strings transferred between the application and Pro/ENGINEER. Typically, this would be done at a level just before or after the string passes the Pro/TOOLKIT API boundary shown in the figure.

For example,

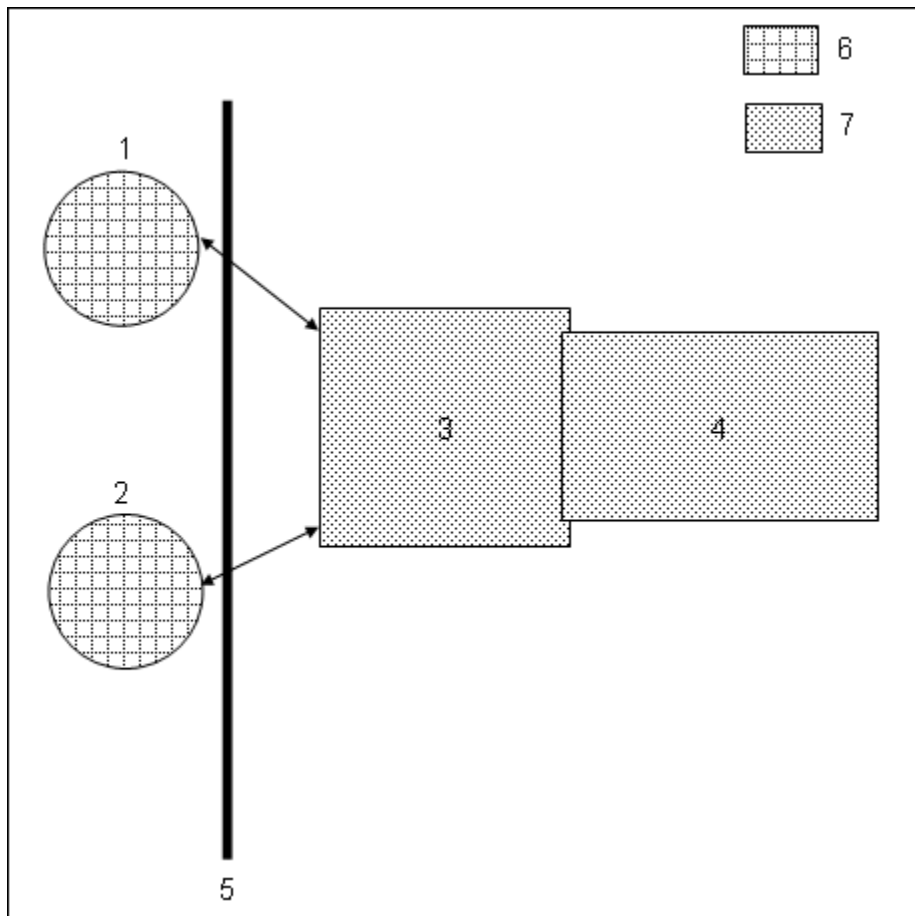
- The function **ProParameterValueGet()** retrieves the string parameter value.
- The function **ProWstringToNativewstring()** converts it to a native wistring.
- Other functions in the application or GUI use only the transcoded wistring.

Or

- The application or GUI generates a multibyte string to be used in a new drawing note.
- The function **ProNativestringToWstring()** transcodes the string to a Unicode wistring (in the releases prior to Wildfire 4.0, a call to the function **ProStringToWstring()** would have been used).
- The function **ProDtlnotetextStringSet()** accepts the Unicode wistring.

Note: It is important to identify not only the functions that directly take strings and wistrings, but also those functions that accept strings as part of a data structure, for example, **ProParameter*** and **ProParamvalue*** functions.

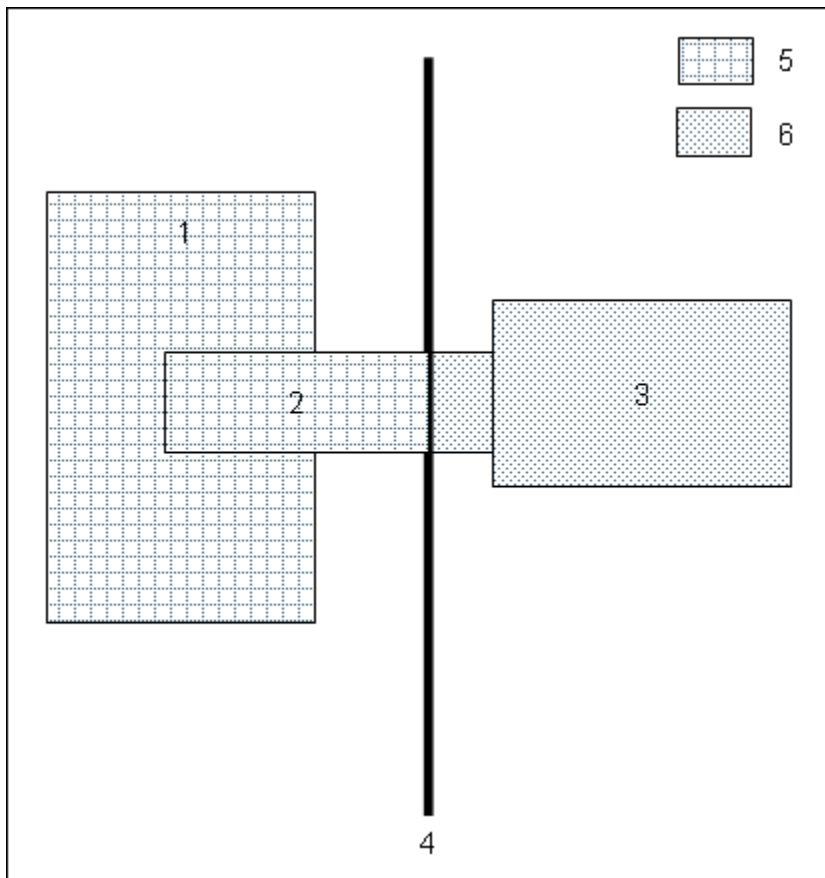
Figure B-1: Application running in Unicode encoding



Unicode Encoding

- 1 External Interface to a third party library
- 2 File containing data to be used by the application
- 3 Pro/TOOLKIT Application
- 4 Pro/ENGINEER
- 5 This boundary indicates where transcoding takes place
- 6 Items with this pattern are operating in the native or legacy encoding
- 7 Items with this pattern are operating in Unicode encoding

Figure B-2: Application running in native encoding



- 1 Graphical User Interface (GUI) used by the application
- 2 Pro/TOOLKIT Application
- 3 Pro/ENGINEER
- 4 This boundary indicates where transcoding takes place
- 5 Items with this pattern are operating in the native encoding
- 6 Items with this pattern are operating in Unicode encoding

External Interface Handling

Pro/TOOLKIT applications running in Unicode will need to create utilities around the interfaces between non-Unicode aware third-party APIs and interfaces. While PTC cannot directly provide such interfaces, this section discusses the considerations for creating such utilities by showing how one external API such as the C runtime library can be used from a Unicode environment.

Any C runtime library accepting `char*` or `wchar_t*` as input may be adversely affected by receiving Unicode data. Typically, it should be possible to create a simple wrapper for each C runtime interface used in the application, where the input string to the interface is expected to be in Unicode. The string should be transcoded before calling the system API. Examples of such C runtime functions are listed below (this is not an exhaustive list):

- `fopen()`
- `access()`, `_access()`
- `chdir()`, `readdir()`, `opendir()`
- `chmod()`, `_chmod()`
- `findfirst()`, `_findnext()`
- `getcwd()`
- `getenv()`
- `open()`, `opendir()`
 - `fgetc()`
 - `fgets()`
 - `fputc()`
 - `fputs()`
 - `fread()`, `fwrite()`
 - `puts()`
- `remove()`, `stat()`, `system()`, `tmpfile()`, `unlink()`

Special External Interface: printf() and scanf() Functions

The **printf()** and **scanf()** family of C runtime functions are a special case of an external interface. The format string must be transcoded when these interfaces are called. The list of variable arguments passed to the functions may also contain string and widestring data that needs to be transcoded and modified in format. Because of the complexity of wrapping these C runtime functions, PTC has provided a standard Pro/TOOLKIT function equivalent for each. These functions support all the format specifiers and modifiers supported by the C language specification.

Functions Introduced:

- **ProTKPrintf()**
- **ProTKFprintf()**
- **ProTKSprintf()**
- **ProTKSnprintf()**
- **ProTKVprintf()**
- **ProTKVfprintf()**
- **ProTKVsprintf()**
- **ProTKVsnprintf()**
- **ProTKScanf()**
- **ProTKFscanf()**
- **ProTKSscanf()**
- **ProTKSnscanf()**
- **ProTKVscanf()**
- **ProTKVfscanf()**
- **ProTKVsscanf()**

The function **ProTKPrintf()** provides the Unicode equivalent to the C runtime function **printf()**. The number of characters returned by this function is sent to stdout. The output data is transcoded to the native encoding format, which may result in out-of-locale characters in the results.

The function **ProTKFprintf()** provides the Unicode equivalent to the C runtime function **fprintf()**. The number of characters returned by this function are copied into the file. This file will receive the data in the Unicode-encoded format.

The functions **ProTKSprintf()** and **ProTKSnprintf()** provide the Unicode equivalent to the C runtime functions **sprintf()** and **snprintf()** respectively. The number of characters returned by these functions are copied into the output buffer.

The function **ProTKSscanf()** provides the Unicode equivalent to the C runtime function **scanf()**. This function parses the contents of the input from stdin. The output data in the string or character format is in Unicode encoding.

The function **ProTKFscanf()** provides the Unicode equivalent to the C runtime function **fscanf()**. This function parses the contents of the input from a file.

The functions **ProTKSscanf()** and **ProTKSnscanf()** provide the Unicode equivalent to the C runtime functions **sscanf()** and **snscanf()** respectively.

The Unicode equivalent of the C runtime functions **v*printf()** and **v*scanf()**, which take a variable arguments list instead of variable number of arguments, have also been provided in the form of **ProTKV*printf()** and **ProTKV*scanf()** functions.

Special External Interface: Windows-runtime Functions

Win32 functions that take `char*` inputs are not Unicode compliant, and thus cannot be used with data directly obtained from Pro/ENGINEER Wildfire 4.0 and later. The simplest approach to using Windows runtime functions is to use the functions accepting `wchar_t*` inputs since these functions are Unicode compliant (Windows native encoding for `whar_t*` is Unicode). For example, use the function **GetMessageW()** instead of **GetMessage()** or **GetMessageA()**.

Special External Interface: Hardcoded Strings

Another example of an external interface is a hardcoded string. You should review all uses of hardcoded strings in your application and ensure that they fit the following categories:

- They use only 7-bit ASCII characters or wide characters.
- They use Unicode escape sequences.

8-bit ASCII or non-Unicode escape sequences in hardcoded strings do not work correctly unless you transcode the string into Unicode before sending it to Pro/ENGINEER.

Example 3: Write a Unicode-encoded widestring from Pro/ENGINEER to a file containing BOM

```
#include <ProToolkit.h>
#include <ProMdlChk.h>
#include <ProParameter.h>
#include <ProSolid.h>
#include <ProDrawing.h>
#include <ProFamtable.h>
#include <ProSelection.h>
#include <ProScope.h>
#include <ProTKRunTime.h>
#include <ProMdl.h>

static ProError status;
static ProFileName message_file;

/*=====*\
FUNCTION : UserUnicodeToUnicode()
PURPOSE  : Write Unicode wide string from Pro/ENGINEER to a file
           containing BOM.
NOTE     : In order for this function to work load a model in Pro/Engineer
           containing a parameter of type PRO_PARAM_STRING having name
           TEST_UNICODE.
\*=====*/
int UserUnicodeToUnicode()
{
    ProMdl model ;           /* current model */
    ProModelitem modelitem;
    ProParamvalue param_value;
    ProParameter param;
    ProName mdl_name, param_name;
    char line[500];
    FILE *fptr_unicode;

    fptr_unicode = PTApplsUnicodeFopen("unicode_to_unicode.txt", "w");

    ProStringToWstring(message_file, "pt_ug_unicode.txt");

/*-----*\
    Check for a model. Check for the correct parameter type, if incorrect
    display error. Retrieve the parameter value and write to a file.
\*-----*/
status = ProMdlCurrentGet(&model);

    status = ProMdlNameGet(model, mdl_name);

    status = ProMdlToModelitem(model, &modelitem);

    param_value.type = PRO_PARAM_STRING;
```



```

ProStringToWstring(param_name, "TEST_UNICODE");

status = ProParameterInit(&modelitem, param_name, &param);

/* Display error to the user that the model does not contain the
   string parameter with name TEST_UNICODE. */

if(status == PRO_TK_E_NOT_FOUND)
{
status = ProMessageDisplay(message_file, "UG ParamMissingError");
return(0);
}

ProParameterValueGet(&param, &param_value);

/* Display error to the user that the TEST_UNICODE
   parameter is not of type string. */

if(param_value.type != PRO_PARAM_STRING)
{
status = ProMessageDisplay (message_file, "UG ParamTypeError");
return(0);
}

ProWstringToString(line,param_value.value.s_val);

status = ProTKFprintf(fp_ptr_unicode, "The Unicode Wide String, is :
%s", line);

fclose(fp_ptr_unicode);
}

```

Example 4: Write Unicode-encoded widestring to Pro/ENGINEER after reading the unicode string from a file containing the BOM

```

/*=====*\
FUNCTION : UserReadUnicodeToUnicode()
PURPOSE  : Write Unicode wide string to Pro/ENGINEER after reading the
           unicode string from a file containing the BOM.
NOTE     : This function reads a unicode string from file
           unicode_param_value.txt and creates a PRO_PARAM_STRING type
           parameter in the model with name TEST_UNICODE_PARAM and sets
           its value to the string read.
           BOM character needs to be removed from the string read from the
file
           before setting the parameter value.
\*=====*/
int UserReadUnicodeToUnicode()
{
    ProMdl model ;                /* current model */
    ProModelitem modelitem;

```

```

ProParamvalue param_value;
ProParameter param;
ProName mdl_name, param_name;
ProPath text_path;
char path[PRO_PATH_SIZE], path_file[PRO_PATH_SIZE],
param_value_str[500];
wchar_t wline[500];
FILE *fptr_unicode;

status = ProToolkitApplTextPathGet(text_path);

ProWstringToString(path, text_path);

status = ProTKSprintf(path_file, "%s/text/usascii/%s", path,
    "unicode_param_value.txt");

fptr_unicode = PTApplsUnicodeFopen(path_file, "r");

status = ProTKFscanf(fptr_unicode, "\357\273\277%s", param_value_str);

param_value.type = PRO_PARAM_STRING;

ProStringToWstring(param_value.value.s_val, param_value_str);

/*-----*\
Check for a model. Create a string type parameter with value set to
the string read above.
\*-----*/
status = ProMdlCurrentGet(&model);

status = ProMdlNameGet(model, mdl_name);

status = ProMdlToModelitem(model, &modelitem);

ProStringToWstring(param_name, "TEST_UNICODE_PARAM");

status = ProParameterCreate(&modelitem, param_name, &param_value,
&param);

return(1);
}

```

Support for Legacy Encoding in Pro/TOOLKIT

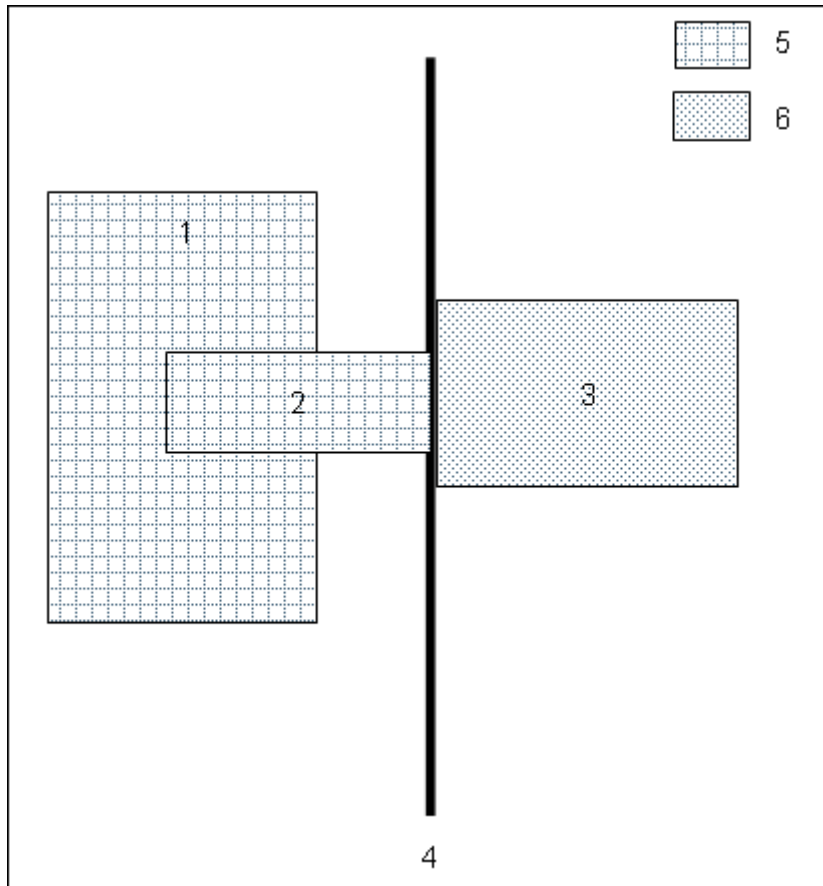
To limit the immediate impact of the change to Unicode encoding on existing Pro/TOOLKIT applications, Pro/TOOLKIT provides a mode where an existing application can exchange information with Pro/ENGINEER Wildfire 4.0 and later using the legacy string and widestring encoding.

The legacy encoding mode should allow a gradual transition to a Unicode-compliant application. However, consider the following points when using this mode in Pro/ENGINEER Wildfire 4.0 and later:

- This mode will be available for a limited set of releases. PTC currently plans to offer this mode in Pro/TOOLKIT Wildfire 4.0 and the following release. After these releases, this mode will not be available and all Pro/TOOLKIT applications will need to accept data in Unicode encoding.
- The automatic transcoding that takes place at the interface level of Pro/TOOLKIT will have a negative impact on the performance of the application, when compared to the same application running with Unicode encoding.
- The automatic transcoding involves an extra layer of code for each function, including a requirement to handle and duplicate allocated memory. It is possible that you may encounter problems for a function in the legacy mode that do not occur when running in the default Unicode encoding.
- The automatic transcoding ensures that data from non-matching languages (such as German text extracted in a Japanese Pro/ENGINEER session, or Japanese text extracted in an English Pro/ENGINEER session) will not be valid. If a transcoding error occurs during the output of a string from Pro/ENGINEER to an application running in the legacy mode, the string will be truncated to the point of the first transcoding error, and the function will return `PRO_TK_COMM_ERROR` to indicate the problem. Thus, running in the legacy mode may result in a loss of data integrity or a change in the application logic when encountering a model containing strings from more than one language.
- When automatic transcoding takes place for `char*` data, there is a small chance that the required native string is larger than the size of the Unicode string, and it does not fit in the buffer available for the application. In such situations, the strings are truncated to the nearest whole multibyte character. No warning or error will be flagged for this situation.

The following figure shows the interaction between the application in the legacy encoding and Pro/ENGINEER.

Figure B-3: Application running in legacy encoding



- 1 Legacy encoding used by the application
- 2 Pro/TOOLKIT application
- 3 Pro/ENGINEER
- 4 This boundary indicates where transcoding takes place
- 5 Items with this pattern are operating in the legacy encoding
- 6 Items with this pattern are operating in the Unicode encoding

Using Legacy Encoding Mode

A new registry file option `unicode_encoding` allows the Pro/TOOLKIT synchronous mode application to run in the Pro/ENGINEER Wildfire 4.0 and later environment in the legacy-encoded format. The default value of this option is `true`, indicating that any application will run using Unicode encoding, by default.

Note: All Pro/TOOLKIT DLLs loaded using **ProToolkitDllLoad()** by an application running in the legacy mode will automatically run using legacy encoding.

Asynchronous mode applications can also run in the legacy mode. Since there is no registry file for asynchronous mode applications, these applications must call a new function **ProToolkitUnicodeencodingDisable()** before attempting to start or connect to Pro/ENGINEER. After this function is executed, all the connections established between the asynchronous application and Pro/ENGINEER use legacy encoding.

Functions Introduced:

- **ProToolkitUnicodeencodingDisable()**
- **ProToolkitUnicodeencodingIsEnabled()**

The function **ProToolkitUnicodeencodingDisable()** sets the Pro/TOOLKIT asynchronous mode application to run with legacy-encoded strings, instead of the default Unicode encoding. This function must be called from the asynchronous mode, when your applications wants to run in the legacy encoding mode.

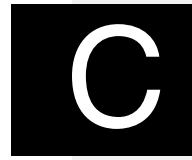
The function **ProToolkitUnicodeencodingIsEnabled()** identifies whether a Pro/TOOKIT application is running with legacy or Unicode encoding. For asynchronous mode applications, this function should be called only after the connection with Pro/ENGINEER has been established.

Legacy Applications

Pro/TOOLKIT applications built in Pro/ENGINEER Wildfire 3.0 and earlier are assumed to require legacy encoding. If Pro/ENGINEER encounters an application built with this earlier library, but the registry file option `unicode_encoding` is not set to `false`, Pro/ENGINEER will not run the application. Asynchronous applications need to be rebuilt in Pro/ENGINEER Wildfire 4.0 and later to override legacy encoding because these applications cannot call the function **ProToolkitUnicodeencodingDisable()**.

Applications built before Pro/ENGINEER Release 2001 are not supported in Pro/ENGINEER Wildfire 4.0 and later. The following error message will be displayed in the interactive mode when trying to start an application of this age:

```
Application "name" was built before Pro/ENGINEER Release
2001. Due to encoding changes in this version of
Pro/ENGINEER, this application cannot be started.
```



ptk_revtool Utility

This appendix describes the purpose and use of the `ptk_revtool` utility. `ptk_revtool` allows users to update legacy Pro/TOOLKIT applications to run with more recent Pro/TOOLKIT versions.

Topic	Page
Overview	C - 2
<code>ptk_revtool</code> Utility Features	C - 2
Running <code>ptk_revtool</code>	C - 3

Overview

The `ptk_revtool` utility allows Pro/TOOLKIT users to update Pro/TOOLKIT applications from older versions. Updating applications allows use of functions that have been updated in the current release.

The update process renames old functions and include files by incorporating a version number into their name. Renaming preserves the capability of the old function. `ptk_revtool` keeps renamed functions separate from newer versions of the same functions. `ptk_revtool` also marks old and obsolete functions. This marking helps Pro/TOOLKIT developers to update their applications.

Pro/TOOLKIT users are responsible for migrating their applications to use newer Pro/TOOLKIT functions. Renaming and reservation of older Pro/TOOLKIT functions should be an interim step only. PTC recommends updating legacy application code to use new versions of Pro/TOOLKIT functions.

Note: This utility updates Pro/TOOLKIT applications created with Pro/TOOLKIT version 19 or greater.

ptk_revtool Utility Features

The `ptk_revtool` utility contains a running history of changes to function and include file names since Pro/TOOLKIT version 19. Each name in the list maps to a newer function or file version that has the same name. To maintain the existing application functionality, users must run `ptk_revtool` to rename each function and include file.

`ptk_revtool` supports upgrades from Pro/TOOLKIT version 19 or greater. The default upgrade is one version forward. The `ptk_revtool` utility finds each function and include file in the legacy application and renames them from the old version to the current Pro/TOOLKIT version.

`ptk_revtool` directly edits, updates, and saves the original application file. The utility creates a backup of the original file and appends a number (.1, .2, and so on) to the end of the filename.

The rename process changes **function_name** to **function_nameR##**, where ## is the number of the previous version. The ## suffix preserves the function in its old state and distinguishes it from the newest version. For example:

- Upgrade from version 19:
ProLayerCreate() becomes **ProLayerCreateR19()**
- Upgrade from version 20:
prodrw_rotate_table() becomes **prodrw_rotate_tableR20()**

If you provide only an input file name, `ptk_revtool` creates an updated file that has the same file name with `.1` appended.

`ptk_revtool` supports output function names of 32 characters or less. If a renamed function name is too long, `ptk_revtool` returns a warning message. The new function is named **function_nameR##**, unless this name would be more than 32 characters long. If the name is too long, the second field is abbreviated.

Running ptk_revtool

The `ptk_revtool` utility runs in both the UNIX and DOS environments. Run the utility from the command prompt.

Syntax

To run the utility, use the following command syntax:

```
ptk_revtool.exe -# input_file
```

Where:

Part	Description
-#	Applies all changes since the specified version #. If a number is not included, then the default is one version upgrade. Use -? to display the syntax statement for the program as a reminder.
input_file	File name of the Pro/TOOLKIT application that you want to update.

For example, if you run the utility on the file `testgeom.c` to update it from version 19 to the newest version, the command would be:

```
ptk_revtool -19 testgeom.c
```

Output Messages

After running the utility, one of the following confirmation or warning messages may display.

Output Message	Description
updated file filename.c	The specified file was successfully updated.
no symbols to change	The -# option wasn't provided. No changes made.
no symbols changed in file filename	The file is already up-to-date. No changes were made to the file.
ptk_revtool: can't open file	File not found. Check the file name and path. You must enter full file name, including extension.
bad arg '-#'; version must be >= 19	You entered a version number less than 19. The program works only on version 19 or greater.
no symbols for version	-
Warning: too-long, MY_NAME written to	Output file name greater than 32 characters.



Pro/TOOLKIT Registry File

This appendix describes how to use the Registry file to have a foreign program communicate with Pro/ENGINEER.

Topic	Page
Registry File Fields	D - 2
Sample Registry Files	D - 4

Registry File Fields

The following table lists the fields in the registry file, *protk.dat*.

Field	Description
<i>name</i>	Assigns a unique name to the Pro/TOOLKIT application. The name is used to identify the application if there is more than one. The name can be the product name and does not have to be the same as the executable name. The maximum size of the name is <code>PRO_NAME_SIZE</code> (defined in the file <i>ProSizeConst.h</i>). Currently the maximum size is 31 characters for the name, plus the end-of-string character.
<i>startup</i>	Specifies the method Pro/ENGINEER should use to communicate with the Pro/TOOLKIT application. This field can take one of three values; <code>spawn</code>, <code>dll</code> or <code>java</code>. • <code>spawn</code>— If the value is <code>spawn</code>, Pro/ENGINEER starts the foreign program using interprocess communications. • <code>dll</code> – If the value is <code>dll</code>, Pro/ENGINEER loads the foreign program as a DLL. • <code>java</code> – If the value is <code>java</code>, Pro/ENGINEER starts the application as a J-Link class. Consult the J-Link User's Guide for more details. The default value is <code>spawn</code>.
<i>fail_tol</i>	Specifies the action of Pro/ENGINEER if the call to <code>user_initialize()</code> in the foreign program returns nonzero, or if the foreign program subsequently fails. If this is <code>TRUE</code>, Pro/ENGINEER continues as normal. If this is <code>FALSE</code>, Pro/ENGINEER shuts down Pro/ENGINEER and other foreign programs.
<i>exec_file</i>	Specifies the full path and name of the file produced by compiling and linking the Pro/TOOLKIT application. In DLL mode, this is a dynamically linkable library; in <code>spawn</code> mode, it is a complete executable. This field has a character limit of <code>PRO_PATH_SIZE-1</code> wide characters (<code>wchar_t</code>).

Field	Description
<i>text_dir</i>	Specifies the full path name to text directory that contains the language-specific directories. The language-specific directories contain the message files, menu files, resource files and UI bitmaps in the language supported by the Pro/TOOLKIT application. Please refer to the ‘User Interface: Menus, Commands, and Popupmenus’ and ‘User Interface: Messages’ chapters for more information. The <i>text_dir</i> does not need to include the trailing “/text”; it is added automatically by Pro/ENGINEER. The search priority for messages and menu files is as follows: 1) Current working directory 2) <i>text_dir</i>/text 3) <proe_loadpoint>/text The <i>text_dir</i> should be different from the Pro/ENGINEER text tree. This field has a character limit of PRO_PATH_SIZE-1 wide characters (wchar_t).
<i>unicode_encoding</i>	Allows a Pro/TOOLKIT synchronous mode application to execute in the legacy string encoding format in the Pro/ENGINEER Wildfire 4.0 environment. The default value of this option is <code>true</code>, indicating that any application will execute using Unicode encoding by default. Applications built in Pro/ENGINEER Wildfire 3.0 and earlier must have this option set to <code>false</code> in the registry file. Refer to the ‘Unicode Encoding’ chapter for more information about how the Pro/ENGINEER string encoding affects Pro/TOOLKIT applications.
<i>delay_start</i>	If you set this to <code>TRUE</code>, Pro/ENGINEER does not invoke the Pro/TOOLKIT application as it starts up, but enables you to choose when to start the application. If this field is missing or is set to <code>FALSE</code>, the Pro/TOOLKIT application starts automatically.
<i>description</i>	Acts as a help line for your auxiliary application. If you leave the cursor on an application in the Start/Stop GUI, Pro/ENGINEER displays the description text (up to 80 characters). You can use non-ASCII characters, as in menu files. To make the description appear in multiple languages, you must use separate <i>protk.dat</i> files in <hierarchy>/<platform>/<text>/<language>.

Field	Description
<i>allow_stop</i>	If you set this to TRUE, you can stop the application during the session. If this field is missing or is set to FALSE, you cannot stop the application, regardless of how it was started.
<i>end</i>	Indicates the end of the description of the Pro/TOOLKIT application. It is possible to add further statements that define other foreign applications. All of these applications are initialized by Pro/ENGINEER.

Sample Registry Files

This section lists several examples that illustrate the various ways to have a foreign program communicate with Pro/ENGINEER.

Note: For delimiter characters in *protk.dat*, use *'* on UNIX platforms and *'\'* on Windows NT platforms.

Example 1

In this example, Pro/ENGINEER spawns the foreign program, which runs on the same machine. The communication is via pipes (the default mode when the foreign program runs on the same machine as Pro/ENGINEER).

File: *protk.dat*

[Start of file on next line]

```
name      Product1
exec_file /home/protk/<machine>/obj/frnpgm1
text_dir  /home/protk
end
```

[End of file on previous line]

Example 2

This example illustrates how to run multiple foreign programs, as specified in the *protk.dat* file.

File: *protk.dat*

[Start of file on next line]

```
name      Product1
startup   dll
```

```
exec_file    /home/protk/<machine>/obj/frnpgm1.dll
text_dir     /home/protk
end
name         Product2
startup      spawn
exec_file     /home/protk2/<machine>/obj/frnpgm2
text_dir      /home/protk2
end
```

[End of file on previous line]

Pro/TOOLKIT
Registry File



Pro/TOOLKIT Library Types

This appendix describes the variety of libraries available in a Pro/TOOLKIT installation, and explains the differences among them.

Topic	Page
Standard Libraries	E - 2
Alternate Libraries	E - 2

Standard Libraries

Most Pro/TOOLKIT users will be able to use the standard Pro/TOOLKIT libraries. These libraries are available on all platforms and are used by the majority of Pro/TOOLKIT sample applications.

Library Name	Purpose
protoolkit.lib (.a)	Spawn mode library
pt_asynchronous.lib (.a)	Asynchronous mode library
protk_dll.lib (.a)	DLL mode library

Alternate Libraries

Pro/TOOLKIT offers alternate libraries that may be useful for applications compiled with /MD flag and built with `msvcrt.lib`. These libraries are similar to the standard Pro/TOOLKIT libraries in content, but differ in using `msvcrt.lib` instead of `libcmt.lib`.

Library Name	Purpose
protkmd.lib	Spawn mode library
ptasyncmd.lib	Asynchronous mode library
protk_dllmd.lib	DLL mode library

The makefiles `make_install_md` and `make_async_md` build with these libraries.

Note: Although /MD provides compatibility with multi-threaded components, Pro/TOOLKIT calls must be made within a single thread. Pro/ENGINEER does not respond to calls made from multiple threads. Extra threads may be created by applications only to do tasks which do not directly call Pro/TOOLKIT functions.

PIC Libraries for Some UNIX platforms

Library name	Purpose
protkpic.a	Spawn mode library
ptasyncpic.a	Asynchronous mode library

Position Independent Code (PIC) libraries are used to build an executable for some UNIX platforms using the appropriate PIC compiler flags. Typically this is required when compiling an asynchronous mode shared library that will be loaded into an external process. The makefiles `make_install_pic` and `make_async_pic` build with these libraries.



Pro/TOOLKIT Sample Applications

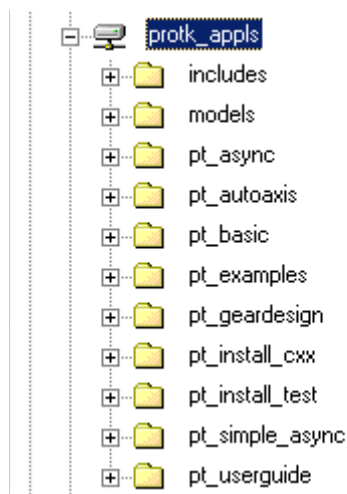
This appendix describes the sample applications provided with Pro/TOOLKIT.

Topic	Page
Installing Sample Applications	F - 2
Details on Sample Applications	F - 2

Installing Sample Applications

When you install Pro/TOOLKIT from the Pro/ENGINEER CD, Pro/TOOLKIT is installed under the loadpoint of Pro/ENGINEER. Refer to the Fundamentals chapter for more information on Installation of Pro/TOOLKIT.

The Pro/TOOLKIT directory contains all the headers, libraries, example applications, and documentation specific to Pro/TOOLKIT. The following diagram illustrates the applications installed under the `protk_appls` directory after installation.



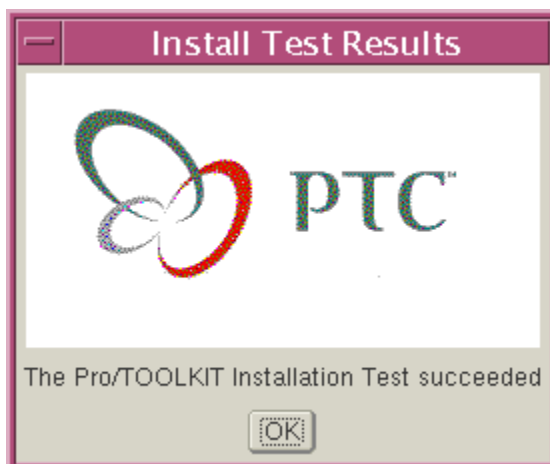
Details on Sample Applications

The sample applications provided with Pro/TOOLKIT are available in directories under the following path *protoolkit/protk_appls*.

pt_inst_test

Location	Makefile
protoolkit/protk_appls/pt_install_test	<i>make_install</i>

The application `pt_inst_test` is used to check the Pro/TOOLKIT Installation. It verifies the ProMenubar and custom user interface dialog box functions.

**pt_inst_cxx**

Location	Makefile
protoolkit/protk_appls/pt_install_cxx	<i>make_install_cxx</i>

The application `pt_inst_cxx` is used to check the C++ version of the Pro/TOOLKIT installation. It verifies the ProMenubar and custom user interface dialog box functions that use the C++ compiler and classes.

pt_inst_test_md

Location	Makefile
protoolkit/protk_appls/pt_install_test	<i>make_install_md</i>

The application `pt_inst_test_md` provides the version of the makefile for Windows platforms that uses MD libraries.

MD libraries are intended for building a DLL for Windows. Some Microsoft libraries must be linked with these libraries. For more information on MD libraries, refer to “Alternate Libraries”.

pt_autoaxis

Location	Makefile
protoolkit/protk_appls/pt_autoaxis	<i>make_autoaxis</i>

The sample application `pt_autoaxis` automatically creates axes on revolved surfaces if they don't already exist. It is intended for manufacturing engineers who receive a model with holes which were not made with standard Pro/ENGINEER hole features. It covers functions for feature creation and geometry analysis.

pt_userguide

Location	Makefile
protoolkit/protk_appls/pt_userguide	<i>make_userguide</i>

The sample application `pt_userguide` consolidates examples that access the User Interface. These examples are also provided in the Pro/TOOLKIT User's Guide. It covers all the examples in the Pro/TOOLKIT User's Guide, and the ProMenubar and ProCmd UI functions.

pt_examples

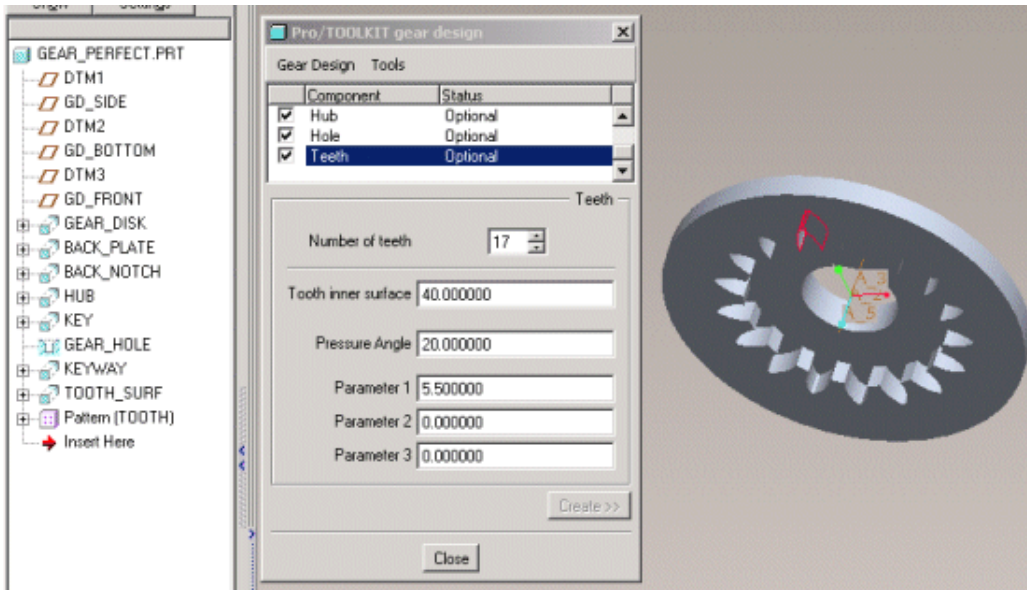
Location	Makefile
protoolkit/protk_appls/pt_examples	<i>make_examples</i>

The sample application `pt_examples` consists of Pro/TOOLKIT application examples. It provides a user interface to access many areas of Pro/TOOLKIT. This application covers Pro/TOOLKIT functions and modules, including ProMenubar and ProMenu based UI functions. The directory `pt_examples` includes sub-directories containing useful utility functions.

pt_geardesign

Location	Makefile
protoolkit/protk_appls/pt_geardesign	<i>make_geardesign</i>

The sample application `pt_geardesign` provides the user interface to create gear models. This application covers feature creation using element trees and the custom user interface dialog box functions.



pt_async

Location	Makefile
protokit/protk_appls/pt_async	<i>make_async</i>

The sample application `pt_async` provides an example for the full asynchronous mode.

pt_async_md

Location	Makefile
protokit/protk_appls/pt_async	<i>make_async_md</i>

The sample application `pt_async_md` provides an example for the asynchronous mode compilation of a DLL using MD libraries on Windows. The application requires two binaries, namely, `pt_async_md.dll`, and the wrapper executable `pt_async_md_wrapper.exe` which will load the DLL and invoke it. The application is identical to `pt_async` in all other respects. For more information on MD libraries, refer to “Alternate Libraries”.

pt_async_pic

Location	Makefile
protoolkit/protk_appls/pt_async_pic	<i>make_async_pic</i>

The sample application `pt_async_pic` provides examples for compilation in asynchronous mode using PIC libraries on UNIX platforms. The application is identical to `pt_async` in all other respects. For more information on PIC libraries, refer to “PIC Libraries for Some UNIX platforms”.

pt_simple_async

Location	Makefile
protoolkit/protk_appls/pt_simple_async	<i>make_simple_async</i>

The sample example `pt_simple_async` provides an example for the simple asynchronous mode.

pt_basic

Location	Makefile
protoolkit/protk_appls/pt_basic	<i>make_basic</i>

The sample application `pt_basic` provides the Pro/TOOLKIT application template. It verifies the user interface, notifications, and the application setup.

pt_af_examples

Location	Makefile
protoolkit/protk_appls/pt_af_examples	<i>make_af_examples</i>

The sample application `pt_af_examples` provides production examples using annotation features and annotations.

pt_udf_examples

Location	Makefile
protoolkit/protk_appls/pt_udf_examples	<i>make_udf_examples</i>

The sample application `pt_udf_examples` demonstrates production examples using UDF placement capabilities. These include a flexible user interface for placing UDF libraries interactively, by reading the properties of the UDF library file.

pt_mech_examples

Location	Makefile
protoolkit/protk_appls/pt_mech_examples	<i>make_mech_examples</i>

The sample application `pt_mech_examples` provides production examples using the ability to assign, create and modify Mechanical item properties.

pd_examples

Location	Makefile
prodevelop/prodev_appls/pd_examples	<i>make_example</i> (prodevelop/<machine_type>/obj)

The sample application `pd_examples` provides Pro/Develop application examples, including user interface access to different areas of Pro/Develop. This application covers Pro/Develop functions and modules, and ProMenubar and ProMenu based user interface functions.

Note: Most areas of Pro/Develop have been superseded and have equivalent functions in Pro/TOOLKIT. The Pro/Develop examples are maintained for legacy purposes only.



Advanced Licensing Options

This chapter describes the licensing requirements for advanced options in Pro/TOOLKIT.

Topic	Page
Advance Licensing Options for Pro/TOOLKIT	G - 2

Advance Licensing Options for Pro/TOOLKIT

To use some of the new functionality introduced in Pro/TOOLKIT, for Pro/ENGINEER Wildfire 3.0, you require special development license options. PTC offers the following development license packages:

- Pro/TOOLKIT for 3D Drawings
 - A comprehensive library of functions to create, modify, and to extract information from annotation features and annotation elements.
 - Increased support for solid model annotations, including notes, surface finish, geometric tolerances, symbols, and dimensions.
 - The Quick Print capability for automatically generating a plot of a solid model without requiring a drawing.
- Pro/TOOLKIT for Mechanica
 - Read and write access for Mechanica items stored in the Pro/ENGINEER model, including loads, constraints, shells, springs, beams, masses, and their related properties.
 - Read access to the non-manifold geometric representations used by the Mechanica package for analyses.

The Pro/TOOLKIT header file entry and the API Wizard description page for each of the functions from the above packages include a note specifying the development license requirement. Advanced licenses are required in the following situations:

- When running with a locked application, Pro/ENGINEER will require the basic Pro/TOOLKIT development option and must detect the presence of the advanced options that the application is coded to call. (Pro/ENGINEER will not check out the advanced license option).
- When unlocking an application, Pro/ENGINEER requires the basic Pro/TOOLKIT development option and must detect the presence of all of the advanced options required by the functions that the application is coded to call. (The unlock utility will not hold the advanced license option as it does the basic option after unlock is completed).
- After properly unlocking the application, Pro/ENGINEER does not require any of the Pro/TOOLKIT licenses to run the application.

Applications are assigned requirements for advanced options based on whether the application is coded to use any functions requiring the special option. This means that some object or library in the application code is referencing the Pro/TOOLKIT function symbol, so that the linker will try to resolve the symbol during the linking process. It does not matter if an application does not use the function requiring licensing during a particular invocation of the application. The licensing requirements are resolved the moment the application is started by or connects to Pro/ENGINEER, not at the first time an advanced function is invoked.

For more information on how to unlock an application, refer to the section Unlocking a Pro/TOOLKIT Application.



Digital Rights Management

This appendix describes the implications of DRM on Pro/TOOLKIT applications.

Topic	Page
Introduction	H - 2
Implications of DRM on Pro/TOOLKIT	H - 2
Additional DRM Implications	H - 6

Introduction

Digital Rights Management (DRM) helps to control access to your intellectual property. Intellectual property could be sensitive design and engineering information that you have stored within Pro/ENGINEER parts, assemblies, or drawings. You can control access by applying policies to these Pro/ENGINEER objects. Such objects remain protected by the policies even after they are distributed or downloaded. Pro/ENGINEER objects for which you have applied policies are called DRM-protected objects. For more information on the use of DRM in Pro/ENGINEER Wildfire 4.0, refer to the DRM online help.

The following sections describe how Pro/TOOLKIT applications deal with DRM protected objects.

Implications of DRM on Pro/TOOLKIT

Any Pro/TOOLKIT application accessing DRM protected objects can run only in interactive Pro/ENGINEER sessions having COPY permissions. As Pro/TOOLKIT applications can extract content from models into an unprotected format, Pro/TOOLKIT applications will not be able to run in a Pro/ENGINEER session lacking COPY permission.

If the user tries to open a model lacking the COPY permission into a session where a Pro/TOOLKIT application is already running, Pro/ENGINEER will ask the user to spawn a new session. Also, new Pro/TOOLKIT applications will not be permitted to start when the Pro/ENGINEER session lacks COPY permission.

If a Pro/TOOLKIT application tries to open a model lacking COPY permission from an interactive Pro/ENGINEER session, the application fails with the error `PRO_TK_NO_PERMISSION`.

When a Pro/TOOLKIT application tries to open a protected model from a non-interactive or batch mode application, the session cannot prompt for DRM authentication, instead the application fails with the error `PRO_TK_AUTHENTICATION_FAILURE`.

Error Return Types

Some Pro/TOOLKIT functions require specific permissions in order to operate on a DRM protected object. If these functions cannot proceed due to DRM restrictions, the following errors are returned:

- **PRO_TK_NO_PERMISSION**—Returned if the function cannot proceed due to lack of needed permissions.
- **PRO_TK_AUTHENTICATION_FAILURE**—Returned if the object cannot be opened because the policy server could not be contacted or if the user was unable to interactively login to the server.
- **PRO_TK_USER_ABORT**—Returned if the object cannot be operated upon because the user cancelled the action at some point.

The following table lists the functions along with the permission required for operating on DRM protected objects.

Functions	Permission Required	Implications
ProAssemblySimpRepRetrieve() ProDrawingFromTmplCreate() ProGraphicsSimpRepRetrieve() ProGeomSimpRepRetrieve() ProMdlLoad() ProMdlMultipleRetrieve() ProMdlRetrieve() ProMdlfileCopy() when input model is not in session ProPartSimpRepRetrieve() ProSymbSimpRepRetrieve()	OPEN	• If file has OPEN and COPY permissions, model opens after authentication. • Returns PRO_TK_NO_PERMISSION otherwise.
ProMdlRename()	OPEN	• File is saved to disk with the current policy if it has COPY permission.
proddb_get_simplfd_rep_list()	OPEN	• Returns an error if file lacks OPEN and COPY permission.
proddb_rename_object()	OPEN	• File is saved to disk with the current policy if it has COPY permission.

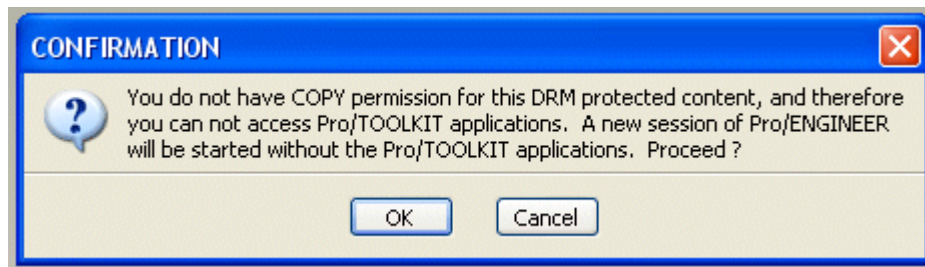
Functions	Permission Required	Implications
proddb_retrieve_object() proddb_retrieve_simplfd_rep() proddb_copy_object() when input model is not in session	OPEN	• If file has OPEN and COPY permissions, model opens after authentication. • Returns an error otherwise.
ProMdlBackup() ProMdlCopy() ProMdlfileCopy()	SAVE	• File is saved to disk with the current policy if it has SAVE and COPY permissions. • Returns PRO_TK_NO_PERMISSION if model has COPY permission but lacks SAVE permission.
ProMdlSave()	SAVE	• File is saved to disk with the current policy if it has SAVE and COPY permissions. • Returns PRO_TK_NO_PERMISSION if model has COPY permission but lacks SAVE permission. • Returns PRO_TK_NO_PERMISSION if the assembly file has models with COPY permission but lacking SAVE permission.
proddb_backup_object() proddb_copy_object() proddb_save_object()	SAVE	• File is saved to disk with the current policy if it has SAVE and COPY permissions. • Returns an error if model has COPY permission but lacks SAVE permission.
ProPlotfileWrite() ProRasterFileWrite() ProPDFExport() ProProductviewFormattedExport() only if the input model is a drawing	PRINT	• Drawing file is printed if it has PRINT permission. • Returns PRO_TK_NO_PERMISSION if drawing file lacks PRINT permission.
pro_export_plot_file() pro_export_plot_file() prointerface_2d()	PRINT	• Drawing file is printed if it has PRINT permission. • Returns an error if drawing file lacks PRINT permission.

Functions	Permission Required	Implications
ProQuickprintExecute()	PRINT	3D model file is printed if it has PRINT permission. - Returns PRO_TK_NO_PERMISSION if 3D model file lacks PRINT permission.

Copy Permission to Interactively Open Models

When the user tries to open protected content lacking COPY permission through the Pro/ENGINEER user interface with a Pro/TOOLKIT application running in the same session:

1. Pro/ENGINEER checks for the authentication credentials through the user interface, if they are not already established.
2. If the user has permission to open the file, Pro/ENGINEER checks if the permission level includes COPY. If the level includes COPY, Pro/ENGINEER opens the file.
3. If COPY permission is not included, the following message is displayed:



4. If the user clicks **Cancel**, the file is not opened in the current Pro/ENGINEER session and no new session is spawned.
5. If the user clicks **OK**, an additional session of Pro/ENGINEER is spawned which does not permit any Pro/TOOLKIT applications. Pro/TOOLKIT applications set to automatically start by Pro/ENGINEER will not be started. Asynchronous applications will be unable to connect to this session.
6. The new session of Pro/ENGINEER is automatically authenticated with the same session credentials as were used in the previous session.

7. The model that Pro/ENGINEER was trying to load in the previous session is loaded in this session.
8. Other models already open in the previous session will **not** be loaded in the new session.
9. Session settings from the previous session will not be carried into the new session.
10. The new session will be granted the licenses currently used by the previous session. This means that the next time the user tries to do something in the previous session, Pro/ENGINEER must obtain a new license from the license server. If the license is not available, the action will be blocked with an appropriate message.

Additional DRM Implications

- **ProMdlIsSaveAllowed()** returns PRO_B_FALSE if prevented from save by DRM restrictions.
- **ProFileCopyToWS()** is designed to fail returning PRO_TK_NO_PERMISSION if passed a DRM protected Pro/ENGINEER model file.
- **ProCurrentWorkspaceImport()** reports a conflict in its output text file and does not copy a DRM protected Pro/ENGINEER model file to the Workspace.
- While erasing an active Pro/ENGINEER model with DRM restrictions, **ProMdlErase()** and **ProMdlEraseAll()** do not clear the data in the memory until the control returns to Pro/ENGINEER from the Pro/TOOLKIT application. Thus, the Pro/ENGINEER session permissions may also not be cleared immediately after these functions return.



Pro/DEVELOP to Pro/TOOLKIT Function Mapping

This appendix describes how to update legacy applications using Pro/DEVELOP functions with current Pro/TOOLKIT functions.

Topic	Page
The Relationship Between Pro/TOOLKIT and Pro/DEVELOP	I - 2
Converting from Pro/DEVELOP	I - 2
Equivalent Pro/DEVELOP Functions	I - 16
Data Structures for Pro/DEVELOP-Style Functions	I - 30

The Relationship Between Pro/TOOLKIT and Pro/DEVELOP

Pro/TOOLKIT replaces and contains Pro/DEVELOP, the customization toolkit until Release 17 of Pro/ENGINEER. Pro/TOOLKIT uses an Object-Oriented style. Many of the library functions familiar to experienced users of Pro/DEVELOP have been replaced by equivalent Pro/TOOLKIT functions. Users do not have to convert existing Pro/DEVELOP applications to use Pro/TOOLKIT because:

- Pro/TOOLKIT uses the same methods as Pro/DEVELOP for integrating the application code into Pro/ENGINEER, so Pro/DEVELOP code and Pro/TOOLKIT code can be mixed within a single application.
- Superseded Pro/DEVELOP functions remain in the Pro/ENGINEER library for an indefinite period.

The first point above also implies that existing Pro/DEVELOP applications can take advantage of new Pro/TOOLKIT features without converting code that uses Pro/DEVELOP-style functions.

In this release of the product, some special areas of Pro/DEVELOP functions are not replaced by equivalent Pro/TOOLKIT-style functions. Users of those functional areas must mix Pro/DEVELOP and Pro/TOOLKIT functions to take advantage of new Pro/TOOLKIT features.

The techniques for mixing Pro/DEVELOP-style and Pro/TOOLKIT-style functions are described in f.

Pro/TOOLKIT OHandles:

Pro/TOOLKIT OHandles are equivalent to the type *Prohandle* used in Pro/DEVELOP. You can convert the handles between Pro/TOOLKIT and Pro/DEVELOP simply by casting to the appropriate type. Pro/TOOLKIT provides different OHandles for different object types where Pro/DEVELOP provided only a single generic handle; this provides for better type-checking during compilation. See Converting from Pro/DEVELOP for more details.

Converting from Pro/DEVELOP

You can convert functions from Pro/DEVELOP to Pro/TOOLKIT, and also mix the two styles of functions.

Using Pro/DEVELOP Applications with Pro/TOOLKIT

Pro/TOOLKIT replaces Pro/DEVELOP and provides most of the functionality that existed in Pro/DEVELOP. Existing Pro/DEVELOP applications will not become obsolete however, for the following reasons:

- Pro/TOOLKIT inherits from Pro/DEVELOP the mechanisms by which the application C code is integrated into Pro/ENGINEER. These mechanisms will continue to be used by Pro/TOOLKIT for the indefinite future.
- The complete library of Pro/DEVELOP functions is installed automatically along with the library of Pro/TOOLKIT functions, and will be installed in this way from Release 2000i² on.

Therefore, Pro/DEVELOP applications built using Release 17 will continue to work with Pro/ENGINEER Release 18 and later, without having to be recompiled and relinked. Using Pro/TOOLKIT, you can recompile and relink Pro/DEVELOP applications developed using Release 17 without having to change the source code. These applications will continue to function as before.

However, you should plan to convert your applications to Pro/TOOLKIT as soon as possible, even if you do not need to use any of the new functionality provided by Pro/TOOLKIT. The conversion is desirable because Pro/TOOLKIT provides more consistent and complete functionality, even in areas already well-covered by Pro/DEVELOP. In addition, PTC will give lower priority to requests for enhancements and maintenance to Pro/DEVELOP functions than to requests for equivalent Pro/TOOLKIT functions, where they exist.

Thanks to the technology they share, you can use functions from both Pro/TOOLKIT and Pro/DEVELOP within a single application. This means that:

- You can convert a Pro/DEVELOP application to use Pro/TOOLKIT functions gradually.
- Pro/DEVELOP applications can use the new functionality provided by Pro/TOOLKIT without the immediate need for a complete conversion.

A final reason for wanting to mix Pro/DEVELOP and Pro/TOOLKIT functions is that not all of the Pro/DEVELOP functions have been replaced by equivalent Pro/TOOLKIT functions.

Techniques of Conversion and Mixing

Besides a difference in the conventions they use, Pro/DEVELOP and Pro/TOOLKIT reference items in the Pro/ENGINEER database in different ways. The following sections describe the technical points to consider when you convert from Pro/DEVELOP to Pro/TOOLKIT, or when you mix both types of functions in a single application.

Terminology

In general, the terminology used by Pro/TOOLKIT is close to that of Pro/DEVELOP. The following table lists the most important terms that differ in meaning between the two toolkits.

Pro/TOOLKIT	Pro/DEVELOP
Object	N/A
Model	Object
Solid (part or assembly)	Model
Surface	Face or surface
Component of an assembly	Member
Component path	Member identifier table (<i>memb_id_tab</i>)
External data	Generic application data

General Functionality

To find the functions in Pro/TOOLKIT that cover a particular area of functionality, scan the chapters in this user's guide, or use the Topical option in the Pro/TOOLKIT browser. Beware of any difference in terminology from Pro/DEVELOP identified in the previous section.

If you want to find the Pro/TOOLKIT equivalent of a particular Pro/DEVELOP function, refer to the table Equivalent Pro/DEVELOP Functions. The table maps each Pro/DEVELOP function to the closest equivalent Pro/TOOLKIT function (or functions).

In some functional areas, especially where Pro/DEVELOP provided good coverage, you can use the equivalent Pro/TOOLKIT functions in an identical way, although the function names, return values, and sometimes the order of the arguments have been changed to conform to Pro/TOOLKIT conventions.

For example, the following Pro/DEVELOP functions are almost exactly equivalent to the Pro/TOOLKIT functions listed.

Pro/DEVELOP Function	Pro/TOOLKIT Function
promenu_create()	ProMenuFileRegister()
promenu_expand()	ProMenuAuxfileRegister()
promenu_on_button()	ProMenubuttonActionSet()

Other functions require more care, however. For example, one of the conventions of Pro/TOOLKIT is that the input arguments come before the output arguments.

In some areas of functionality, traditional Pro/DEVELOP techniques have been replaced in Pro/TOOLKIT by techniques that are more general, flexible, and consistent with the techniques used within Pro/ENGINEER. A good example is the visit functions, which replace two different Pro/DEVELOP techniques. For example:

Pro/DEVELOP Function	Pro/TOOLKIT Equivalent
proddb_get_feature_ids()	ProSolidFeatVisit()
proddb_first_part_face(), proddb_next_part_face()	ProSolidSurfaceVisit()

It is possible to use the Pro/TOOLKIT visit functions to create a utility that follows one of the Pro/DEVELOP styles. An example is shown in the section Expandable Arrays.

Some areas of Pro/TOOLKIT functionality reveal a more general, and more consistent, view of the contents of the Pro/ENGINEER database than that familiar to users of Pro/DEVELOP, and therefore require a slightly deeper understanding. For example, Pro/TOOLKIT does not contain exact equivalents of the following Pro/DEVELOP functions for traversing the components of an assembly:

- **proddb_first_member()**
- **proddb_next_member()**

Assembly components (called “members” in Pro/DEVELOP) are represented as features in the Pro/ENGINEER database, so these two functions can be replaced by a call to **ProSolidFeatVisit()**, using **ProFeatureTypeGet()** to identify the features of type PRO_FEAT_COMPONENT. The feature identifier for an assembly component is identical to the member identifier used in Pro/DEVELOP.

In the same way, the following Pro/DEVELOP functions that find datum planes and datum curves are also replaced by more generic functions in Pro/TOOLKIT:

- **proddb_first_datum()**
- **proddb_next_datum()**
- **proddb_get_datum_curves()**

Here, too, the first step is to traverse the features using **ProSolidFeatVisit()**. You can then traverse all the geometrical items in a feature using **ProFeatureGeomitemVisit()**. Datum planes are geometry items of type PRO_SURFACE, in features of type PRO_FEAT_DATUM; datum curves are geometry items of type PRO_CURVE, which can occur in features of many types.

This manual always explains the structure of the Pro/ENGINEER database wherever necessary, without assuming any prior knowledge of the Pro/DEVELOP viewpoint. As shown in the previous examples, if you are converting a Pro/DEVELOP application that traverses Pro/ENGINEER geometry, you should pay particular attention to the ‘Core: 3D Geometry’ chapter.

You can use Pro/TOOLKIT functions to create utilities for the specific cases you need. Many such utilities are provided in the sample code located under the Pro/TOOLKIT loadpoint.

Finally, Pro/TOOLKIT covers whole new areas of functionality that were not supported at all by Pro/DEVELOP, such as the direct programmatic creation of features, including simple kinds of sketched features, datum planes, and manufacturing features. Some Pro/DEVELOP applications, especially those that create features using user-defined features (UDFs), and which customize Pro/MFG, may therefore benefit from a complete redesign to take full advantage of Pro/TOOLKIT.

Registry Files

The Pro/TOOLKIT registry file has the same format as the Pro/DEVELOP registry file. The search path used by Pro/TOOLKIT to find the registry file is like that used by Pro/DEVELOP. However, the file name *prodev.dat* is now *protk.dat*, and the configuration file option PRODEV DAT is now either PROTK DAT or TOOLKIT_REGISTRY_FILE. To convert from Pro/DEVELOP to Pro/TOOLKIT, simply substitute these names.

For an extended period, the search path for the Pro/DEVELOP registry file will continue to be used by Pro/ENGINEER, in addition to the search path for Pro/TOOLKIT. Therefore, you do not need to rename the Pro/DEVELOP registry file or configuration file option immediately.

Menu and Message Files

Although the Pro/DEVELOP functions for accessing menus and messages have been replaced by close equivalents in Pro/TOOLKIT, the menu and message files themselves retain exactly the same form and function. No conversion is necessary.

Unlocking Your Application

The Pro/TOOLKIT script for unlocking a finished application is named *protk_unlock*, but is otherwise identical to *prodev_unlock*.

Application Program Structure

All the Pro/DEVELOP run modes are available in identical form in Pro/TOOLKIT, and the structure of a Pro/TOOLKIT application is the same as that of a Pro/DEVELOP application. Some of the core functions have been given new Pro/TOOLKIT-style names for the sake of consistency, but are otherwise the same. The functions **user_initialize()** and **user_terminate()** remain identical in name and purpose.

Handles and Data Types

Although Pro/TOOLKIT is more rigorous than Pro/DEVELOP in the way it references objects in the Pro/ENGINEER database, there are some close correspondences that simplify the task of mixing Pro/DEVELOP and Pro/TOOLKIT functions.

As a general rule, database items referred to in Pro/DEVELOP by the type *Prohandle*, and referred to as OHandles (opaque handles) in Pro/TOOLKIT, are pointers to the same Pro/ENGINEER data structures. You can directly convert them by casting. The following table lists the most important examples.

Pro/DEVELOP Prohandle for the item type...	Can be cast directly to the Pro/TOOLKIT object
Object	<i>ProMdl</i>
Assembly	<i>ProAssembly</i>
Part	<i>ProPart</i>
Model (part or assembly)	<i>ProSolid</i>
Surface	<i>ProSurface</i>
Contour	<i>ProContour</i>
Edge	<i>ProEdge</i>
Curve	<i>ProCurve</i>
Datum quilt	<i>ProQuilt</i>
Point	<i>ProPoint</i>
Axis	<i>ProAxis</i>
Coordinate system	<i>ProCsys</i>

For database items that can be identified in Pro/DEVELOP by an integer identifier, that identifier is the same one generated by Pro/TOOLKIT functions such as **ProSurfaceIdGet()** and **ProEdgeIdGet()**, and is the same one required as input to functions such as **ProSurfaceInit()**. It is also the value of the *id* field when one of these objects is represented as a *ProGeomitem*.

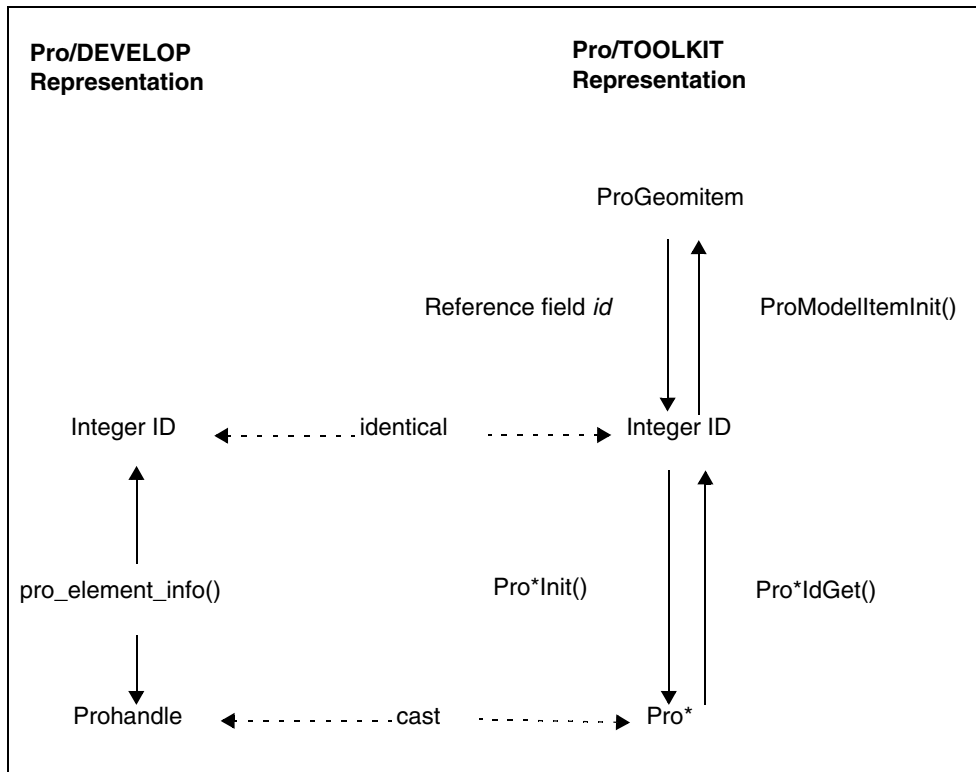
Converting a Pro/DEVELOP integer identifier to a Pro/TOOLKIT OHandle can be done in two ways:

- Convert to a *Prohandle* within Pro/DEVELOP using **pro_element_info()**, then cast the resulting pointer.
- Use the identifier directly as the input to the appropriate **Pro*Init()** function.

The following diagrams show the possible conversion paths between Pro/DEVELOP and Pro/TOOLKIT for database items.

The first diagram applies to objects of type Surface, Edge, Axis, Csys, Curve, Point, and Quilt. In each case, replace the asterisk (*) with the appropriate name.

Figure I-1: Pro/DEVELOP Database Item Conversion Path



Pro/DEVELOP to
Pro/TOOLKIT Function
Mapping

The exception to the previous diagram is that the Pro/DEVELOP function **pro_element_info()** is not supported for coordinate system datums.

A contour does not have an integer identifier in either Pro/DEVELOP or Pro/TOOLKIT, but you can convert the Pro/DEVELOP *Prohandle* to *ProContour* and back by casting.

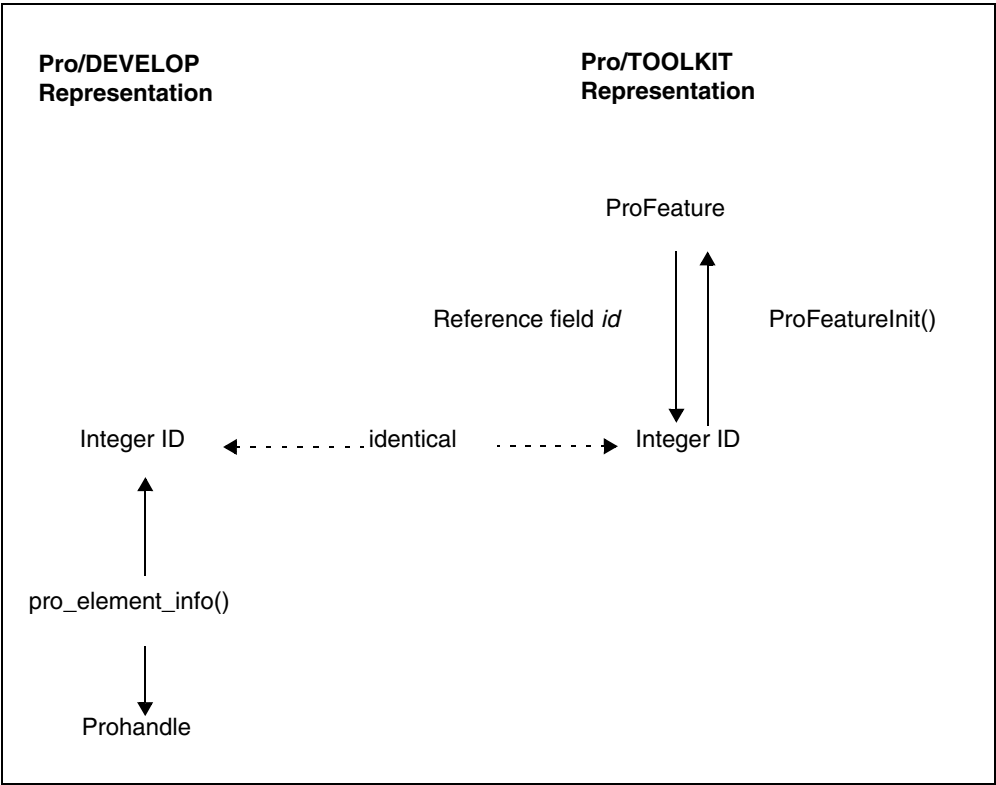
A view does not have an integer identifier in Pro/TOOLKIT, but you can convert the Pro/DEVELOP *Prohandle* to *ProView* by casting.

A vertex is represented in Pro/DEVELOP as an edge (or a curve) and a value for the parameter *t* of either 0 or 1. Pro/TOOLKIT uses this technique in the *ProSelection* object, but for function inputs and in *ProGeomitem* it uses the specific types PRO_EDGE_START and PRO_EDGE_END (and PRO_CRV_START and PRO_CRV_END for datum curve ends). Because PRO_EDGE_START and PRO_CRV_START always refer to the end where *t* = 0, conversion is easy.

A feature is represented in Pro/TOOLKIT by *ProFeature*, which is a DHandle, and therefore *not* equivalent to a Pro/DEVELOP *Prohandle*. The integer identifier still maps directly, however.

The following diagram applies to converting features.

Figure I-2: Feature Conversion



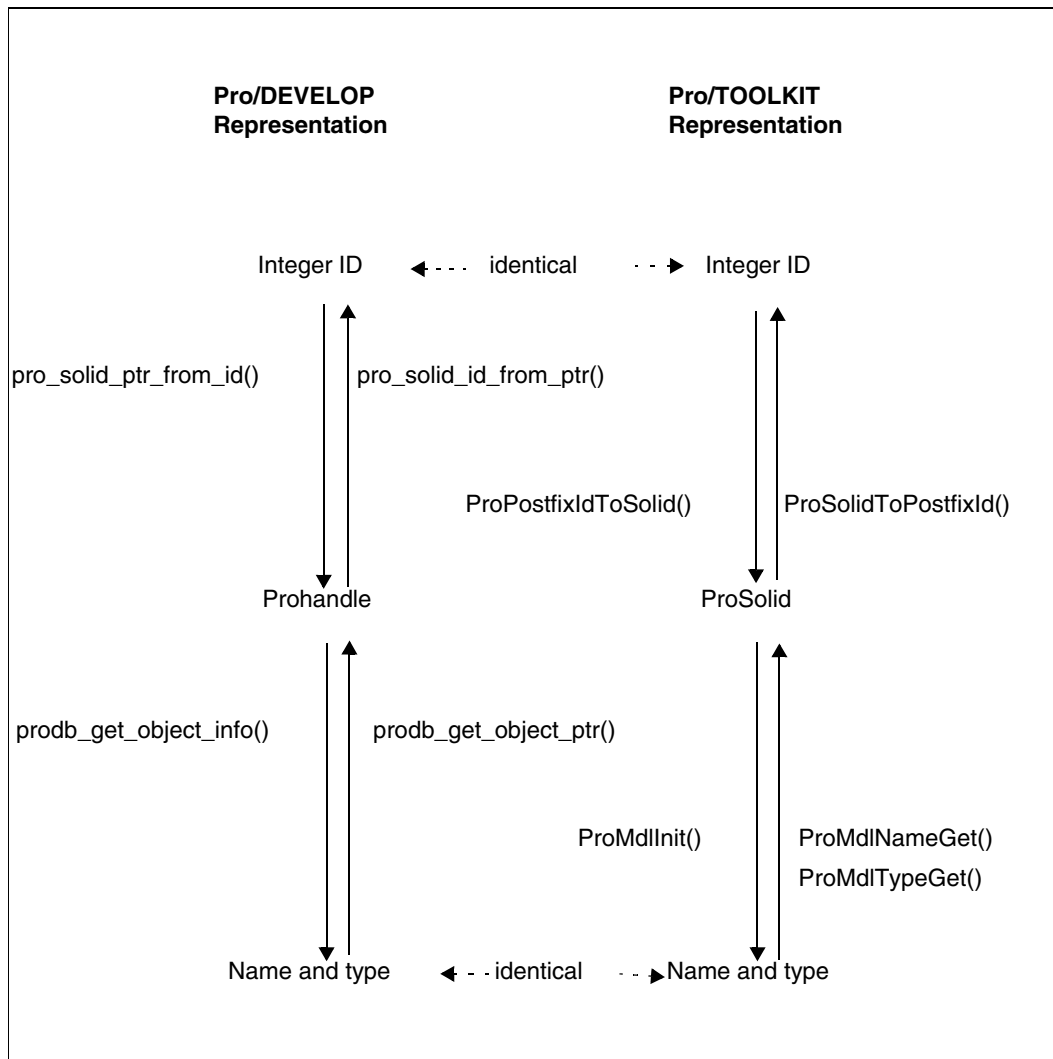
The following objects are DHandles, which are identical in form to ProModelitem and were identified only by an integer identifier in Pro/DEVELOP. They also inherit from ProModelitem, which means that, for example, **ProSelectionModelitemGet()** can be used to unpack them from a ProSelection object after calling **ProSelect()**. In each case, the *id* field in the object handle corresponds to the integer *id* used to identify these objects in Pro/DEVELOP.

ProDimension	ProGtol	ProDtlnote
ProDtlentity	ProDtlsyminst	ProDtlsymdef
ProDtlgroup	ProDgmitem	ProNote

Note: The objects ProDtlnote and ProNote refer to detail (drawing) notes, and notes respectively. They share the same value of the type field - PRO_NOTE - but they are distinguished by the type of their owning model.

The following diagram applies to Pro/TOOLKIT objects *ProSolid*, *ProPart*, and *ProAssembly* when you map to Pro/DEVELOP parts and assemblies. For objects of type *ProMdl* that are not parts or assemblies, the integer identifiers are not applicable, but the rest of the diagram is correct.

Figure I-3: Pro/DEVELOP Part and Assembly Mapping



Many explicit data types from Pro/DEVELOP have been carried across into Pro/TOOLKIT directly, and, although they generally have been given new names, they are directly compatible. In fact, the remaining Pro/DEVELOP include files now reference the new definitions in the Pro/TOOLKIT include files.

Enumerated types have also been given new names for their values, and in some cases where Pro/DEVELOP used *#defined* values of an integer, Pro/TOOLKIT provides an enum. However, the integer mapping of the values is retained in every case. Some examples of Pro/DEVELOP data types now defined in Pro/TOOLKIT are shown in the following table.

Pro/DEVELOP	Pro/TOOLKIT
<i>Ptc_surf</i>	<i>ProSurfacedata</i>
<i>Ptc_curve</i>	<i>ProCurvedata</i>
<i>Pro_linestyle</i>	<i>ProLinestyle</i>
<i>pro_mode</i>	<i>ProMode</i>
<i>Pro_text_attributes</i>	<i>ProTextAttribute</i>
int	<i>ProMousebutton</i>
int	<i>ProDrawMode</i>
int	<i>ProColortype</i>

For some items that have representations in both Pro/TOOLKIT and Pro/DEVELOP, the data structures are not the same, and no direct conversion is possible. However, you can always convert by reducing such structures to their component data items. For example, to convert the Pro/DEVELOP representation of a parameter, *Pro_parameter_info*, to the Pro/TOOLKIT *ProParameter* object, you can use functions such as **ProParameterInit()** and **ProParameterValueSet()**, using the fields in the Pro/DEVELOP structure as inputs. (In this case, it would probably be better to make a complete conversion to the Pro/TOOLKIT functions, and thus avoid mixing these types in an application.)

A more complex example of this is the Pro/DEVELOP Select3d structure (defined in section Select3d Structure Definition), whose Pro/TOOLKIT counterpart is *ProSelection*. The following table explains the mapping by showing the fields of *Select3d* alongside the Pro/TOOLKIT functions that extract the equivalent information from *ProSelection*.

Select3d Field	Pro/TOOLKIT Read Access
<i>sel_type</i>	ProSelectionModelitemGet() , then read <i>modelitem.type</i> . See the note that follows this table.
<i>selected_id</i>	ProSelectionModelitemGet() , then read <i>modelitem.id</i> .
<i>selected_ptr</i>	ProSelectionModelitemGet() , then Pro*Init() , depending on the type.
<i>select_pnt</i>	ProSelectionPoint3dGet() .
<i>sel_param</i>	ProSelectionUvParamGet() .
<i>sel_depth</i>	ProSelectionDepthGet() .
<i>part_ptr</i>	ProSelectionAsmcomppathGet() , and ProAsmcomppathMdlGet() .
<i>assembly_ptr</i>	ProSelectionAsmcomppathGet() , then read <i>comppath.owner</i> .
<i>memb_num</i>	ProSelectionAsmcomppathGet() , then read <i>comppath.table_num</i> .
<i>memb_id_tab</i>	ProSelectionAsmcomppathGet() , then read <i>comppath.comp_id_tab</i> .
<i>view_ptr</i>	ProSelectionViewGet() .

Note: The values of the *Select3d* field *sel_type* do not map directly to values of *ProType*, used in *ProModelitem*. Do not convert these types by direct assignment. See the next table for the mapping.

This table also makes clear what data from Pro/DEVELOP you need to build a *ProSelection* object in Pro/TOOLKIT, using the functions **ProSelectionAlloc()**, **ProSelectionSet()**, **ProSelectionUvParamSet()**, and **ProSelectionVerify()**.

There was an anomaly in selecting datum points in Pro/DEVELOP that has been corrected in Pro/TOOLKIT. The *selected_id* is the identifier of the feature, and the datum point identifier is given by the field *sel_elem_id*. However, this does not carry over into Pro/TOOLKIT—the *ProModelitem* identifier is the identifier of the datum point. (To get the feature, use the function **ProGeomitemFeatureGet()**.)

The following table shows how the values of the *sel_type* field in *Select3d* (and the corresponding **pro_select()** option strings) map to values of *ProType* used in Pro/TOOLKIT for *ProModelitem* and *ProGeomitem* objects used in Pro/TOOLKIT as object handles that inherit from *ProModelitem*, such as *ProGeomitem*, *ProDimension*, *ProGtol*, *ProDtlnote*, and so forth.

pro_select() option	Select3d sel_type	ProType
point	SEL_3D_PNT	PRO_POINT
axis	SEL_3D_AXIS	PRO_AXIS
datum	SEL_3D_SRF	PRO_SURFACE
csys	SEL_3D_CSYS	PRO_CSYS
feature	SEL_3D_FEAT	PRO_FEATURE
edge	SEL_3D_EDG	PRO_EDGE
edge_end	SEL_3D_VERT	PRO_EDGE_START or PRO_EDGE_END
curve	SEL_3D_CURVE	PRO_CURVE
curve_end	SEL_CURVE_END	PRO_CRV_START or PRO_CRV_END
sldedge	SEL_3D_EDGE	PRO_EDGE
qltedge	SEL_3D_EDGE	PRO_EDGE
surface	SEL_3D_SRF	PRO_SURFACE
sldface	SEL_3D_SRF	PRO_SURFACE
qltface	SEL_3D_SRF	PRO_SURFACE
dtmqlt	SEL_3D_SRF_LIST	PRO_QUILT
part	SEL_3D_PART	PRO_PART
prt_or_asm	SEL_3D_PART	PRO_PART or PRO_ASSEMBLY
dimension	DTL_DIM	PRO_DIMENSION
ref_dim	DTL_REFDIM	PRO_REF_DIMENSION

pro_select() option	Select3d sel_type	ProType
gtol	DTL_GTOL	PRO_GTOL
dtl_symbol	DTL_SYMBOL	PRO_SYMBOL_INSTANCE
dwg_table	SEL_DWG_TABLE	PRO_DRAW_TABL
any_note	DTL_USER_NOTE	PRO_NOTE
note_3d	SEL_3D_NOTE	PRO_NOTE
dgm_obj	SEL_DGM_REF_OBJ	PRO_DIAGRAM_OBJECT
dgm_non_cable_wire	DTL_WIRE	PRO_DIAGRAM_WIRE
draft_ent	DTL_DRAFT_ENT	PRO_DRAFT_ENTITY
ext_obj	SEL_3D_EXT_OBJ	PRO_EXTOBJ
table_cell	SEL_TABLE_CELL	PRO_DRAW_TABLE_CELL

To minimize the need to convert between *Select3d* and *ProSelection*, follow these guidelines:

- If you need a *Select3d* (defined in section *Select3d Structure Definition*) as an input to a Pro/DEVELOP function, and the element referred to is to be selected interactively, use one of the Pro/DEVELOP select functions listed below instead of **ProSelect()** until you can convert the whole application.

pro_select()
pro_get_selection()
pro_set_and_get_selection()

- If you need to use a Pro/DEVELOP function whose output is, or contains, a *Select3d* (defined in section *Select3d Structure Definition*), try to process the output using Pro/DEVELOP functions instead of converting to a *ProSelection* object where possible. For example, use **pro_show_select()** instead of **ProSelectionHighlight()**.

To help you maintain such mixtures, the online browser retains the description of the Pro/DEVELOP selection functions and *Select3d* that are, strictly speaking, superseded by Pro/TOOLKIT.

However, PTC recommends that you retain your documentation for Pro/DEVELOP to use with Pro/TOOLKIT.

Equivalent Pro/DEVELOP Functions

The following table lists the Pro/DEVELOP functions that have equivalents in Pro/TOOLKIT. If the Pro/DEVELOP function is not included in this list, the function retains the Pro/DEVELOP style in Pro/TOOLKIT. For ease of use, the Pro/DEVELOP functions are presented by functional group.

Pro/DEVELOP Function	Equivalent Pro/TOOLKIT Function
Core Functions	
<i>Synchronous Mode</i>	
pro_term()	ProEngineerEnd()
<i>Asynchronous Mode</i>	
prodev_start_proengineer()	ProEngineerStart()
prodev_set_interrupt_func()	Not supported in Pro/TOOLKIT. It can be safely removed from applications which call it. For information on how to structure a full asynchronous mode application to accept Pro/ENGINEER events, refer to the chapter Core: Asynchronous Mode.
prodev_handle_interrupt()	ProEventProcess()
prodev_set_proe_term_func()	ProTermFuncSet()
user_proe_term_func()	ProTerminationAction()
prodev_get_proe_status()	ProEngineerStatusGet()
<i>User-Supplied Main</i>	
prodev_main()	ProToolkitMain()
Menus	
<i>Adding a Menu Button</i>	
promenu_create()	ProMenuFileRegister()
promenu_expand()	ProMenuAuxfileRegister()
promenu_on_button()	ProMenubuttonActionSet()
promenu_load_action()	ProMenubuttonGenactionSet()
<i>New Menus</i>	
promenu_action()	ProMenuProcess()
promenu_exit_up()	ProMenuDelete()
promenu_make()	ProMenuCreate()

Pro/DEVELOP Function	Equivalent Pro/TOOLKIT Function
promenu_no_exit()	ProMenuHold()
promenu_exit_action_up()	ProMenuDeleteWithStatus()
promenu_make_compound()	ProCompoundmenuCreate()
<i>Preempting Pro/ENGINEER Commands</i>	
promenu_load_pre_func()	ProMenubuttonPreactionSet()
promenu_load_post_func()	ProMenubuttonPostactionSet()
<i>Manipulating Menus</i>	
promenu_set_item_location()	ProMenubuttonLocationSet()
promenu_set_item_visible()	PropMenubuttonVisibilitySet()
promenu_remove_item()	ProMenubuttonDelete()
<i>Data Menus</i>	
promenu_set_mode()	ProMenuModeSet()
promenu_set_data_mode()	ProMenuDatamodeSet()
<i>Setting Menu Buttons</i>	
promenu_set_item()	ProMenubuttonHighlight()
promenu_reset_item()	ProMenubuttonUnhighlight()
<i>Controlling Accessibility of Menu Buttons</i>	
promenu_make_item_accessible()	ProMenubuttonActivate()
promenu_make_item_inaccessible()	ProMenubuttonDeactivate()
<i>Pushing and Popping Menus</i>	
promenu_is_up()	ProMenuVisibilityGet()
promenu_push()	ProMenuPush()
promenu_pop()	ProMenuPop()
<i>Run-Time Menus</i>	
pro_select_strings()	ProMenuStringsSelect()
<i>Entering Pro/ENGINEER Commands</i>	
proload_cmd_sequence()	ProMacroLoad()
promenu_push_command()	ProMenuCommandPush()
Message Window	
<i>Writing a Message</i>	
promsg_print()	ProMessageDisplay()
promsg_clear()	ProMessageClear()

Pro/DEVELOP Function	Equivalent Pro/TOOLKIT Function
<i>Writing a Message to an Internal Buffer</i>	
promsg_sprint()	ProMessageToBuffer()
<i>Getting Keyboard Input</i>	
promsg_getint()	ProMessageIntegerRead()
promsg_getdouble()	ProMessageDoubleRead()
promsg_getstring()	ProMessageStringRead()
promsg_getpasswd()	ProMessagePasswordRead()
Graphics and Object Display	
<i>Manipulating Windows</i>	
pro_clear_window()	ProWindowClear()
pro_view_repaint()	ProWindowRepaint()
pro_refresh_window()	ProWindowRefresh()
pro_get_current_window()	ProWindowCurrentGet()
pro_set_current_window()	ProWindowCurrentSet()
pro_open_object_window()	ProObjectwindowCreate()
pro_close_object_window()	ProWindowDelete()
<i>Model Orientation</i>	
pro_get_cur_window_matrix()	ProWindowCurrentMatrixGet()
pro_get_view_matrix()	ProViewMatrixGet()
pro_set_view_matrix()	ProViewMatrixSet()
pro_reset_view()	ProViewReset()
pro_rotate_view()	ProViewRotate()
pro_store_view()	ProViewStore()
pro_retrieve_view()	ProViewRetrieve()
pro_get_view_names()	ProViewNamesGet()
<i>Displaying Pro/ENGINEER Objects</i>	
progr_display_object()	ProSolidDisplay() (for parts and assemblies) ProMdlDisplay()
pro_show_select()	ProSelectionHighlight() ProSelectionDisplay() ProSelectionUnhighlight()
<i>Graphics Colors and Line Styles</i>	

Pro/DEVELOP Function	Equivalent Pro/TOOLKIT Function
progr_text_color()	ProTextColorSet()
progr_color()	ProGraphicsColorSet()
progr_get_color_map()	ProColormapGet()
progr_set_color_map()	ProColormapSet()
progr_set_line_style()	ProLinestyleSet()
progr_get_line_style_def()	ProLinestyleDataGet()
<i>Displaying Graphics</i>	
progr_move_3d()	ProGraphicsPenPosition()
progr_draw_3d()	ProGraphicsLineDraw()
progr_put_polyline()	ProGraphicsPolylineDraw()
progr_put_multi_polylines()	ProGraphicsMultiPolylinesDraw()
progr_put_arc()	ProGraphicsArcDraw()
progr_put_circle()	ProGraphicsCircleDraw()
progr_draw_polygon_2d()	ProGraphicsPolygonDraw()
<i>Displaying Text</i>	
progr_put_text()	ProGraphicsTextDisplay()
pro_get_text_attributes()	ProCurrentTextAttributesGet()
pro_set_text_attributes()	ProCurrentTextAttributesSet()
progr_get_default_font_id()	ProTextfontDefaultIdGet()
progr_get_font_id()	ProTextfontIdGet()
progr_get_font_name()	ProTextFontNameGet()
<i>Getting Mouse Input</i>	
promenu_get_pick()	ProMousePickGet()
pro_sample_xy()	ProMouseTrack()
progr_set_draw_mode()	ProGraphicsModeSet()
pro_getbox()	ProMouseBoxInput()
<i>Display Lists</i>	
pro_create_2d_disp_list()	ProDisplist2dCreate()
pro_display_2d_disp_list()	ProDisplist2dDisplay()
pro_delete_2d_disp_list()	ProDisplist2dDelete()
pro_create_3d_disp_list()	ProDisplist3dCreate()
pro_display_3d_disp_list()	ProDisplist3dDisplay()

Pro/DEVELOP Function	Equivalent Pro/TOOLKIT Function
pro_delete_3d_disp_list()	ProDisplist3dDelete()
<i>Layers</i>	
prolayer_get_names()	ProMdlLayerNamesGet()
prolayer_add_item()	ProLayerItemAdd()
prolayer_remove_item()	ProLayerItemRemove()
prolayer_display()	ProLayerDisplaystatusSet()
prolayer_get_display()	ProLayerDisplaystatusGet()
prolayer_create_layer()	ProLayerCreate()
prolayer_delete_layer()	ProLayerDelete()
prolayer_get_items()	ProLayerItemsGet()
Database Support	
<i>Session Objects</i>	
pro_get_current_object()	ProMdlCurrentGet()
pro_get_current_mode()	ProModeCurrentGet()
prodb_find_declared_objects()	ProMdlDeclaredList()
prodb_find_nobject_depend()	ProMdlDependenciesList()
prodb_get_object_info()	ProMdldataGet()
prodb_get_object_ptr()	ProMdlInit()
prodb_first_name_in_list()	ProSessionMdlList()
prodb_next_name_in_list()	ProSessionMdlList()
prodb_was_object_modified()	ProMdlModificationVerify()
pro_solid_id_from_ptr()	ProSolidToPostfixId()
pro_solid_ptr_from_id()	ProPostfixIdToSolid()
<i>File Management Operations</i>	
prodb_create_object()	ProSolidCreate()
prodb_create_obj()	ProSolidCreate()
prodb_retrieve_object()	ProMdlRetrieve()
prodb_save_object()	ProMdlSave()
prodb_rename_object()	ProMdlRename()
prodb_copy_object()	ProMdlCopy()
prodb_erase_object()	ProMdlErase()
prodb_backup_object()	ProMdlBackup()

Pro/DEVELOP Function	Equivalent Pro/TOOLKIT Function
<i>Simplified Representations</i>	
prodb_get_simplfd_rep_info()	ProSimprepActiveGet()
prodb_get_simplfd_rep_list()	ProSolidSimprepVisit()
prodb_retrieve_simplfd_rep()	ProAsmSimprepRetrieve()
<i>Selecting Objects</i>	
pro_select()	ProSelect()
pro_get_selection()	ProSelect()
pro_set_and_get_selection()	ProSelect()
<i>Tracing a Ray Through a Model</i>	
pro_ray_x_model()	ProSolidRayIntersectionCompute()
<i>Element Information</i>	
pro_element_info()	PRO_IDENTIFY: Pro*IdGet(), Pro*Init() PRO_BELONG_TO: ProGeomitemFeatureGet()
<i>Regenerating Models</i>	
pro_regenerate()	ProSolidRegenerate()
pro_regenerate_object()	ProSolidRegenerate()
<i>Part Accuracy</i>	
prodb_get_model_accuracy()	ProSolidAccuracyGet()
prodb_set_model_accuracy()	ProSolidAccuracySet()
<i>Mass Properties</i>	
prodb_mass_prop()	ProSolidMassPropertyGet()
<i>Utilities</i>	
pro_str_to_wstr()	ProStringToWstring()
pro_wstr_to_str()	ProWstringToString()
pro_wchar_t_check()	ProWcharSizeVerify()
pro_show_file()	ProInfoWindowDisplay()
pro_show_info_window()	ProInfoWindowDisplay()
prodb_edit_file()	ProFileEdit()
<i>Session Tools</i>	
pro_get_prodevdat_info()	ProToolkitApplExecPathGet() ProToolkitApplTextPathGet()

Pro/DEVELOP Function	Equivalent Pro/TOOLKIT Function
pro_getenvironment()	ProConfigoptGet()
pro_change_dir()	ProDirectoryChange()
pro_get_current_directory()	ProDirectoryCurrentGet()
pro_is_option_ordered()	ProOptionOrderedVerify()
pro_get_config()	ProConfigoptGet()
pro_set_config()	ProConfigoptSet()
progr_invalidate_display_list()	ProDisplistInvalidate()
<i>Exporting and Importing Files</i>	
pro_export_file_from_pro()	ProOutputFileWrite()
pro_export_plot_file()	ProPlotfileWrite()
pro_read_file_to_pro()	ProInputFileRead()
pro_export_fea_mesh()	ProFemmeshExport()
<i>Material Names</i>	
prodb_get_material_props()	ProPartMaterialdataGet()
prodb_get_material_name()	ProPartMaterialNameGet()
prodb_set_material_name()	ProPartMaterialSet()
<i>Storing Generic Data</i>	
proappdata_register_class()	ProExtdataClassRegister()
proappdata_create_data()	ProExtdataSlotCreate()
proappdata_write_data()	ProExtdataSlotWrite()
proappdata_read_data()	ProExtdataSlotRead()
proappdata_delete_data()	ProExtdataSlotDelete()
proappdata_list_classes()	ProExtdataClassNamesList()
proappdata_list_data_in_class()	ProExtdataSlotIdsList()
Geometry	
<i>Traversing the Geometry of a Part</i>	
prodb_first_part_face()	ProSolidSurfaceVisit()
prodb_next_part_face()	ProSolidSurfaceVisit()
prodb_get_solid_surfaces()	ProSolidSurfaceVisit()
prodb_first_face_contour()	ProSurfaceContourVisit()
prodb_next_face_contour()	ProSolidSurfaceVisit()
prodb_first_cntr_edge()	ProContourEdgeVisit()

Pro/DEVELOP Function	Equivalent Pro/TOOLKIT Function
prodb_next_cntr_edge()	ProContourEdgeVisit()
prodb_get_datum_curves()	ProSolidFeatVisit(), ProFeatureGeomitemVisit()
prodb_get_datum_surfaces()	ProSolidQuiltVisit(), ProQuiltSurfaceVisit()
prodb_edge_data()	ProEdgeNeighborsGet()
prodb_edge_direction()	ProEdgeDirGet()
prodb_contour_traversal()	ProContourTraversalGet()
prodb_containing_contour()	ProContainingContourFind()
prodb_vertex_data()	ProEdgeVertexdataGet()
prodb_get_solid_volumes()	ProSldsurfaceVolumesFind()
<i>Evaluating Geometry</i>	
prodb_edge_tessellation()	ProEdgeTessellationGet()
pro_eval_xyz_edge()	ProEdgeXyzdataEval()
pro_eval_xyz_entity()	ProCurveXyzdataEval()
pro_eval_uv_edge()	ProEdgeUvdataEval()
pro_eval_xyz_face()	ProSurfaceXyzdataEval()
pro_get_edge_param()	ProEdgeParamEval()
pro_get_entity_param()	ProCurveParamEval()
pro_get_face_params()	ProSurfaceParamEval()
prodb_uv_in_face_domain()	ProSurfaceUvpntVerify()
pro_point_on_geom()	ProGeometryAtPointFind()
prodb_get_edge_uv_points()	ProEdgeTessellationGet()
prodb_get_surface_tessellation()	ProSurfaceTessellationGet()
<i>Geometry Equations</i>	
prodb_get_curve_type()	ProCurveTypeGet()
pro_get_curve_type_geom()	ProCurvedataGet() ProAxisDataGet()
prodb_get_edge_type()	ProEdgeTypeGet()
prodb_get_edge_curve()	ProEdgedataGet()
prodb_rls_edge_curve()	ProGeomitemdataFree()
prodb_get_face_type()	ProSurfaceTypeGet()
prodb_get_surface()	ProSurfacedataGet()

Pro/DEVELOP Function	Equivalent Pro/TOOLKIT Function
prodb_rls_surface()	ProGeomitemdataFree()
prodb_surface_to_nurbs()	ProSurfaceToNURBS()
prodb_edge_to_nurbs()	ProEdgeToNURBS()
prodb_entity_to_nurbs()	ProCurveToNURBS()
<i>Measurement</i>	
pro_edge_length()	ProEdgeLengthEval()
pro_face_area()	ProSurfaceAreaEval()
prodb_measure()	ProSurfaceDiameterEval(), ProGeomitemAngleEval(), ProGeomitemDistanceEval()
pro_face_extremes()	ProSurfaceExtremesEval()
prodb_get_envelope()	ProSolidOutlineGet()
prodb_compute_outline()	ProSolidOutlineCompute()
Parameters and Dimensions	
<i>Parameters</i>	
prodb_get_parameters()	ProParameterVisit() and ProParameterValueGet()
prodb_set_parameters()	ProParameterValueSet()
prodb_add_parameters()	ProParameterCreate()
prodb_delete_parameters()	ProParameterDelete()
prodb_reset_parameters()	ProParameterValueReset()
prodb_designate_param()	ProParameterDesignationAdd() ProParameterDesignationRemove() ProParameterDesignationVerify()
Features	
<i>Listing Features</i>	
prodb_get_feature_ids()	ProSolidFeatVisit()
prodb_get_feat_parent_child()	ProFeatureParentsGet() ProFeatureChildrenGet()
prodb_get_feat_type()	ProFeatureTypeGet()
<i>Names of Features and Other Elements</i>	
prodb_get_element_name()	ProModelitemNameGet()
prodb_set_element_name()	ProModelitemNameSet()
<i>Feature Geometry</i>	

Pro/DEVELOP Function	Equivalent Pro/TOOLKIT Function
prodb_get_feature_surfaces()	ProFeatureGeomitemVisit()
prodb_get_surface_feature()	ProGeomitemFeatureGet()
<i>Manipulating Features</i>	
prodb_suppress_feature()	ProFeatureSuppress() ProSolidFeatstatusSet()
prodb_resume_feature()	ProFeatureResume() ProSolidFeatstatusSet()
prodb_delete_feature()	ProFeatureDelete()
<i>Datum Planes</i>	
prodb_first_datum()	ProSolidFeatVisit() ProFeatureSurfaceVisit()
prodb_next_datum()	ProSolidFeatVisit() ProFeatureSurfaceVisit()
<i>Datum Points</i>	
prodb_get_feature_dtm_points()	ProSolidFeatVisit() ProFeatureGeomitemVisit()
prodb_eval_xyz_dtm_point()	ProPointCoordGet()
<i>Axes</i>	
prodb_get_first_axis()	ProSolidAxisVisit() ProGeomitemFeatureGet()
prodb_get_next_axis()	ProSolidAxisVisit() ProGeomitemFeatureGet()
<i>Coordinate Systems</i>	
prodb_find_csys()	ProSolidCsysVisit() ProGeomitemFeatureGet(),
prodb_unpack_csys()	ProCsysdataGet()
<i>Surface Quilts</i>	
prodb_get_surface_quilt()	ProSurfaceQuiltGet()
Assemblies	
<i>Finding Assembly Members</i>	
prodb_first_member()	ProSolidFeatVisit() ProFeatureTypeGet()
prodb_next_member()	ProSolidFeatVisit() ProFeatureTypeGet()
prodb_member_to_object()	ProAsmcompMdlGet()

Pro/DEVELOP Function	Equivalent Pro/TOOLKIT Function
prodb_obj_from_assem_rel()	ProPostfixIdToSolid()
<i>Location of Assembly Members</i>	
prodb_member_transform()	ProAsmcomppathTrfGet()
pro_vectors_to_transf()	ProMatrixInit()
prodb_get_assembly_transform()	ProAsmcomppathTrfGet()
prodb_set_member_transform()	ProAsmcomppathTrfSet()
<i>Assembling and Deleting Members</i>	
prodb_assemble_by_transform()	ProFeatureCreate()
prodb_assemble_component()	ProFeatureCreate()
prodb_explode_assembly()	ProAssemblyExplode() ProAssemblyUnexplode()
prodb_get_assembly_constraints()	ProFeatureElemtreeCreate()
prodb_is_assembly_exploded()	ProAssemblyIsExploded()
Notify	
<i>Basic Notify Functions</i>	
prodev_notify()	ProNotificationSet() ProNotificationUnset()
pro_on_regenerate_end()	ProNotificationSet() ProNotificationUnset()
<i>Notify for File Management Operations</i>	
user_menu_dbms_save_pre()	ProMdlSavePreAction()
user_menu_dbms_save_post()	ProMdlSavePostAction()
user_menu_dbms_copy_pre()	ProMdlCopyPreAction()
user_menu_dbms_copy_post()	ProMdlCopyPostAction()
user_menu_dbms_rename_pre()	ProMdlRenamePreAction()
user_menu_dbms_rename_post()	ProMdlRenamePostAction()
user_menu_dbms_erase_pre()	ProMdlErasePreAction()
user_menu_dbms_erase_post()	ProMdlErasePostAction()
user_menu_dbms_purge_pre()	ProMdlPurgePreAction()
user_menu_dbms_purge_post()	ProMdlPurgePostAction()
user_menu_dbms_delete_pre()	ProMdlDeletePreAction()
user_menu_dbms_delete_post()	ProMdlDeletePostAction()
user_menu_dbms_create_pre()	ProMdlCreatePreAction()

Pro/DEVELOP Function	Equivalent Pro/TOOLKIT Function
user_menu_dbms_create_post()	ProMdlCreatePostAction()
user_menu_dbms_retrieve_pre()	ProMdlRetrievePreAction()
user_menu_dbms_retrieve_post()	ProMdlRetrievePostAction()
user_dbms_save_post_all()	ProMdlSavePostAllAction()
user_dbms_copy_post_all()	ProMdlCopyPostAllAction()
user_dbms_erase_post_all()	ProMdlErasePostAllAction()
user_dbms_delete_post_all()	ProMdlDeletePostAllAction()
user_dbms_retrieve_post_all()	ProMdlRetrievePostAllAction()
<i>Failure Notify</i>	
user_dbms_failure_function()	ProMdlDbmsFailureAction()
<i>Change Notify</i>	
user_change_window_post()	ProWindowChangePostAction()
user_change_directory()	ProDirectoryChangePostAction()
<i>Graphics Notify</i>	
user_graphics_object_output_pre()	ProMdlDisplayPreAction()
user_graphics_object_output_post()	ProMdlDisplayPostAction()
<i>Manufacturing Notify</i>	
user_mfg_oper_cl_file_post()	ProMfgoperClPostAction()
user_mfg_feat_cl_file_post()	ProNcseqClPostAction()
Manufacturing Operations	
<i>Manufacturing Components</i>	
prodb_get_comp_role()	ProMfgAssemGet() ProMfgSolidGet() ProMfgFeatureOwnerGet()
<i>Manufacturing Parameters</i>	
promfg_get_tool_ids()	ProMfgToolVisit()
promfg_get_tool_parameters()	ProToolParamGet()
promfg_set_tool_parameters()	ProToolElemParamAdd()
promfg_get_nc_type()	ProNcseqTypeGet()
Cabling Operations	
prodbl_create_cables_from_logical()	ProCablesFromLogicalCreate()
prodbl_get_cables_from_logical()	ProCablesFromLogicalGet()

Pro/DEVELOP Function	Equivalent Pro/TOOLKIT Function
prodbl_get_connectors_from_logical()	ProConnectorsFromLogicalGet()
prodbl_get_assy_spools()	ProAssemblySpoolsCollect()
prodbl_create_spool()	ProSpoolCreate()
prodbl_get_spool_params()	ProSpoolParametersCollect()
prodbl_get_spool_param()	ProSpoolParameterGet()
prodbl_get_assy_connectors()	ProAssemblyConnectorsGet()
prodbl_designate_connector()	ProConnectorDesignate()
prodbl_undesignate_connector()	ProConnectorUndesignate()
prodbl_get_connector_entry_ports()	ProConnectorEntryPortsGet()
prodbl_set_spool_params()	ProSpoolParametersSet()
prodbl_delete_spool_param()	ProSpoolParameterDelete()
prodbl_get_spools_from_logical()	ProSpoolsFromLogicalGet()
prodbl_get_spools_from_logical()	ProSpoolsFromLogicalGet()
prodbl_get_connector()	ProConnectorParamGet()
prodbl_delete_connector_param()	ProConnectorParamDelete()
prodbl_get_connector_params()	ProConnectorParamsCollect()
prodbl_set_connector_params()	ProConnectorParamsSet()
prodbl_get_assy_harnesses()	ProAssemblyHarnessesCollect()
prodbl_create_harness()	ProHarnessCreate()
prodbl_get_harness_cables() prodbl_get_cable_id()	ProHarnessCablesCollect()
prodbl_get_harness_locations()	ProHarnessLocationsCollect()
prodbl_get_wire_harnesses()	ProCableHarnessesGet()
prodbl_get_wire_length()	ProCableLengthGet()
prodbl_get_cable_id()	ProCableByNameGet()
prodbl_get_cable_name()	ProCableNameGet()
prodbl_get_cable_subtype()	ProCableTypeGet()
prodbl_create_cable()	ProCableCreate()
prodbl_create_bundle()	ProBundleCreate()
prodbl_get_bundle_cables()	ProBundleCablesCollect()
prodbl_get_cable_logical_ends()	ProCableLogicalEndsGet()
prodbl_get_cable_geom()	ProCableIsComplete()

Pro/DEVELOP Function	Equivalent Pro/TOOLKIT Function
prodbl_set_cable_params()	ProCableParametersSet()
prodbl_delete_cable_param()	ProCableParameterDelete()
prodbl_get_cable_param()	ProCableParameterGet()
prodbl_get_cable_params()	ProCableParametersCollect()
prodbl_cable_clearance()	ProCableClearanceCompute()
prodbl_get_cable_geom()	ProCableSegmentsGet()
prodbl_get_cable_geom()	ProCablesegmentPointsGet()
prodbl_get_cable_geom()	ProCablesegmentIsInBundle()
prodbl_get_cable_geom()	ProCablesegmentIsNew()
prodbl_get_cable_locations()	ProCableLocationsCollect()
prodbl_get_location_cables() prodbl_get_cable_id()	ProCablelocationCablesGet()
prodbl_get_location_type()	ProCablelocationTypeGet()
prodbl_get_location_pnt()	ProCablelocationPointGet()
prodbl_routing_start()	ProCableRoutingStart()
prodbl_route_thru_location()	ProCableThruLocationRoute()
prodbl_routing_end()	ProCableRoutingEnd()
UDF Functions	
proddb_first_udf()	ProSolidGroupVisit() or ProSolidGroupsCollect()
proddb_next_udf()	ProSolidGroupVisit() or ProSolidGroupsCollect()
proddb_get_group()	ProFeatureGroupGet() plus ProGroupFeatureVisit() or ProGroupFeaturesCollect() plus ProGroupIsTabledriven()
proddb_get_udf_name()	ProUdfNameGet()
proddb_first_dim_udf()	ProUdfDimensionVisit() or ProUdfDimensionsCollect()
proddb_next_dim_udf()	ProUdfDimensionVisit() or ProUdfDimensionsCollect()
proddb_get_udf_dim_name()	ProUdfDimensionNameGet()
proddb_place_udf()	Replaced
proddb_create_group()	ProUdfCreate()

Pro/DEVELOP Function	Equivalent Pro/TOOLKIT Function
proddb_get_udf_instance_name()	ProUdfNameGet()

Data Structures for Pro/DEVELOP-Style Functions

If you are using Pro/DEVELOP-style functions in your Pro/TOOLKIT application, you can include the file *prodevelop.h* before *ProToolkit.h*.

The data structures used by the Pro/DEVELOP-style functions are as follows:

- Prodtl_item Structure Definition on page I - 30
- Ptc_surf Structure Definition on page I - 33
- Select3d Structure Definition on page I - 38

Prodtl_item Structure Definition

```
typedef struct prodtl_item
{
    int    type; /* PRODTL_ENTITY, PRODTL_NOTE, PRODTL_SYM_DEFINITION,
                  PRODTL_SYM_INSTANCE, or PRODTL_DRAFT_GROUP */
    int    id;
} Prodtl_item;

typedef struct prodtl_entity
{
    int          type; /* PRODTL_ENTITY */
    int          id;
    Ptc_curve    curve;
    int          attributes; /* PRODTL_ENTITY_CONSTRUCTION
                             or PRODTL_ENTITY_HIDDEN */
    Ptc_color    color; /* A standard color, or
                        USER_RGB_COLOR */
    wchar_t      font[PRODTL_NAME_LEN]; /* A null-terminated string */
    double       width; /* Width, in world
                        coordinates */
    int          view_id;
} Prodtl_entity;

typedef union ptc_curve
{
    Ptc_line     line;
    Ptc_arc      arc;
}
```

```

Ptc_spline    spline;
Ptc_b_spline  b_spline;
Ptc_ellipse   ellipse;
Ptc_point     point;
Ptc_polygon   polygon;
} Ptc_curve;
typedef struct ptc_line
{
    int         type;
    double      end1[3];
    double      end2[3];
} Ptc_line;

typedef struct ptc_arc
{
    int         type;
    double      vector1[3]; /* first vector of the arc coordinate
                           system */
    double      vector2[3]; /* second vector of the arc coordinate
                           system */
    double      origin[3];  /* center of the arc coordinate system */
    double      start_angle; /* starting angle (in radians) of the
                           arc. */
    double      end_angle;  /* end angle (in radians) of the arc */
    double      radius;     /* radius of the arc */
} Ptc_arc;

typedef struct ptc_spline
{
    int         type;
    double      *par_arr;    /* array of spline parameters */
    Ptc_pnt     *pnt_arr;    /* array of spline interpolant
                           points */
    Ptc_pnt     *tan_arr;    /* array of tangent vectors at each
                           point */
    int         num_points;  /* size for all arrays */
} Ptc_spline;

typedef double Ptc_pnt[3]; /* 3D point */

typedef struct ptc_b_spline
{
    int         type;
    int         degree;      /* Basis functions degree. */
    double      *params;     /* Expandable array of knots on the
                           parameter line */
    double      *weights;    /* For rational B-splines, the
                           expandable array of the same
                           dimension as c_pts. Otherwise,
                           NULL. */
    Ptc_pnt     *c_pts;     /* Expandable array of control

```

```

                                points. */
    int      num_knots;      /* The size of the parameters
                                array. */
    int      num_c_points;  /* The size of c_pnts and weights (if
                                not NULL) */
} Ptc_b_spline;

typedef struct ptc_ellipse
{
    int      type;
    Ptc_pnt  center;
    Ptc_pnt  major_axis_unit_vect;
    Ptc_pnt  norm_axis_unit_vect;
    double   major_len;
    double   minor_len;
    double   start_ang;
    double   end_ang;
} Ptc_ellipse;

typedef struct ptc_point
{
    int      type;
    double   position[3];
} Ptc_point;

typedef struct ptc_polygon
{
    int      type;
    int      fill_color;
    Ptc_pnt  *vertices;
    int      num_vertices;
} Ptc_polygon;

typedef struct ptc_color
{
    int      type;
    int      color;
    double   rgb[3];
}

```

Data Structure Constants

Data Structure Constants	Value	Description
PTC_COLOR_DEFAULT	0	Use the default color.
PTC_COLOR_RGB	1	Specify the red, green, and blue color values.
PTC_COLOR_INDEX	2	Specify an index into the color table.

Ptc_surf Structure Definition

```
typedef struct ptc_surf
{
    int            type;           /* surface type */
    double         uv_min[2];      /* minimum uv extents */
    double         uv_max[2];      /* maximum uv extents */
    double         xyz_min[3];     /* minimum XYZ extents */
    double         xyz_max[3];     /* maximum XYZ extents */
    int            orient;         /* 1 if du x dv points outside,
                                   else -1 */
    Ptc_srfshape   srf_shape;      /* the surface shape */
    int            user_int[4];    /* space for user integer data */
    Prohandle      user_ptr[4];    /* space for user pointer data */
} Ptc_surf;

typedef union ptc_srfshape
{
    Ptc_plane      plane;
    Ptc_cylinder   cylinder;
    Ptc_cone       cone;
    Ptc_torus      torus;
    Ptc_srfrev     srfrev;
    Ptc_tabcyl     tabcyl;
    Ptc_rulsrf     rulsrf;
    Ptc_coons      coons;
    Ptc_filsrf     filsrf;
    Ptc_splsrfs    spl_srf;
    Ptc_b_splsrfs  b_spl_srf;
    Ptc_cyl_splsrfs cyl_splsrfs;
    Ptc_frgnsrf    frgnsrf;
} Ptc_srfshape;

typedef struct Ptc_plane      /* the local coordinate system */
{
    double    e1[3];
    double    e2[3];
    double    e3[3];
    double    origin[3];
} Ptc_plane;

typedef struct ptc_cylinder
{
    double    e1[3];
    double    e2[3];
    double    e3[3];
    double    origin[3];
    double    radius;
} Ptc_cylinder;

typedef struct ptc_cone
```

```

{
    double    e1[3];
    double    e2[3];
    double    e3[3];
    double    origin[3];
    double    alpha;
} Ptc_cone;

typedef struct ptc_torus
{
    double    e1[3];
    double    e2[3];
    double    e3[3];
    double    origin[3];
    double    radius1;
    double    radius2;
} Ptc_torus;

typedef struct ptc_srfrev
{
    double    e1[3];
    double    e2[3];
    double    e3[3];
    double    origin[3];
    Ptc_curve curve;
} Ptc_srfrev;

typedef union ptc_curve
{
    Ptc_line    line;
    Ptc_arc     arc;
    Ptc_spline  spline;
    Ptc_b_spline b_spline;
    Ptc_ellipse ellipse;
    Ptc_point   point;
    Ptc_polygon polygon;
} Ptc_curve;

typedef struct ptc_line
{
    int        type;
    double     end1[3];
    double     end2[3];
} Ptc_line;
typedef struct ptc_arc
{
    int        type;
    double     vector1[3]; /* first vector of the arc coordinate
                           system */
    double     vector2[3]; /* second vector of the arc coordinate
                           system */

```



```

    double  origin[3];    /* center of the arc coordinate system */
    double  start_angle; /* starting angle (in radians) of the
                           arc. */
    double  end_angle;    /* end angle (in radians) of the arc */
    double  radius;       /* radius of the arc */
} Ptc_arc;

typedef double Ptc_pnt[3]; /* 3D point */

typedef struct ptc_spline
{
    int      type;
    double   *par_arr;     /* array of spline parameters */
    Ptc_pnt  *pnt_arr;     /* array of spline interpolant
                           points */
    Ptc_pnt  *tan_arr;     /* array of tangent vectors at each
                           point */
    int      num_points;   /* size for all arrays */
} Ptc_spline;

typedef struct ptc_b_spline
{
    int      type;
    int      degree;      /* Basis functions degree. */
    double   *params;     /* Expandable array of knots on the
                           parameter line */
    double   *weights;     /* For rational B-splines, the
                           expandable array of the same
                           dimension as c_pts. Otherwise,
                           NULL. */
    Ptc_pnt  *c_pts;      /* Expandable array of control
                           points. */
    int      num_knots;    /* The size of the parameters
                           array. */
    int      num_c_points; /* The size of c_pts and weights (if
                           not NULL) */
} Ptc_b_spline;

typedef struct ptc_ellipse
{
    int      type;
    Ptc_pnt  center;
    Ptc_pnt  major_axis_unit_vect;
    Ptc_pnt  norm_axis_unit_vect;
    double   major_len;
    double   minor_len;
    double   start_ang;
    double   end_ang;
} Ptc_ellipse;

typedef struct ptc_point

```

```

    {
        int      type;
        double   position[3];
    } Ptc_point;

typedef struct ptc_polygon
{
    int      type;
    int      fill_color;
    Ptc_pnt  *vertices;
    int      num_vertices;
} Ptc_polygon;

typedef struct ptc_tabcyl
{
    double    e1[3];
    double    e2[3];
    double    e3[3];
    double    origin[3];
    Ptc_curve curve;
} Ptc_tabcyl;

typedef struct ptc_rulsrf
{
    double    e1[3];
    double    e2[3];
    double    e3[3];
    double    origin[3];
    Ptc_curve curve_1;
    Ptc_curve curve_2;
} Ptc_rulsrf;

typedef struct ptc_coons
{
    Ptc_curve le_curve;           /* u = 0 boundary */
    Ptc_curve ri_curve;           /* u = 1 boundary */
    Ptc_curve dn_curve;           /* v = 0 boundary */
    Ptc_curve up_curve;           /* v = 1 boundary */
    Ptc_pnt   point_matrix[2][2]; /* corner points */
    Ptc_pnt   uvder_matrix[2][2]; /* corner mixed derivatives */
} Ptc_coons;

typedef struct ptc_filsrf
{
    Ptc_spline pnt_spline; /* spline running along the u = 0
                           boundary */
    Ptc_spline ctr_spline; /* spline running along the axis */
    Ptc_spline tan_spline; /* spline of unit tangents along the
                           axis */
} Ptc_filsrf;

```

```

typedef struct ptc_splsrfr
{
    double    *u_par_arr; /* array of u parameter grid */
    double    *v_par_arr; /* array of v parameter grid */
    Ptc_pnt   *point_arr; /* array of interpolant points */
    Ptc_pnt   *u_tan_arr; /* array of u derivatives at the interpolant
                          points */
    Ptc_pnt   *v_tan_arr; /* array of v derivatives at the interpolant
                          points */
    Ptc_pnt   *uvder_arr; /* array of uv derivatives at the interpolant
                          points */
    int        num_u;      /* size in u */
    int        num_v;      /* size in v */
} Ptc_splsrfr
typedef struct ptc_b_splsrfr
{
    int        deg[2];     /* Basis functions degrees (in the
                          directions u and v). */
    double    *u_par_arr; /* Expandable array of knots on the
                          parameter line u. */
    double    *v_par_arr; /* Expandable array of knots on the
                          parameter line v. */
    double    *wghts;      /* In the case of rational B-splines, an
                          expandable array of the same dimension
                          as c_point_arr. Otherwise, NULL. */
    Ptc_pnt   *c_point_arr; /* Expandable array of control points. */
    int        num_u;      /* Size in u. */
    int        num_v;      /* Size in v. */
    int        num_c_point; /* Number of control points. */
} Ptc_b_splsrfr;

typedef struct ptc_gen_splsrfr
{
    double     e1[3];
    double     e2[3];
    double     e3[3];
    double     origin[3];
    Ptc_splsrfr splsrfr; /* in appropriate coordinates */
} Ptc_cyl_splsrfr, Ptc_sph_splsrfr;

typedef struct Ptc_frgrsrfr
{
    double     e1[3];      /* local coordinate system */
    double     e2[3];
    double     e3[3];
    double     origin[3];
    int        foreign_id; /* foreign_id returned from the function
                          user_init_surf() */
} Ptc_frgrsrfr;

typedef char * Prohandle;

```

Data Structure Constants

Data Structure Constant	Constant	Definition
Surface types	PTC_OTHER_SRF	Another surface
	PTC_PLANE	Plane
	PTC_CYLINDER	Cylinder
	PTC_CONE	Cone
	PTC_TORUS	Torus
	PTC_SRFREV	General surface of revolution
	PTC_TABCYL	Tabulated cylinder
	PTC_RULSRF	Ruled surface
	PTC_COONS	Coons patch
	PTC_FILSRF	Fillet surface
	PTC_SPLSRF	Spline surface
	PTC_B_SPLSRF	NURBS surface
	PTC_FRGNSRF	Foreign surface
	PTC_CYL_SPLSRF	Cylindrical spline surface
Line types	PRO_SOLIDFONT	Solid line
	PRO_DOTFONT	Dotted line
	PRO_CTRLFONT	Alternating long and short dashes
	PRO_PHANTOMFONT	Alternating long dash and two dots
Entity types	PTC_POINT	Point
	PTC_LINE	Line
	PTC_ARC	Arc
	PTC_SPLINE	Spline curve
	PTC_B_SPLINE	B-spline curve
	PTC_ELLIPSE	Ellipse
	PTC_POLYGON	Polygon

Select3d Structure Definition

Always initialize the Select3d structure using the *memset* function, as follows:

```
Select3d  sel;

memset (&sel, '\0', sizeof (Select3d));

sel.memb_id_tab[0] = -1;
mem_num = 0;
```

```

typedef struct select3d
{
    char      *part_ptr;                /* part that selected_ptr
                                         belongs to (NULL if
                                         irrelevant) */

    char      *assembly_ptr;            /* root assembly on which
                                         the part is located
                                         (NULL if
                                         irrelevant) */

    int        memb_id_tab[MAX_SELPATH_DEPTH]; /* path to the selected
                                         part through the
                                         assembly members */

    double     aux_depth;
    int        sel_elem_id;
    INT_FUNC   pre_filter;
    INT_FUNC   post_filter;
    char      *auxiliary;
    char      *p_object;                /* viewport host */
    int        memb_num;                /* number of members in the assembly
                                         path */

    int        sel_type;                /* type of the selected element */
    int        selected_id;             /* identifier of the selected element */
    char      *selected_ptr;            /* pointer to the selected element */
    double     select_pnt[3];           /* coordinates of the selected point in the
                                         member (or drawing) coordinate system*/

    double     sel_param[2];            /* uv parameters of selected point. For */
                                         /* tables: row and column number of selected*/
                                         /* table cell. */

    double     sel_depth;               /* selection depth in screen coordinates */
    char      *view_ptr;                /* pointer to the selected view */
    int        window_id;               /* selected window identifier */
    int        use_cache_node;
    int        node_type;               /*type of node picked in tree tool, or 0 */
    char      *cache_data;
    char      *ptr_exp;
}
Select3d;

typedef int (*INT_FUNC)(); /* pointer to the function returning
                           an integer */

```

Data Structure Constants

Constant	Description
SEL_3D_SRF	Surface
SEL_3D_EDG	Edge
SEL_3D_AXIS	Axis
SEL_3D_PNT	Point
SEL_3D_FEAT	Feature
SEL_3D_VERT	Vertex

SEL_3D_CSYS	Coordinate system
SEL_3D_SRF_LIST	Surface list
SEL_3D_CURVE	Curve
SEL_CURVE_END	The same as SEL_3D_VERT for curves
SEL_DWG_VIEW	Drawing view
SEL_3D_PART	Part
SEL_DWG_TABLE	Drawing table
SEL_3D_IPAR	Integer parameter
DTL_DIM	Dimension
DTL_GTOL	Geometric tolerance
DTL_SYMBOL	Symbol
DTL_USER_NOTE	User note
DTL_DRAFT_ENT	Draft entity
SEL_DGM_REF_OBJ	Single-view component, single-view fixed connector, or parametric connector

The following table lists the possible options for the selection functions and the resultant data.

option	sel_type	selected_ptr	selected_id
point	SEL_3D_PNT	point	feature The datum point ID is given by the <i>Select3d</i> field <i>sel_elem_id</i> .
axis	SEL_3D_AXIS	axis	axis
datum	SEL_3D_SRF	surface	surface
csys	SEL_3D_CSYS	entity	entity
feature	SEL_3D_FEAT	feature	feature
edge	SEL_3D_EDG	edge	edge
edge_end	SEL_3D_VERT	edge	edge
curve	SEL_3D_CURVE	entity	entity
curve_end	SEL_CURVE_END	entity	entity
sldedge	SEL_3D_EDG	edge	edge
qltedge	SEL_3D_EDG	edge	edge
surface	SEL_3D_SRF	surface	surface
sldface	SEL_3D_SRF	surface	surface
qltface	SEL_3D_SRF	datum surface	datum surface
dtmqlt	SEL_3D_SRF_LIST	quilt	quilt

option	sel_type	selected_ptr	selected_id
dimension	DTL_DIM	N/A	dimension
ipar	SEL_3D_IPAR	N/A	parameter
gtol	DTL_GTOL	N/A	gtol
dtl_symbol	DTL_SYMBOL	N/A	symbol instance
dwg_table	SEL_DWG_TABLE	N/A	table
any_note	DTL_USER_NOTE	N/A	note
dwg_view	SEL_DWG_VIEW	view	view
dgm_obj	SEL_DGM_REF_OBJ	fixed connector, fixed component, or parametric connector	identifier for <i>selected_ptr</i>
dgm_mv_obj	SEL_DGM_MV_REF_OBJ	multiview connector or multiview component	identifier for <i>selected_ptr</i>
dgm_non_cable_wire	DTL_DGM_CONNECTION	wire is not in the cable	identifier for <i>selected_ptr</i>
dgm_cable_connector	DTL_DGM_CONNECTION	cable conductor	identifier for <i>selected_ptr</i>
dgm_cable	DTL_DGM_CONNECTION	cable	identifier for <i>selected_ptr</i>
dgm_rail	SEL_DGM_RAIL	rail	identifier for <i>selected_ptr</i>
dgm_hwy	DTL_DGM_HIGHWAY	highway	identifier for <i>selected_ptr</i>
draft_ent	DTL_DRAFT_ENT	N/A	entity
part	SEL_3D_PART	part	part Part and assembly IDs are not preserved over Pro/ENGINEER sessions.

option	sel_type	selected_ptr	selected_id
prt_or_asm	SEL_3D_PART	part, assembly	part, assembly Part and assembly IDs are not preserved over Pro/ENGINEER sessions.
table_cell	SET_TABLE_CELL	table	Segment ID

The following table lists the fields of the *Select3d* structure.

Type	Field	Description
char *	<i>part_ptr*</i>	A <i>ProMdl</i> handle (cast as char*) to a part to which the <i>selected_ptr</i> belongs. In Assembly mode, this is really the component pointer (either a part or an assembly). NULL if irrelevant.
char *	<i>assembly_ptr</i>	A <i>ProMdl</i> handle (cast as char*) to a root assembly. NULL if irrelevant.
int	<i>memb_id_tab[]</i>	Path to selected part through assembly members.
double	<i>aux_depth</i>	Not User Accessable
int	<i>sel_elem_id</i>	For tables, Segment ID
INT_FUNC	<i>pre_filter</i>	User-specified filter function
INT_FUNC	<i>post_filter</i>	User-specified filter function
char	<i>*auxiliary</i>	Use Not Recommended
char	<i>*p_object</i>	For tables, references the drawing model.
int	<i>memb_num</i>	Number of members in the assembly path.

Type	Field	Description
int	<i>sel_type</i>	Type of the selected element. For tables, SEL_TABLE_CELL.
int	<i>selected_id</i>	Identifier of the selected element.
char *	<i>selected_ptr</i>	Pointer to the selected element, for example, the table.
double	<i>select_pnt[3]</i>	xyz of the selected point in the member coordinate system.
double	<i>sel_param[2]</i>	<i>uv</i> parameters of selected point on surface; the <i>t</i> parameter for an edge (<i>sel_param[0]</i>). For tables, row and column number of a table cell, (<i>sel_param[row,column]</i>).
double	<i>sel_depth</i>	Selection depth, in model coordinates.
char *	<i>view_ptr</i>	View to which the <i>selected_ptr</i> belongs. This is used in Drawing mode.
int	<i>window_id</i>	Selected window ID
int	<i>use_cache_node</i>	PTC Internal
int	<i>node_type</i>	Type of node picked in tree tool, or 0
char	<i>*cache_data</i>	PTC Internal
char	<i>*ptr_exp</i>	PTC Internal



Geometry Traversal

This appendix illustrates the relationships between faces, contours, and edges.

Topic	Page
Overview	J - 2
Geometry Terms	J - 2

Overview

Note the following:

- A simple rectangular face has one contour and four edges.
- A contour will traverse a boundary such that the part face is always on the right-hand side (RHS). For an external contour, the direction of traversal is clockwise. For an internal contour, the direction of traversal is counterclockwise.
- If a part is extruded from a sketch that has a U-shaped cross section, there will be separate surfaces at each leg of the U-channel.
- If a part is extruded from a sketch that has a square-shaped cross section, and a slot feature is then cut into the part to make it look like a U-channel, there will be one surface across the legs of the U-channel. In other words, the original surface of the part is represented as one surface with a cut through it.

To Walk Through the Geometry of a Block

1. Walk through the surfaces of a part, using **ProSolidSurfaceVisit()**.
2. Walk through the contours of each surface, using **ProSurfaceContourVisit()**.
3. Walk through the edges of each contour, using **ProContourEdgeVisit()**.

Geometry Terms

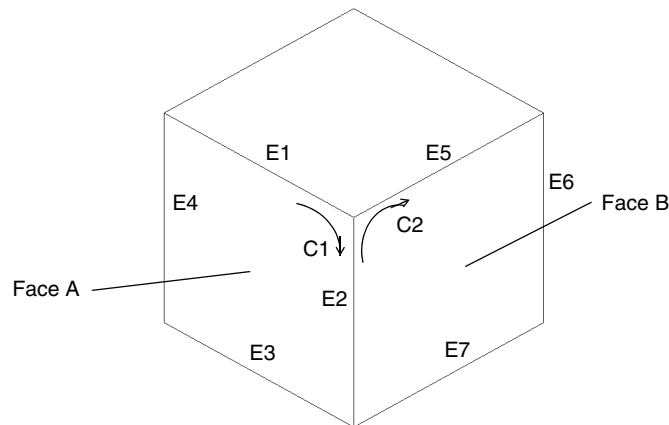
Consider the following definitions:

- *surface*—An ideal geometric representation, that is, an infinite plane.
- *face*—A trimmed surface. A face has one or more contours.
- *contour*—A closed loop on a face. A contour consists of multiple edges. A contour can belong to one face only.
- *edge*—The boundary of a trimmed surface.

An edge of a solid is the intersection of two surfaces. The edge belongs to those two surfaces, hence to two contours. An edge of a datum surface can be either the intersection of two datum surfaces, or the external boundary of the surface. If the edge is the intersection of two datum surfaces, it will belong to those two surfaces (hence, to two contours). If the edge is the external boundary of the datum surface, it will belong to that surface alone (hence, to a single contour).

Examples 1 through 5 show some sample parts and list the information about their surfaces, faces, contours, and edges.

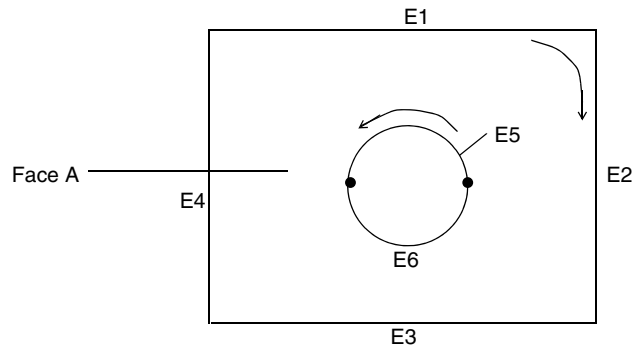
Example 1



This part has 6 faces.

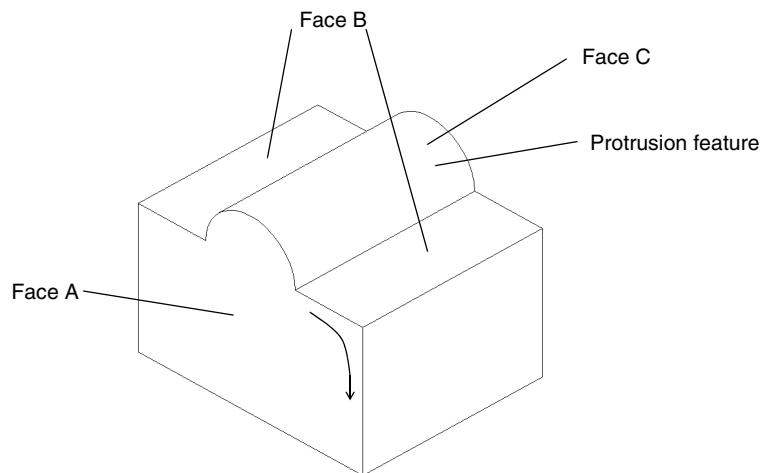
- Face A has 1 contour and 4 edges.
- Edge E2 is the intersection of faces A and B.
- Edge E2 is a component of contours C1 and C2.

Example 2



Face A has 2 contours and 6 edges.

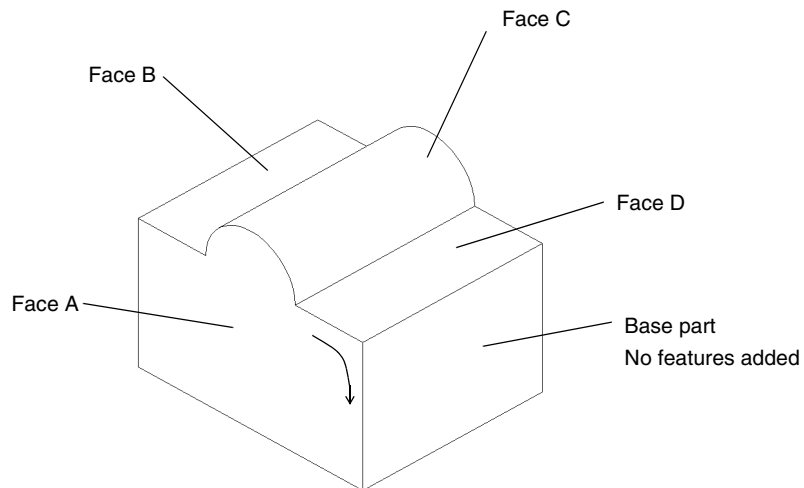
Example 3



This part was extruded from a rectangular cross section. The feature on the top was added later as an extruded protrusion in the shape of a semicircle.

- Face A has 1 contour and 6 edges.
- Face B has 2 contours and 8 edges.
- Face C has 1 contour and 4 edges.

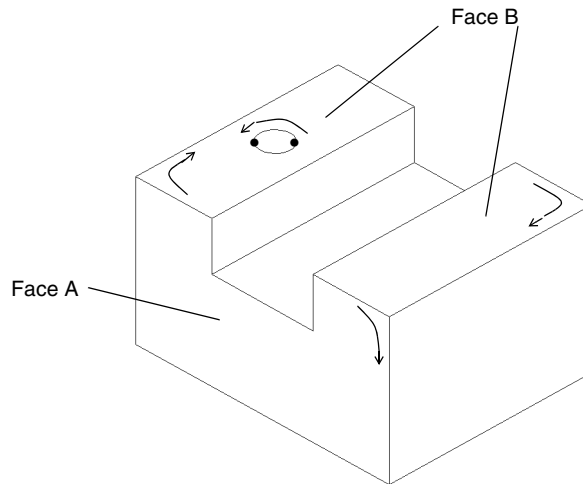
Example 4



This part was extruded from a cross section identical to Face A. In the Sketcher, the top boundary was sketched with two lines and an arc. The sketch was then extruded to form the base part, as shown.

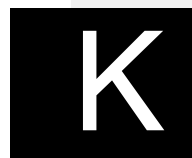
- Face A has 1 contour and 6 edges.
- Face B has 1 contour and 4 edges.
- Face C has 1 contour and 4 edges.
- Face D has 1 contour and 4 edges.

Example 5



This part was extruded from a rectangular cross section. The slot and hole features were added later.

- Face A has 1 contour and 8 edges.
- Face B has 3 contours and 10 edges.



Geometry Representations

This appendix describes the geometry representations of the data structures defined in ProGeomitem.h. These structures are output by the geometry functions described in detail in the 'Core: 3D Geometry' chapter.

Topic	Page
Domain of Evaluation	K - 2
Surface Data Structures	K - 3
Edge and Curve Data Structures	K - 16

Domain of Evaluation

Surfaces and edges can be extended from their original domain as the user continues to add features to the model. For example, the user can add a feature such as a draft surface or local push, which requires the original surface to be extended outside its original domain.

When this occurs, you will find that the *uv* parameters of the surface have been extended. The *ProSurfacedata* data structure reflects the extension, and returns the updated values for the *u* and *v* extents.

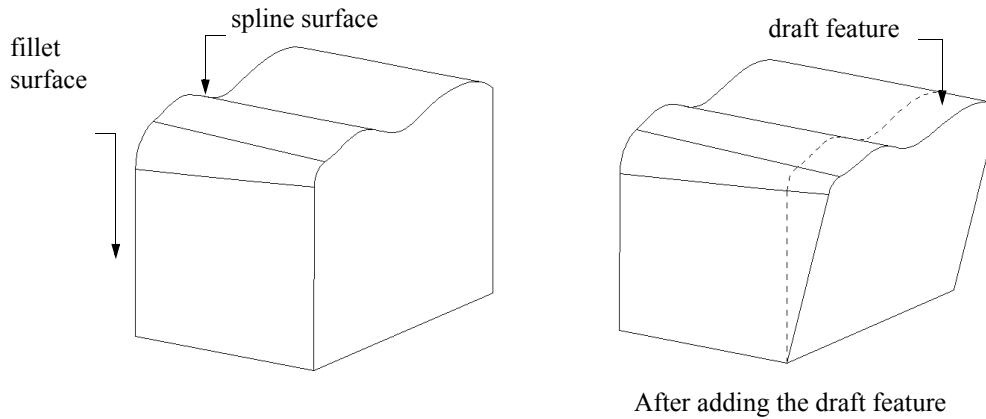
Because the evaluator functions use the analytical surface (or curve) definition, they work for any parameter values. Thus, any surface (or curve) can be extended as needed. In addition, if you pass in parameters outside the current *uv* domain, the evaluator functions still return values for the parameters as requested.

If you are using the evaluators supplied by Pro/TOOLKIT, you do not have to do anything. For surfaces, the evaluator functions work over this extended range of parameters. Your evaluator function for foreign datum surfaces is also expected to allow for extrapolation.

Edges are always parameterized between 0.0 and 1.0. When surfaces are extended, new edges are created that have parameters in the range 0.0 to 1.0.

If you develop your own evaluator functions, you must be aware that the domain of a surface can be extended, as with foreign datum surfaces.

Figure K-1: Surfaces



Surface Data Structures

The surface structure contains data that describes the boundary of the surface, and a pointer to the primitive surface on which it lies. The primitive surface is a three-dimensional geometric surface parameterized by two variables (u and v). The surface boundary consists of closed loops (contours) of edges. Each edge is attached to two surfaces, and each edge contains the u and v values of the portion of the boundary that it forms for both surfaces. Surface boundaries are traversed clockwise around the outside of a surface, so an edge has a direction in each surface with respect to the direction of traversal.

Other data found in the surface structure includes the rectangular extents of the two-dimensional domain, the three-dimensional surface, and a flag indicating whether the surface normal points towards the inside or outside of the part. The user data is intended for run-time use only, and this information is not stored with the surface.

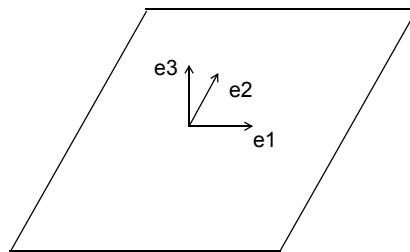
This section describes the surface data structures. The data structures are listed in order of complexity. For ease of use, the alphabetical listing of the data structures is as follows:

- Cone
- Coons Patch
- Cylinder

- Cylindrical Spline Surface
- Fillet Surface
- Foreign Surface
- General Surface of Revolution
- NURBS Surface
- Plane
- Ruled Surface
- Second Derivative Spline Surface
- Spline Surface
- Tabulated Cylinder
- Torus

Plane

Figure K-2: Plane



The plane entity consists of two perpendicular unit vectors ($e1$ and $e2$), the normal to the plane ($e3$), and the origin of the plane.

Data Format:

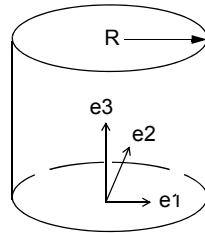
```
e1[3]      Unit vector, in the u direction
e2[3]      Unit vector, in the v direction
e3[3]      Normal to the plane
origin[3]   Origin of the plane
```

Parameterization:

$$(x, y, z) = u * e1 + v * e2 + origin$$

Cylinder

Figure K-3: Cylinder



The generating curve of a cylinder is a line, parallel to the axis, at a distance R from the axis. The radial distance of a point is constant, and the height of the point is v .

Data Format:

<code>e1[3]</code>	Unit vector, in the u direction
<code>e2[3]</code>	Unit vector, in the v direction
<code>e3[3]</code>	Normal to the plane
<code>origin[3]</code>	Origin of the cylinder
<code>radius</code>	Radius of the cylinder

Parameterization:

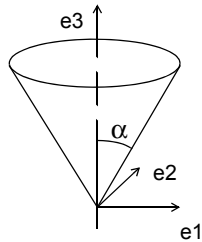
$$(x, y, z) = \text{radius} * [\cos(u) * e1 + \sin(u) * e2] + v * e3 + \text{origin}$$

Engineering Notes:

For the cylinder, cone, torus, and general surface of revolution, a local coordinate system is used that consists of three orthogonal unit vectors ($e1$, $e2$, and $e3$) and an origin. The curve lies in the plane of $e1$ and $e3$, and is rotated in the direction from $e1$ to $e2$. The u surface parameter determines the angle of rotation, and the v parameter determines the position of the point on the generating curve.

Cone

Figure K-4: Cone



The generating curve of a cone is a line at an angle α to the axis of revolution that intersects the axis at the origin. The v parameter is the height of the point along the axis, and the radial distance of the point is $v * \tan(\alpha)$.

Data Format:

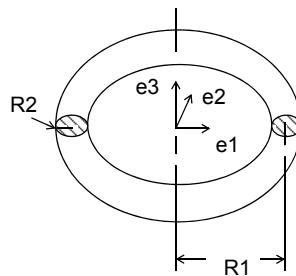
<code>e1[3]</code>	Unit vector, in the u direction
<code>e2[3]</code>	Unit vector, in the v direction
<code>e3[3]</code>	Normal to the plane
<code>origin[3]</code>	Origin of the cone
<code>alpha</code>	Angle between the axis of the cone and the generating line

Parameterization:

$$(x, y, z) = v * \tan(\alpha) * [\cos(u) * e_1 + \sin(u) * e_2] + v * e_3 + \text{origin}$$

Torus

Figure K-5: Torus



The generating curve of a torus is an arc of radius $R2$ with its center at a distance $R1$ from the origin. The starting point of the generating arc is located at a distance $R1 + R2$ from the origin, in the direction of the first vector of the local coordinate system. The radial distance of a point on the torus is $R1 + R2 * \cos(v)$, and the height of the point along the axis of revolution is $R2 * \sin(v)$.

Data Format:

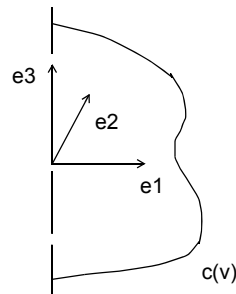
```
e1[3]      Unit vector, in the u direction
e2[3]      Unit vector, in the v direction
e3[3]      Normal to the plane
origin[3]   Origin of the torus
radius1     Distance from the center of the
            generating arc to the axis of
            revolution
radius2     Radius of the generating arc
```

Parameterization:

$$(x, y, z) = (R1 + R2 * \cos(v)) * [\cos(u) * e1 + \sin(u) * e2] + R2 * \sin(v) * e3 + \text{origin}$$

General Surface of Revolution

Figure K-6: General Surface of Revolution



A general surface of revolution is created by rotating a curve entity, usually a spline, around an axis. The curve is evaluated at the normalized parameter v , and the resulting point is rotated around the axis through an angle u . The surface of revolution data structure consists of a local coordinate system and a curve structure.

Data Format:

```
e1[3]      Unit vector, in the u direction
e2[3]      Unit vector, in the v direction
e3[3]      Normal to the plane
```

origin[3] Origin of the surface of revolution
 curve Generating curve

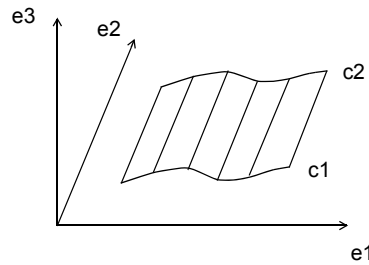
Parameterization:

curve(v) = (c1, c2, c3) is a point on the curve.

$$(x, y, z) = [c1 * \cos(u) - c2 * \sin(u)] * e1 + [c1 * \sin(u) + c2 * \cos(u)] * e2 + c3 * e3 + \text{origin}$$

Ruled Surface

Figure K-7: Ruled Surface



A ruled surface is the surface generated by interpolating linearly between corresponding points of two curve entities. The u coordinate is the normalized parameter at which both curves are evaluated, and the v coordinate is the linear parameter between the two points. The curves are not defined in the local coordinate system of the part, so the resulting point must be transformed by the local coordinate system of the surface.

Data Format:

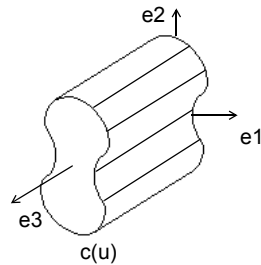
e1[3] Unit vector, in the u direction
 e2[3] Unit vector, in the v direction
 e3[3] Normal to the plane
 origin[3] Origin of the ruled surface
 curve_1 First generating curve
 curve_2 Second generating curve

Parameterization:

(x', y', z') is the point in local coordinates.
 $(x', y', z') = (1 - v) * C1(u) + v * C2(u)$
 $(x, y, z) = x' * e1 + y' * e2 + z' * e3 + \text{origin}$

Tabulated Cylinder

Figure K-8: Tabulated Cylinder



A tabulated cylinder is calculated by projecting a curve linearly through space. The curve is evaluated at the u parameter, and the z coordinate is offset by the v parameter. The resulting point is expressed in local coordinates and must be transformed by the local coordinate system to be expressed in part coordinates.

Data Format:

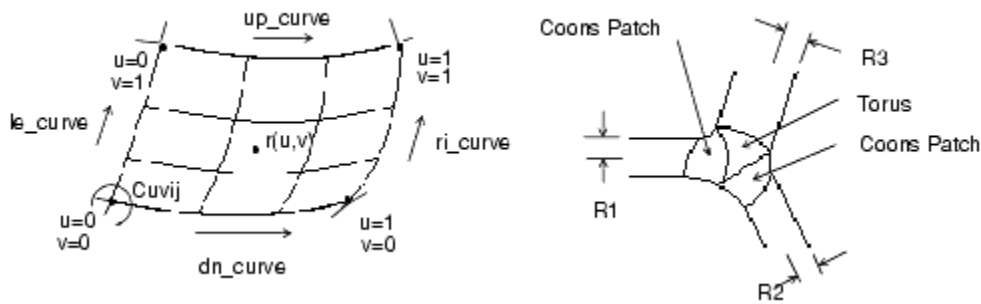
<code>e1[3]</code>	Unit vector, in the u direction
<code>e2[3]</code>	Unit vector, in the v direction
<code>e3[3]</code>	Normal to the plane
<code>origin[3]</code>	Origin of the tabulated cylinder
<code>curve</code>	Generating curve

Parameterization:

(x', y', z') is the point in local coordinates.
 $(x', y', z') = C(u) + (0, 0, v)$
 $(x, y, z) = x' * e1 + y' * e2 + z' * e3 + origin$

Coons Patch

Figure K-9: Coons Patch



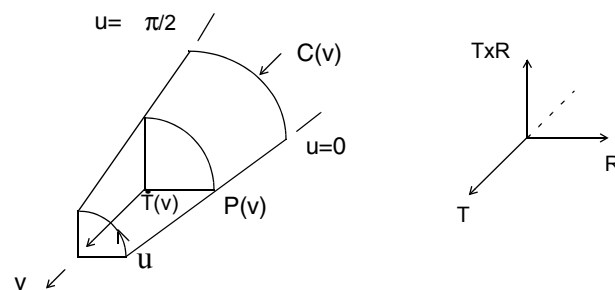
A Coons patch is used to blend surfaces together. For example, you would use a Coons patch at a corner where three fillets (each of a different radius) meet.

Data Format:

le_curve	$u = 0$ boundary
ri_curve	$u = 1$ boundary
dn_curve	$v = 0$ boundary
up_curve	$v = 1$ boundary
point_matrix[2][2]	Corner points
uvder_matrix[2][2]	Corner mixed derivatives

Fillet Surface

Figure K-10: Fillet Surface



A fillet surface is found where a round or a fillet is placed on a curved edge, or on an edge with non-constant arc radii. On a straight edge, a cylinder would be used to represent the fillet.

Data Format:

```
pnt_spline    P(v) spline running along the u = 0 boundary
ctr_spline    C(v) spline along the centers of the
               fillet arcs
tan_spline    T(v) spline of unit tangents to the
               axis of the fillet arcs
```

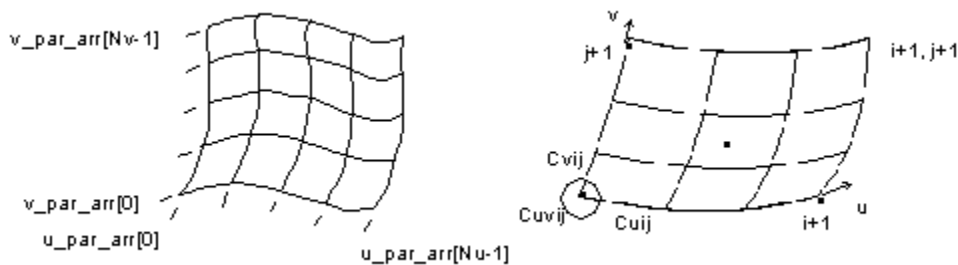
Parameterization:

$$R(v) = P(v) - C(v)$$

$$(x, y, z) = C(v) + R(v) * \cos(u) + T(v) \times R(v) * \sin(u)$$

Spline Surface

Figure K-11: Spline Surface



The parametric spline surface is a nonuniform bicubic spline surface that passes through a grid with tangent vectors given at each point. The grid is curvilinear in uv space. Use this for bicubic blending between corner points.

Data Format:

```
u_par_arr[]    Point parameters, in the u
                direction, of size Nu
v_par_arr[]    Point parameters, in the v
                direction, of size Nv
point_arr[][3] Array of interpolant points, of
                size Nu x Nv
u_tan_arr[][3] Array of u tangent vectors
                at interpolant points, of size
                Nu x Nv
v_tan_arr[][3] Array of v tangent vectors at
```

`uvder_arr[][3]` Array of mixed derivatives at
interpolant points, of size
 $N_u \times N_v$

Engineering Notes:

- Allows for a unique 3x3 polynomial around every patch.
- There is second order continuity across patch boundaries.
- The point and tangent vectors represent the ordering of an array of $[i][j]$, where u varies with i , and v varies with j . In walking through the `point_arr[][3]`, you will find that the innermost variable representing $v(j)$ varies first.

Second Derivative Spline Surface

The Second Derivative Spline Surface (ProSpline2ndDersrfddata) is a bicubic spline surface with possibility of altering the degree of boundary segments to accommodate corresponding curvature (2nd derivatives) conditions. Use this for bicubic blending with second degree derivatives along boundaries.

Data format:

<code>u_par_arr[]</code>	Point parameters, in the u direction, of size N_u
<code>v_par_arr[]</code>	Point parameters, in the v direction, of size N_v
<code>point_arr[][3]</code>	Array of interpolant points, of size $N_u \times N_v$
<code>u_tan_arr[][3]</code>	Array of u tangent vectors at interpolant points, of size $N_u \times N_v$
<code>v_tan_arr[][3]</code>	Array of v tangent vectors at interpolant points, of size $N_u \times N_v$
<code>uvder_arr[][3]</code>	Array of mixed derivatives at interpolant points, of size $N_u \times N_v$
<code>(*u_der2_arrs[2])[3]</code>	2 Arrays of 2nd U derivatives along V boundaries interpolation points, size of N_v .
<code>(*v_der2_arrs[2])[3]</code>	2 Arrays of 3rd V derivatives along U boundaries interpolation points, size of N_u .

u_par_arr[]	Point parameters, in the u direction, of size Nu
(*uuv_der[2])[3]	2 Arrays of uuv mixed derivatives along V boundaries interpolation points, size of Nv.
(*vvu_der[2])[3]	Arrays of vvu mixed derivatives along U boundaries interpolation points, size of Nv.
der4[4][3] UUVV	derivatives at the corners

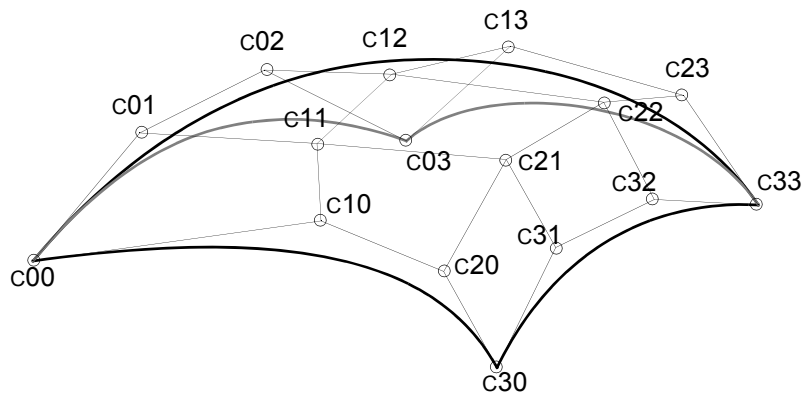
Engineering notes:

- Each second degree derivatives array can be NULL. If it is defined then corresponding 3rd degree mixed derivatives should be defined.
- Each der4 can be NULL and should be defined if 2nd degree derivatives are defined in both directions at the corresponding corner.

NURBS Surface

The NURBS (nonuniform rational B-spline) surface is defined by basis functions (in u and v), expandable arrays of knots, weights, and control points.

Figure K-12: NURBS Surface



Cubic NURBS Surface

Data Format:

deg[2] Degree of the basis

	functions (in u and v)
<code>u_par_arr[]</code>	Array of knots on the parameter line u
<code>v_par_arr[]</code>	Array of knots on the parameter line v
<code>wgths[]</code>	Array of weights for rational NURBS, otherwise NULL
<code>c_point_arr[][3]</code>	Array of control points

Definition:

$$R(u, v) = \frac{\sum_{i=0}^{N1} \sum_{j=0}^{N2} C_{i,j} \times B_{i,k}(u) \times B_{j,l}(v)}{\sum_{i=0}^{N1} \sum_{j=0}^{N2} w_{i,j} \times B_{i,k}(u) \times B_{j,l}(v)}$$

k = degree in u
 l = degree in v
 $N1$ = (number of knots in u) - (degree in u) - 2
 $N2$ = (number of knots in v) - (degree in v) - 2
 $B_{i,k}$ = basis function in u
 $B_{j,l}$ = basis function in v
 $w_{i,j}$ = weights
 $C_{i,j}$ = control points $(x,y,z) * w_{i,j}$

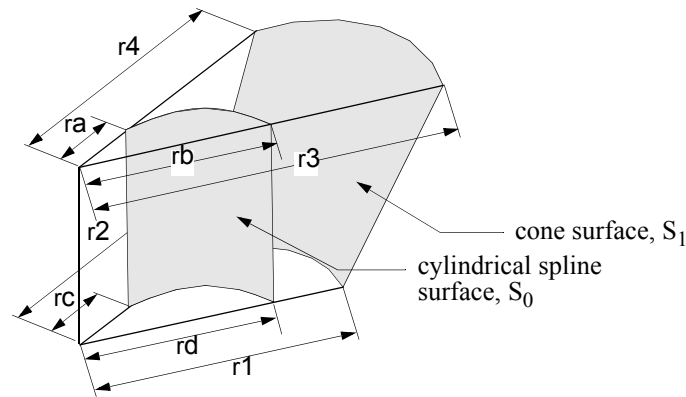
Engineering Notes:

The *weights* and *c_points_arr* arrays represent matrices of size *wgths*[$N1+1$][$N2+1$] and *c_points_arr* [$N1+1$][$N2+1$]. Elements of the matrices are packed into arrays in row-major order.

Cylindrical Spline Surface

The cylindrical spline surface is a nonuniform bicubic spline surface that passes through a grid with tangent vectors given at each point. The grid is curvilinear in modeling space.

Figure K-13: Cylindrical Spline Surface



Data Format:

```
e1[3]      x' vector of the local coordinate system
e2[3]      y' vector of the local coordinate system
e3[3]      z' vector of the local coordinate system,
            which corresponds to the axis of revolution
            of the surface
origin[3]   Origin of the local coordinate system
splsrfs     Spline surface data structure
```

The spline surface data structure contains the following fields:

```
u_par_arr[]      Point parameters, in the
                  u direction, of size Nu
v_par_arr[]      Point parameters, in the
                  v direction, of size Nv
point_arr[][3]    Array of points, in cylindrical
                  coordinates, of size Nu x Nv.
                  The array components are as follows:
                  point_arr[i][0] - Radius
                  point_arr[i][1] - Theta
                  point_arr[i][2] - Z
u_tan_arr[][3]    Array of u tangent vectors.
                  in cylindrical coordinates,
                  of size Nu x Nv
v_tan_arr[][3]    Array of v tangent vectors,
                  in cylindrical coordinates,
                  of size Nu x Nv
uvder_arr[][3]    Array of mixed derivatives,
                  in cylindrical coordinates,
                  of size Nu x Nv
```

Engineering Notes:

If the surface is represented in cylindrical coordinates (r , θ , z), the local coordinate system values (x' , y' , z') are interpreted as follows:

```
x' = r cos (theta)
y' = r sin (theta)
z' = z
```

A cylindrical spline surface can be obtained, for example, by creating a smooth rotational blend (shown in the figure on the previous page).

In some cases, you can replace a cylindrical spline surface with a surface such as a plane, cylinder, or cone. For example, in the figure, the cylindrical spline surface S_1 was replaced with a cone ($r_1 = r_2$, $r_3 = r_4$, and $r_1 \neq r_3$).

If a replacement cannot be done (such as for the surface S_0 in the figure ($r_a \neq r_b$ or $r_c \neq r_d$)), leave it as a cylindrical spline surface representation.

Foreign Surface

The foreign surface consists of two perpendicular unit vectors (e_1 and e_2), the normal to the plane (e_3), the origin of the plane and the foreign ID returned by the function **user_init_surf()**.

Data Format:

```
e1[3]      Unit vector, in the u direction
e2[3]      Unit vector, in the v direction
e3[3]      Normal to the plane
origin[3]   Origin of the plane
foreign_id  Foreign ID returned by user_init_surf()
```

Parameterization is established by the Foreign Surface callback function **user_eval_surf()**.

Edge and Curve Data Structures

The data structures are used to represent edges (line, arc, and spline) as well as the curves (line, arc, spline, and NURBS) within the surface data structures.

This section describes the edge and curve data structures, arranged in order of complexity. For ease of use, the alphabetical listing of the edge and curve data structures is as follows:

- Arc on page K - 17
- Line on page K - 17
- NURBS on page K - 18
- Spline on page K - 19
- Ellipse on page K - 19

Arc

The arc entity is defined by a plane in which the arc lies. The arc is centered at the origin, and is parameterized by the angle of rotation from the first plane unit vector in the direction of the second plane vector. The start and end angle parameters of the arc and the radius are also given. The direction of the arc is counterclockwise if the start angle is less than the end angle, otherwise it is clockwise.

Data Format:

vector1[3]	First vector that defines the plane of the arc
vector2[3]	Second vector that defines the plane of the arc
origin[3]	Origin that defines the plane of the arc
start_angle	Angular parameter of the starting point
end_angle	Angular parameter of the ending point
radius	Radius of the arc.

Parameterization:

```
t' (the unnormalized parameter) is
(1 - t) * start_angle + t * end_angle
(x, y, z) = radius * [cos(t') * vector1 +
sin(t') * vector2] + origin
```

Line

Data Format:

end1[3]	Starting point of the line
end2[3]	Ending point of the line

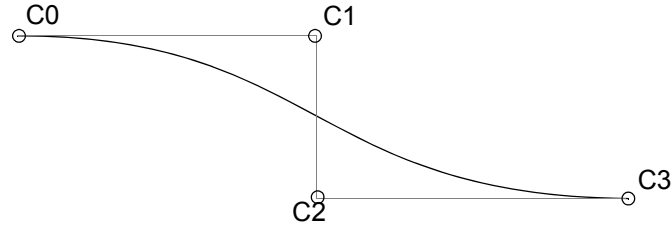
Parameterization:

$$(x, y, z) = (1 - t) * \text{end1} + t * \text{end2}$$

NURBS

The NURBS (nonuniform rational B-spline) curve is defined by expandable arrays of knots, weights, and control points.

Figure K-14: Cubic NURBS Curve



Data Format:

degree	Degree of the basis function
params[]	Array of knots
weights[]	Array of weights for rational NURBS, otherwise NULL.
c_pnts[][3]	Array of control points

Definition:

$$R(t) = \frac{\sum_{i=0}^N C_i \times B_{i,k}(t)}{\sum_{i=0}^N w_i \times B_{i,k}(t)}$$

k = degree of basis function

N = (number of knots) - (degree) - 2

w_i = weights

C_i = control points $(x, y, z) * w_i$

$B_{i,k}$ = basis functions

By this equation, the number of control points equals $N+1$.

Spline

The spline curve entity is a nonuniform cubic spline, defined by a series of three-dimensional points, tangent vectors at each point, and an array of unnormalized spline parameters at each point.

Data Format:

```
par_arr[]      Array of spline parameters
                (t) at each point.
pnt_arr[][3]   Array of spline interpolant points
tan_arr[][3]   Array of tangent vectors at
                each point
```

Parameterization:

x , y , and z are a series of unique cubic functions, one per segment, fully determined by the starting and ending points, and tangents of each segment.

Let p_{min} be the parameter of the first spline point, and p_{max} be the parameter of the last spline point. Then, t' , the unnormalized parameter, is $t * p_{max} + (1-t) * p_{min}$.

Locate the i th spline segment such that:

```
par_arr[i] < t' < par_arr[i+1]
(If  $t < 0$  or  $t > +1$ , use the first or last segment.)

t0 = (t' - par_arr[i]) / (par_arr[i+1] - par_arr[i])
t1 = (par_arr[i+1] - t') / (par_arr[i+1] - par_arr[i])
The coordinates of the points are then:

(x, y, z) = pnt_arr[i] * t1^2 * (1 + 2 * t0) +
            pnt_arr[i+1] * t0^2 * (1 + 2 * t1) +
            (par_arr[i+1] - par_arr[i]) * t0 * t1 *
            (tan_arr[i] * t1 - tan_arr[i+1] * t0)
```

Ellipse

Ellipses in 3D geometry is split into two identical half-ellipses. The ellipse is defined by its major and minor axis radius values. Similar to arcs, elliptic segments are defined by a plane in which the ellipse lies, centered at the origin, and parameterized by the angle of rotation. The direction of the ellipse is counterclockwise if the start angle is less than the end angle, otherwise it is clockwise.

Data Format:

```
center - Center point of the ellipse
```

major_axis_unit_vect - Direction for the X-axis of the ellipse
norm_axis_unit_vect - Direction for the Y-axis
major_len - The "radius" in the X-direction
minor_leng - The "radius" in the Y-direction
start_ang - The ellipse start angle
end_ang - The end angle for the ellipse

Parameterization:

t' (the unnormalized parameter) is
 $(1 - t) * \text{start\_angle} + t * \text{end\_angle}$
 $(x, y, z) = \text{major\_len} * [\cos(t') * \text{vector1}] +$
 $\text{minor\_len} * [\sin(t') * \text{vector2}] + \text{origin}$

References:

Faux, I.D., M.J. Pratt. *Computational Geometry for Design and Manufacture*. Ellis Harwood Publishers, 1983.

Mortenson, M.E. *Geometric Modeling*. John Wiley & Sons, 1985.



Debugging Pro/TOOLKIT Applications

This appendix describes how to debug Pro/TOOLKIT applications.

Topic	Page
Building a Pro/TOOLKIT Application for Debugging	L - 2
Debugging Techniques	L - 2
Debugging a Multiprocess Application	L - 5

Building a Pro/TOOLKIT Application for Debugging

The following tables list commonly used changes for sample makefiles that allow debugging of a Pro/TOOLKIT application. Consult your compiler documentation for more information on the flags specified below and other debugging options.

On Windows

Compile Line Changes	Link Line Changes
Change CCFLAGS or CPPFLAGS to add <code>/Od /Z7</code> .	Change the <code>/debug</code> flag to <code>/debug:full</code> . Remove or comment the lines that delete <code>\$(OBJS)</code> after a successful build.

On UNIX and Linux

Compile Line Changes	Link Line Changes
Change CCFLAGS or CPPFLAGS to add <code>-g</code> .	Remove or comment the lines that delete <code>\$(OBJS)</code> after a successful build.

Debugging Techniques

Depending on the type and implementation for your application, you can use one of the following debugging techniques:

- If your application is a DLL that creates user interface and notification callbacks to be used by the interactive user, refer to the following section on Debugging an Interactive DLL.
- If your application is a DLL that runs in batch mode (from `user_initialize()`), refer to Debugging a Batch Mode DLL.
- If your application is a spawn or asynchronous application, refer to Debugging a Multiprocess Application.

Debugging an Interactive DLL

1. Start Pro/ENGINEER with the DLL registered and loaded.
2. Use the debugger to attach to the process `xtop` or `xtop.exe`.
3. Set breakpoints in the callback functions that you wish to debug.

Debugging a Batch Mode DLL

Debugging Without Setting Breakpoints Before Loading Code

The following steps outline the procedure to debug applications using a debugger such as dbx that does not allow breakpoints to be set before the code is loaded.

1. Setup a Pro/ENGINEER start command that uses the debugger as follows:
 - a) Edit the proe.psf file (or the related .psf file for the given command).
 - b) In this file change the RUN line for xtop, to invoke the debugger first, as:

```
RUN="$PRO_DIRECTORY/bin/proe1"
```

```
RUN="dbx $PRO_DIRECTORY/bin/proe1"
```

2. Run the Pro/ENGINEER start command. The debugger starts.
3. Set up a breakpoint that triggers when the DLL is loaded.

In dbx, enter `stop dlopen <DLL name>`. For example,
`stop dlopen pt_inst_test.dll`.

If you are using a graphical debugger such as Sun's Workshop, the same procedure is possible. Workshop allows a "CUSTOM" breakpoint using dbx syntax, such as `dlopen pt_inst_test.dll`.

4. Enter run. The program runs to the point of loading the DLL.
5. When the dlopen breakpoint is triggered, add a breakpoint to `user_initialize()`. At this point, the DLL is loaded and the `user_initialize()` function is in memory.

Debugging With Breakpoints Set Before Loading Code

The following steps outline the procedure to debug applications using a debugger such as Microsoft Visual Studio that allows breakpoints to be set before the code is loaded.

1. Setup a Pro/ENGINEER start command that uses the debugger as follows:
 - a) Edit the `proe.psf` file (or the related `.psf` file for the given command).
 - b) In this file, change the RUN line for `xtop`, to invoke the debugger first as:

```
RUN="%PRO_DIRECTORY%\%PRO_MACHINE_TYPE%\obj\xtop.exe"

RUN=devenv
"%PRO_DIRECTORY%\%PRO_MACHINE_TYPE%\obj\xtop.exe"
```
2. Run the Pro/ENGINEER start command. The debugger starts.
3. Load the source file containing the functions into the debugger, and graphically set a breakpoint to the start of `user_initialize()`.
4. Run the program from the debugger User Interface. Even if the debugger indicates that no debugger information is available for `xtop.exe`, continue debugging.

When the DLL is loaded, the program will stop in the DLL's `user_initialize()`.

Debugging Pro/TOOLKIT DLLs on Windows

The Pro/ENGINEER executable contains a default exception handler on Windows. This handler will catch and react to exceptions generated by a Pro/TOOLKIT DLL by default.

To allow debugging on Windows without using the Pro/ENGINEER exception handler, set the environment variable `ALLOW_MS_DEBUG` to `true` in the **Control Panel**.

Setting `ALLOW_MS_DEBUG` to `true` forces Pro/ENGINEER executable not to execute the exception handler and thus the process will stop in the debugger on any exception. Alternatively, it is possible to set exception handling in the debugger to stop on the first occurrence of the exception, even for exceptions that are handled.

Debugging a Multiprocess Application

Debugging a Synchronous Spawn Mode Application

1. Start Pro/ENGINEER with the application registered and started.
2. Use the debugger to attach to the application process, for example `pt_inst_test.exe`.
3. Set breakpoints in the callback functions that you wish to debug.

Debugging an Asynchronous Spawn Mode Application

1. Start the application from the debugger. Set a breakpoint to stop the application before the call to **ProEngineerStart()** or **ProEngineerConnect()**.

Glossary

Pro/TOOLKIT and Pro/ENGINEER Terminology

This glossary contains words that have meanings specific to Pro/TOOLKIT. For definitions of words that apply to Pro/ENGINEER in general, see the glossary in the *Introduction to Pro/ENGINEER*.

asynchronous mode

The mode in which Pro/TOOLKIT can start or connect to a new Pro/ENGINEER session and invoke operations in it.

child

An item, such as view, part, or feature, that is dependent on another item for its existence. See also *Parent*.

complement mode

In this mode, the color becomes the *exclusive or* of the old color and the color being drawn. You can use this for creating dynamic graphics, such as rubberbands.

configuration file

A special text file that contains default settings for many Pro/ENGINEER functions. Default environment, units, files, directories, and so on are set when Pro/ENGINEER reads this file when it is started. A configuration file can reside in the startup directory to set the values for your working session only, or it can reside in the load directory to set values for all users running that version of Pro/ENGINEER. Also known as the *config.pro* file.

contour

A closed loop on a face. A contour consists of multiple edges. A contour can belong to only one face.

coordinate system

A means of identifying points in space using a particular point in three-dimensional space (the origin) and three mutually perpendicular axes through the origin (the coordinate axes). Pro/TOOLKIT uses four different coordinate systems: model, screen, window, and drawing.

curve

A continuous one-dimensional subset of three-dimensional space. The Pro/TOOLKIT definition covers the geometry of not only datum curves, but also other features whose geometry is treated in the same way: axes and geometry edges.

display list

A list of vectors used to represent the shape of the model in the view.

domain

The portion or portions of a surface that correspond to real geometry.

draft entity

The graphical items created in Pro/ENGINEER using the options under **Detail, Sketch**. The possible values are arc, ellipse, line, point, polygon, and spline.

draft group

A group of detail items that can contain notes and symbol instances, as well as draft entities.

edge

The curve along which two geometrical faces intersect.

element

A structural component in a Pro/ENGINEER model that has its own internal identifier (and sometimes a name) and can be selected. Examples of elements are face, edge (but not contour), datum plane, datum surface, datum curve, axis, datum point, and feature.

entity

A geometric element that has the geometry of a curve, excluding edges. Entities such as datum curves are accessed using their own Pro/TOOLKIT functions, even though geometrically they behave like edges. Note that entities and draft entities are quite different: draft entities refer to two-dimensional items on a drawing.

evaluate

To invoke the evaluation, at a point on an edge or surface, of the Pro/ENGINEER parametric equations for that edge or surface. Evaluation provides a description of the three-dimensional geometry, in model coordinates.

face

A geometry element that describes a geometrical surface and its relationship with other geometry elements (edges and faces).

highlight

To emphasize an element by modifying its appearance on the workstation surface, usually by changing its color.

information window

A Pro/ENGINEER window that displays information such as object lists, mass properties, and BOM.

leader

The arrow that points from a note or symbol to either a point on an edge of the geometry of a model in a drawing view or to a point in the drawing.

load directory

The directory where Pro/ENGINEER is loaded.

macro keys

The function keys or key sequences for which you predefined a menu option or sequence of menu options. The predefined menu options enable you to pick a macro from a menu that is currently on the screen.

main menu

An autonomous menu with its own title. The other type of menu is a submenu.

mass properties

The information about the distribution of mass in the part or assembly. The C structure used to describe mass properties is *ProMassProperty*, and is declared in the header file *ProSolid.h*. The data structure includes fields for the following: volume, surface area, density, mass, center of gravity (COG), inertia matrix, inertia tensor, COG inertia tensor, eigenvalues of the COG inertia, and eigenvectors of the COG inertia.

matrix

A two-dimensional array used for transformations. See also *Transformation*.

menu

A list of options presented by Pro/ENGINEER that you select using the mouse or predefined macro keys. See the 'User Interface: Menus, Commands, and Pop-up menus' chapter for more information on menus.

menu file

A text file that enables you to specify your own text for the name of a menu button, the one-line help text that appears when you place the cursor over that button, and translations for both of these.

message file

A text file that enables you to provide your own translation of the text message. The message file consists of groups of four lines, one group for each message that you want to write out.

model

A top-level object in a Pro/ENGINEER mode.

model coordinates

The coordinate system Pro/ENGINEER uses internally to define the geometry of a model. You can visualize these coordinates by creating a coordinate system datum with the option **Default**.

model item

A generic object used to represent any item contained in any type of model, for the purpose of functions whose actions are applicable to all these types of item.

notify

Enables you to trap certain classes or events in the Pro/ENGINEER session and arrange for a function in the Pro/TOOLKIT program to be called before or after such a trapped event.

object

An item stored as a single file, such as a part, assembly, or drawing.

overlay view

The view for a whole drawing sheet.

parameter

A user-driven property that can be added to elements in a Pro/ENGINEER model and used to drive dimensional relations. Parameters consist of a name, type, and a value that can be an integer, double, or string. Parameters are accessible through the Pro/ENGINEER user interface, as opposed to attributes that are private to Pro/TOOLKIT.

parent

An item that has other items dependent upon it for their existence. For example, the base feature has all other features dependent upon it. If a parent is deleted, all dependent (children) items are deleted.

pipeline

A set of interconnecting pipes and fitments consisting of an extension which terminates at open ends, non-open ends, or junctions (branches).

pipeline branch

Pipes grouped into extensions such that the extension which continues across the branch has a continuous direction of flow.

pipeline extension

A non-branching sequence of pipeline items.

pipeline feature

A feature which names the pipeline to show its grouping but contains no geometry.

pipeline fitting

A component that connects two pipe segments, for example, a corner or a valve.

pipeline junction

An assembly component or a datum point that represents a part which joins three or more pipe segments.

pipeline member

A extension terminator, series, or junction.

pipeline network

A data structure which contains references to pipeline objects and are structured to show their connectivity and sequence in relation to the flow.

pipeline object

A segment, a fitting, or a stubin.

pipeline segment

A section of pipe, either straight or arced.

pipeline series

A non-branching sequence of pipeline objects.

pipeline stubin

A datum point which joints three or more series.

pipeline terminator

The open or non-open ends of the pipeline.

set mode

In this mode, any graphics draw command sets the appropriate pixels to the color being drawn.

submenu

A menu that acts as an extension to the menu above it. A submenu has no title and is active at the same time as the menu above it. Selecting from the menu above it does not close the submenu.

surface

A continuous two-dimensional subset of three-dimensional space.

synchronous mode

The normal mode of operation that makes it appear that Pro/TOOLKIT is part of the Pro/ENGINEER process.

tessellation

The process of subdividing an edge into multiple smaller edges.

transformation

A change from one coordinate system to another. A transformation between two coordinate systems is represented by a 4x4 matrix.

user attributes

The attributes added by a user to add description to the object beyond the geometric definition. For example, stock number, price, and cost per unit are all user attributes.

vector

A straight line segment that has both magnitude and direction.

version stamp

Provides a way of keeping track of changes in a Pro/ENGINEER model to which your Pro/TOOLKIT application may need to respond.

view

In Part mode, a view is the orientation of the object. In Drawing mode, a view is part of the drawing that represents the model.

wide string

A data type that allows for the fact that some character sets (such as Japanese KANJI) use a bigger character set than can be coded into the usual 1-byte char type.

window

A rectangular area of the workstation surface in which you work or in which the system displays messages. Pro/ENGINEER uses a Main Window, Message Window, and subwindows. You can also specify a single-window environment.

Index

Numerics

- 2-D
 - display list 18-19
 - sections
 - adding dimensions to 40-12
 - adding entities 40-9
 - allocating 40-3
 - 3-D
 - display list 18-19
 - 7-bit ASCII
 - definition B-3
 - 8-bit ASCII
 - definition B-3
- ## A
- Access
 - material data 3-22
 - menu buttons 15-58
 - parameters 7-4
 - to explode states 45-15
 - Accessing
 - Mechanica Items 61-6
 - Accuracy
 - of solids 3-10
 - Action functions
 - for visits 1-59
 - Actions
 - adding 15-5
 - Activate
 - explode state 45-15
 - window 18-6
 - Add
 - action to Pro/ENGINEER 15-5
 - animation frames to a movie 21-5
 - animation objects to frames 21-3

- datum points 5-31
- dimensions to a 2-D section 40-12
- external object data 66-11
- family table items 9-7
- items to layers 2-15
- menu buttons 15-44
- menu to menu bar 15-15
- section entities 40-9
- add or update 1-23
- Adding a Customized Function to the Relations Dialog in Pro/ENGINEER 6-4
- Allocate
 - 2-D sections 40-3
 - display data 66-7
 - external object references 66-13
 - selection data for external objects 66-10
 - simplified representations 49-6
 - text style structures 22-52
 - version stamps 2-12
- Analyze
 - manufacturing model 53-3
- Angles
 - example 5-26
- Animation 21-2
 - batch 21-5
 - creating 21-3
 - dialog box 21-2
 - frame
 - description 21-3
 - movies 21-5
 - object
 - description 21-2
 - refresh 18-3
 - single 21-4
- Annotation

- Associativity 22-19
- associativity
 - attachment 22-19
 - position 22-19
- Orientation 22-17
- plane
 - datum plane 22-17
 - flat surface 22-17
 - flat to screen 22-17
 - named view 22-17
 - rotation angle 22-19
 - viewing direction 22-19
- Annotation Elements
 - accessing 22-6
 - modifying 22-9
 - parameters,assigned 22-11
 - visiting 22-5
- Annotation Features
 - creating 22-4
 - overview 22-3
 - redefining 22-4
 - redefining, interactive 22-12
 - visiting 22-5
- Annotations
 - accessing 22-14
 - selection, interactive 22-21
- APIWizard
 - browsing
 - categories 1-8
 - functions 1-11
 - objects 1-10
 - user's guide 1-11
 - code examples 1-7
 - defined 1-2
 - display frame 1-6
 - documentation
 - online 1-2
 - PDF format 1-2
 - find (search mechanism) 1-13
 - frames in web browser 1-6
 - function descriptions 1-7
 - interface defined 1-6
 - Internet Explorer
 - Java2 plug in 1-3, 1-5
 - Java2 plugin 1-6
 - requirements 1-3, 1-5
 - troubleshooting 1-5
 - navigating the tree 1-7
 - Netscape Navigator
 - Java2 plugin not recommended 1-5
 - requirements 1-3
 - object description 1-7
 - searching for a string 1-13
 - supported web browsers 1-3
- Swing
 - class path-NT 1-4
 - class path-UNIX 1-4
 - download JFC archive 1-4
 - Java Foundation Class 1-3
 - topic/object/category selection frame 1-6
 - tree updating
 - defined 1-5
 - Java2 plugin requirements 1-5
- Applications
 - compiling and linking 1-32
 - core 1-43
 - debugging 1-39
 - program structure I-7
 - registering 1-33
 - stopping and restarting 1-37
 - structure of I-7
 - unlocking 1-39
 - using with Pro/TOOLKIT I-3
- Arcs
 - drawing 18-16
 - edges
 - extracting the diameter 5-23
 - representation K-17
- Argument
 - Management 69-2
- Arrays
 - element
 - description 29-5
 - expandable 1-53
 - allocating 1-54
 - code example 1-56
 - freeing 1-54
- Assemblies
 - active explode state 45-15
 - automatic interchange 45-15
 - components 45-2
 - assembling 45-13
 - assembling by feature creation 47-17
 - deleting 45-13
 - locating 45-11
 - traversing 45-5
 - visiting 45-5
 - conversion paths I-11
 - coordinate systems 8-3
 - cut out 45-15
 - explode states 45-15
 - exploded 45-14
 - hierarchy 45-4
 - interference checking 5-35
 - merging 45-15
 - process step 56-2
 - structure of 45-2
- Asynchronous mode
 - definition Glossary-1

- full 13-9, 13-11
 - non-interactive 13-10
 - simple 13-7
- Attachment
 - points and leaders 51-91
- Attributes
 - text 18-18
 - user
 - definition Glossary-7
- Auto Round Feature 36-21
- Automatic dimensioning 40-9
- Automatic interchange 45-15
- Auxiliary Applications
 - dialog box 1-37
- Auxiliary tools
 - parameters 53-11
- Axes
 - datums
 - visiting 5-9
 - geometry of 5-27

B

- Backup
 - model 2-7
- Bar elements 63-10
- Base window
 - identifier 18-1
- Batch animation 21-5
 - code example 21-6
- Batch mode
 - example 1-46
- Batch sessions 1-46
- Bind
 - evaluation functions 68-7
- Browsing
 - Pro/TOOLKIT categories with APIWizard 1-8
 - Pro/TOOLKIT functions with APIWizard 1-11
 - Pro/TOOLKIT objects with APIWizard 1-10
 - Pro/TOOLKIT user's guide with APIWizard 1-11
- Button
 - placing 15-22
- Buttons
 - accessibility 15-58
 - adding 15-44
 - example 15-46
 - adding to menu bar 15-8
 - menu bar 15-2
 - mode-specific 15-41
 - position 18-21
 - setting 15-57

C

- Cabling
 - cable geometry 58-17
 - cable identifiers 58-14
 - cable types 58-14
 - connectivity 58-15
 - harness clearance 58-17
 - routing locations 58-16
 - routing procedure 58-18
- Callbacks
 - external Analysis 67-5
 - external analysis feature 67-5
 - for external objects 66-14
- Checks 4-6
- Child
 - definition Glossary-1
- Circles
 - code example 18-16
 - drawing 18-16
- CL commands 55-3
 - customizing 55-4
- Classes 10-2
 - external objects 66-3
 - notification 65-2
 - of foreign datum surfaces 68-2
- Classification of messages
 - critical 14-8
 - error 14-8
 - info 14-7
 - prompt 14-7
 - warning 14-8
- Clear
 - single animation 21-4
 - window 18-3
- Clearance
 - harness 58-17
- Close
 - windows 18-4
- Collect functions 1-56
- Collection
 - access from feature element trees 20-15
 - access from selection buffer 20-6
 - adding to the selection buffer 20-7
 - interactive 20-3
 - introduction 20-2
 - programmatic access 20-7
 - contents of curve collection 20-8
 - contents of surface collection 20-11
 - creation and modification of curve collections 20-10
 - creation and modification of surface collections 20-14
 - programmatic access to legacy collection 20-16
- Color

- changing 18-13
 - external objects 66-9
 - graphics 18-10
 - map
 - modifying 18-13
- Command
 - adding 15-20
- Commands
 - designating 15-20
 - Pro/ENGINEER
 - entering 15-69
 - preempting 15-53
- Compatibility
 - Pro/TOOLKIT and Pro/ENGINEER
 - versions 1-35
- Compiling 1-32
- Complement mode
 - definition Glossary-1
- Components
 - manufacturing
 - traversing 53-5
 - simplified representation
 - gathering 49-15
- Composite curves
 - geometry 5-29
 - visiting 5-14
- Compound element 29-5
- Compound menus 15-53
- Cone K-6
- Configuration file
 - definition Glossary-1
 - options
 - getting and setting 12-2
 - PROTKDAT option 1-33
 - toolkit_registry_file option 1-33
- Confirmation
 - using menu buttons 15-54
- Connectivity
 - cable 58-15
 - diagram 58-5
- Connectors
 - finding 58-9
 - parameters 58-9
 - file 58-10
- Constraints
 - case names 63-5
 - Convection 61-42
 - Displacement 61-43
 - FEM 63-7
 - section 40-5
 - status values 40-6
 - Symmetry 61-54
- Contacts 63-10
- Continuity
 - of foreign datum curves 68-6
- Contouring tools
 - parameters 53-11
- Contours
 - definition Glossary-2
 - internal or external 5-7
 - traversal J-2
- Conventional milling
 - required parameters 53-38
- Conversion
 - OHandles to DHandles 1-52
 - paths I-8
 - techniques I-4
- Convert
 - ProExt to ProModelitem 66-5
 - ProSelection to ProExtobj 66-11
 - toolkit applications I-2
- Coons patch K-10
- Coordinate System Transformations 8-4
- Coordinate systems 8-2
 - datum 8-4
 - visiting 5-10
 - definition Glossary-2
 - drawing 8-3
 - drawing view 8-3
 - in assemblies 8-3
 - screen 8-2
 - section 8-4
 - solid 8-2
 - window 8-3
- Copy
 - model 2-7
- Cosmetic properties 18-23
- Create
 - 2-D sections 40-2
 - code example 40-26
 - animation movies 21-5
 - animation objects 21-3
 - batch animation
 - example 21-6
 - compound menus 15-53
 - conventional milling sequence 53-41
 - cross sections 11-7
 - datum planes
 - code example 29-21
 - display lists 18-19
 - drawing views 51-24
 - external object classes 66-3
 - external object entities 66-7
 - external object references 66-13
 - external objects 66-3
 - code example 66-17
 - family table instances 9-5
 - features 29-1
 - file paths 12-6
 - fixtures 53-26
 - layers 2-14
 - local groups 4-19

- manufacturing features 53-25
- manufacturing objects 53-9
- manufacturing operations 53-33
- material 3-22
- material removal volumes 53-40
- menus 15-47
- NC sequences 53-36
 - elements 53-37
- operations
 - elements 53-33
- patterns 39-23
 - element tree 39-2
- process steps 56-3
- relation sets 6-3
- section models 40-2
- simplified representations 49-6
- solid objects 3-2
- submenus 15-55
- sweeps 43-1
- Task Libraries 69-3
- tool tables 53-28
- Toolkit DLL
 - Task Libraries 69-3
- tools 53-9
 - elements 53-13
- window 18-4
- workcells 53-27
 - elements 53-27
- Cross sections
 - creating and modifying 11-7
 - deleting 11-7
 - geometry of 11-3
 - listing 11-2
 - mass properties of 11-8
 - visiting 11-6
- CrvcollinstrAttributeSet() function
 - description 20-10
- Current
 - directory 12-3
 - drawing sheet 51-12
 - window 18-4
- Curves
 - data structures K-16
 - datum
 - parametric equations 5-22
 - visiting 5-13
 - definition Glossary-2
 - evaluating 5-18
 - foreign datum 68-3
- Customized plot driver 28-1
- Cut out 45-15
- Cylinders K-5
 - spline surfaces K-14
 - tabulated K-9

D

- Data
 - external object 66-6
 - handles
 - description 1-51
 - material 3-22
 - types I-7
- Data menus 15-56
- Data types
 - Pro/DEVELOP versus Pro/TOOLKIT I-7
- Database
 - search 54-3
- Database items
 - conversion paths I-8
- Datum
 - axes
 - visiting 5-9
 - coordinate systems
 - visiting 5-10
 - curves
 - geometry 5-29
 - parametric equations 5-22
 - visiting 5-13
 - planes
 - creating
 - code example 29-21
 - geometry 5-28
 - visiting 5-11
 - points
 - editing 5-31
 - geometry 5-29
 - selecting 5-31
 - visiting 5-14
 - surfaces
 - foreign 68-2
 - geometry 5-29
 - visiting 5-12
- Datum axis
 - creating 31-62
 - normal planes 31-61
 - point on surface 31-56
 - tangent 31-58
 - through edge or surface 31-59
 - two planes 31-59
 - two points 31-60
- Datum coordinate system
 - creating 31-77
 - feature element tree 31-68
 - orienting by selecting csys axes 31-75
 - orienting by selecting references 31-74
 - using 3Planes or 2 edges and axes 31-72
 - using a csys 31-75
 - using a vertex or a datum point 31-73
 - using curve, edges, or plane and axis 31-73

- Datum plane
 - creating 31-6
 - feature element tree 31-3
- Datum point
 - at an offset 31-47
 - at center of curve or surface 31-53
 - at intersection of 3 surfaces 31-50
 - at intersection of a curve and a surface 31-52
 - feature element tree 31-10
 - field 31-19
 - general 31-32
 - offset csys 31-24
 - on a vertex 31-47
 - on curve 31-54
 - on/offset from a Surface 31-51
 - sketched 31-10
- Debugging
 - applications 1-39
- Default values 14-9
 - constant 14-9
 - in text box 14-9
 - variable 14-10
- Delete
 - animation frames 21-4
 - animation objects 21-3
 - cross sections 11-7
 - datum points 5-31
 - display lists 18-19
 - external object classes 66-3
 - external objects 66-3
 - features 4-7, 45-13
 - layers 2-14
 - material data 3-22
 - menus 15-47
 - models 2-7
 - pattern 4-14
 - relation sets 6-3
 - section dimensions 40-13
 - section entities 40-9, 40-10
 - simplified representations 49-6
 - windows 18-4
- Deleting Cable Sections Cable
 - Delete 58-19
- Density 3-19
 - parts 3-19
- Deny
 - constraints 40-5
- Design intent
 - defined 46-2
- Design Manager 46-1
 - assembly component functions 46-5
 - assembly structure
 - design intent 46-3
 - populating 46-3
 - external reference data gathering functions
 - 46-9
 - external reference functions 46-6
 - external references 46-3
 - Overview 46-2
 - part interdependencies 46-3
 - product structure 46-2
 - skeleton model functions 46-5
 - skeleton models 46-2
- Designate
 - parameters 7-10
- Designating
 - command 15-21
 - commands 15-20
 - icon 15-21
- Designating commands 15-20
- Detail items 51-49
 - attachment points 51-91
 - leaders 51-91
- DHandles
 - description 1-51
- Diagram
 - item
 - color 58-3
 - line style 58-3
 - linestyle 58-3
- Diagrams
 - connectivity 58-5
 - listing diagram contents 58-2
 - parameters 58-4
 - wire spools 58-5
- Diameter
 - code example 5-23
- Digital Rights Management
 - error types H-2
 - interactive Pro/ENGINEER session H-5
 - introduction H-2
- Dimension
 - entity location 22-36
 - text 22-35
 - tolerances 22-32
- Dimension-driven patterns 39-6
- Dimensions 22-22
 - adding to a 2-D section 40-12
 - designating 22-21
 - driven
 - accessing 22-45
 - extracting location 22-36
 - feature 4-13
 - modifying 22-27
 - reference
 - accessing 22-45
 - section 40-12
 - types of 40-13
 - visiting 22-22
- Directories
 - changing 12-3

- Display
 - display lists 18-19
 - files 12-4
 - graphics 18-15
 - highlighting 19-10
 - messages 14-4
 - objects 18-10
 - solid objects 3-4
 - text 18-17
- Display data
 - allocating 66-7
 - color 66-9
 - for external objects 66-6
 - line styles 66-9
 - properties 66-8
 - scale 66-10
- Display lists 18-19
 - definition Glossary-2
- display mode argument 51-54
- Display modes 22-21
- Displayed entities
 - visiting 2-20
- Distance
 - gathering by 49-17
 - minimum 5-19
- DLL mode
 - overview 1-30
 - registry file for 1-34
- Documentation
 - see APIWizard 1-2
- Domain of evaluation K-2
- Draft
 - entities 51-49
 - definition Glossary-2
 - group
 - definition Glossary-2
 - groups 51-50
- Draft Feature
 - creation 35-8, 35-40
 - inquiring 35-8, 35-41
- Draw
 - graphics 18-16
- Drawing
 - circles and arcs 18-16
 - coordinate system 8-3
 - detail items 51-49
 - edges 51-90
 - display properties 51-90
 - ProDrawingEdgeDisplay 51-90
 - format 51-12
 - models 51-16
 - code example 51-25
 - rubber-band lines
 - example 18-21
 - sheets 51-11
 - example 51-13
 - symbol groups 51-87
 - transformations 8-9
 - views 51-16
 - code example 51-25
 - creating 51-24
- Drawing symbol groups 51-87
- Drawings
 - access grid locations 51-7
 - creating, from templates 51-2
 - errors diagnosing 51-4
 - setup 51-6
- DRM error types H-3
- DRM implications H-6

E

- ECAD Area Feature
 - ProEcadArea.h 44-2
- Edge and curve data structures K-16
- Edges
 - definition Glossary-2
 - evaluating 5-18
 - getting the description of 5-22
 - parametric equations 5-22
 - traversing J-2
 - types 5-22
- Edit
 - datum point arrays 5-31
 - files 12-4
- Edit Menu Features
 - Merge 33-11
- Element Tree
 - extruded features 42-2
 - revolved features 42-68
- Element tree
 - first features 42-106
- Element trees
 - auxiliary NC sequences 55-2
 - custom NC sequences 55-3
 - description 29-3
 - patterns 39-2
- Elements
 - definition Glossary-2
 - in an element tree 29-3
 - paths 29-11
 - roles 29-4
 - types 29-4
 - types of 29-4
 - values 29-5
- Enter
 - Pro/ENGINEER Mechanica Environment 61-3
- Entities
 - adding to 2-D sections 40-9
 - definition Glossary-3
- Epsilon

- specifying 40-8
- Equations
 - geometry 5-21
 - parametric 5-22
 - of surfaces 5-25
- Erase
 - family table instances 9-6
 - family tables 9-2
 - models 2-7
- Errors 1-19
 - section 40-24
- Evaluation
 - definition Glossary-3
 - domain of K-2
 - functions
 - for foreign datum curves 68-4
 - inverse 5-19
 - of faces, edges, and curves 5-18
 - of geometry 5-18
 - of relations 6-4
- EXample
 - weld callback notification 60-4
- Example
 - creating a datum axis 31-62
 - creating a datum coordinate system 31-77
 - creating a datum plane 31-6
 - creating a field datum point 31-20
 - creating a sketched datum point 31-12
 - creating an offset csys datum point 31-26
 - creating general datum point 31-37
 - offset coordinate system datum 67-11
- example 13-7, 13-11
- Examples
 - adding a menu button 15-46
 - adding help text 15-44
 - adding items to a layer 2-17, 2-21
 - adding surfaces to an element 53-39
 - adding to menu bar menus 15-17
 - asking for confirmation on Quit Window 15-54
 - batch mode 1-46
 - calculating the mass properties of a cross section 11-8
 - computing the outline of a solid 3-9, 3-19
 - creating a 2-axis lathe workcell 53-29
 - creating a batch animation 21-6
 - creating a conventional milling sequences 53-41
 - creating a datum plane 29-21
 - creating a menu that selects a value 15-51
 - creating a parameter tree 53-18, 53-22
 - creating a parameter-driven tool 53-15
 - creating a section 40-26
 - creating a sinusoidal foreign datum curve 68-7
 - creating a sweep 43-5
 - creating a sweep protrusion feature by
 - conventional approach 43-14
 - creating a tool from a solid model 53-13
 - creating an external object 66-17
 - creating an extruded feature 42-8
 - creating an operation 53-34
 - creating drawing views 51-25
 - defining a new menu that closes itself 15-48
 - defining a new menu that the user must close 15-49
 - designating a command 15-24
 - displaying a solid 3-4
 - displaying lines and circles 18-16
 - displaying messages 14-10
 - drawing a rubber-band line 18-21
 - extracting the diameter of an arc edge 5-23
 - finding the handle to a model 2-5
 - finding the position of a component 45-12
 - finding the surface penetrated by a hole 5-7
 - getting the angle of a conical surface 5-26
 - identifying workcell features of a Pro/NC model 53-6
 - inserting a feature 4-8
 - interference checking for assemblies and parts 5-35
 - labeling a feature with a string parameter 7-8
 - listing cross sections 11-4
 - listing the holes in a model 1-60
 - listing the members of an assembly 45-8
 - listing views 51-20
 - loading and displaying a solid 3-4
 - menu file 15-43
 - modifying colors 18-13
 - renaming a selected surface 2-11
 - reordering features 4-10
 - retrieving keyboard input 14-10
 - saving views 18-8
 - transforming solid coordinates 8-6
 - using a new menu 15-50
 - using drawing sheets 51-13
 - using expandable arrays 1-56
 - using the FEM functions 63-14
 - using the model item functions 2-11
 - using the plot driver functions 28-3
 - visiting the items in a simplified representation 49-10
 - writing a family table to a file 9-7
- Exit actions
 - defining 15-48
- Expandable arrays 1-53
 - allocating 1-54
 - code example 1-56
 - freeing 1-54

- Explode states
 - access 45-15
 - activating 45-15
 - visiting 45-14
- Exploded assemblies 45-14
- Export
 - 2D Models 26-5
 - 3D Models 26-21
 - FEA mesh 63-2
 - information files 26-2
 - Shrinkwrap Models 26-31
 - To PDF format 26-37
- External analysis
 - attributes
 - get and set 67-27
 - PROANALYSIS_COMPUTE_OFF 67-27
 - callbacks 67-5
 - defined 67-2
 - do not use global variables with 67-8
 - entity shape union 67-10
 - entity type enum 67-10
 - feature parameter structure 67-9
 - geometry item structure 67-9
 - interactive creation 67-3
 - use without Pro/ENGINEER 67-28
- External analysis feature
 - defined 67-2
 - interactive creation 67-3
 - storage as feature dimensions or geometry
 - references 67-4
 - use without Pro/ENGINEER 67-28
 - when stored as external data 67-4
- External data 10-1
 - classes 10-2
 - retrieving 10-5
 - slots 10-2
 - storing 10-4
- External databases
 - query 54-1
- External objects
 - callbacks 66-14
 - classes 66-3
 - color 66-9
 - creating entities 66-7
 - data 66-6
 - manipulating 66-11
 - display data 66-6
 - allocating 66-7
 - properties 66-8
 - displaying 66-7
 - identifiers 66-5
 - information for 66-2
 - line styles 66-9
 - parameters 66-5
 - recycling identifiers 66-4

- references 66-12
 - creating 66-13
 - types 66-13
 - visiting 66-13
 - scale 66-10
 - selection data 66-10
 - summary 66-2
 - transformation 66-8
 - types 66-5
 - visiting 66-6
 - warning mechanism 66-16
- External references
 - defined 46-4

F

- Face milling
 - required parameters 53-38
- Faces
 - definition Glossary-3
 - evaluating 5-18
 - traversal J-2
- Family tables
 - editing 9-2
 - instances
 - creating 9-5
 - erasing 9-6
 - generic 9-6
 - locks 9-6
 - operations 9-4
 - retrieving 9-6
 - items
 - from model items 9-7
 - from parameters 9-7
 - operations on 9-7
 - objects 9-2
 - showing 9-2
 - utilities 9-2
 - visiting 9-3
 - writing to a file 9-7
- Feature
 - extruded
 - creating 42-8
- Feature element tree
 - datum axis 31-56
 - datum coordinate system 31-68
 - datum plane 31-2
 - datum point 31-10
 - field datum point 31-19
 - general datum point 31-34
 - merge 33-12
 - offset csys datum point 31-25
 - sketched datum point 31-11
- Feature Element Tree for the Sheetmetal Flat Wall Feature in Pro/ENGINEER

- 52-10
- Features
 - Auto Round 36-21
 - chamfer 36-22
 - access 36-30
 - create 36-29
 - feature element tree 36-22
 - redefine 36-30
 - children 4-5
 - converting from Pro/DEVELOP to Pro/TOOLKIT I-10
 - corner chamfer 36-46
 - access 36-47
 - create 36-47
 - feature element tree 36-46
 - redefine 36-47
 - creating 29-1
 - assembly components by 47-17
 - code example 29-21
 - datums 29-21
 - steps 29-2
 - datum coordinate system 31-68
 - datum plane 31-2
 - datum point 31-10
 - deleting 4-7, 45-13
 - dimensions 4-13
 - draft
 - access 35-8, 35-41
 - create 35-8, 35-40
 - feature element tree 35-2
 - introduction 35-2
 - redefine 35-8, 35-40
 - ECAD Area 44-2
 - feature element tree 44-2
 - ProEcadArea.h 44-2
 - element table 29-5
 - element tree 29-5
 - element values 29-5
 - elements
 - creating 29-13
 - manipulating 29-13
 - paths 29-11
 - values 29-9
 - fill 33-8
 - access 33-10
 - create 33-10
 - feature element tree 33-8
 - redefine 33-10
 - geometry 4-6
 - visiting 5-4
 - header files for 29-6
 - incomplete 29-20
 - description 4-5

- inquiry 4-3
 - detailed explanation 29-26
- intersect 33-11
- manipulating 4-7
- manufacturing
 - analyzing 53-45
 - creating 53-25
- Merge 33-11
 - access 33-14
 - create 33-13
 - feature element tree 33-12
 - ProMerge.h 33-12
 - redefine 33-13
- merge
 - access 33-14
- mirror 33-2
 - access 33-4
 - create 33-4
 - feature element tree 33-3
 - redefine 33-4
- move 33-4
 - access 33-8
 - create 33-7
 - feature element tree 33-6
 - redefine 33-7
- objects 4-2
- offset 33-21
- parents 4-5
- pattern 33-14
- patterns 4-13
 - creating 39-2
- process steps 56-2
- redefining 4-8
- Remove 33-27
 - access 33-31
 - create 33-31
 - feature element tree 33-28
 - ProRemoveSurf.h 33-28
 - redefine 33-31
- reordering 4-10
- round 36-2
 - access 36-13
 - create 36-12
 - feature element tree 36-2
 - redefine 36-13
- selecting 4-5
- Shell
 - access 38-5
 - create 38-5
 - feature element tree 38-3
 - introduction 38-2
 - ProShell.h 38-3
 - redefine 38-5

- solidify 33-24
 - access 33-27, 33-31
 - create 33-26
 - feature element tree 33-24
 - redefine 33-27, 33-31
- status 4-4
- suppressing 4-7
- sweeps 43-1
- thicken 33-21
 - access 33-24
 - create 33-23
 - feature element tree 33-22
 - redefine 33-23
- trim 33-14
 - access 33-21
 - create 33-20
 - feature element tree 33-15
 - redefine 33-21
- tweak surface replacement 34-1
- types 4-3
- user-defined 4-20
- visiting 4-2
- workcell
 - identifying 53-6
- wrap 33-14
- FEM**
 - exporting an FEA mesh 63-2
 - getting bar elements 63-10
 - getting constraint case names 63-5
 - getting constraints 63-7
 - getting contacts 63-10
 - getting mass elements 63-11
 - getting mesh controls 63-5
 - getting shell pairs 63-12
 - overview of functionality 63-2
- Files**
 - connector parameters 58-10
 - displaying 12-4
 - editing 12-4
 - include 1-41
 - management
 - operations 2-7
 - material 3-27
 - maximum length 15-42
 - menu 15-42
 - message 14-5
 - naming restrictions 14-5
 - opening 12-6
 - parsing 12-6
 - trail 12-3
 - writing a family table to 9-7
- Fillet surfaces K-10
- Filter functions
 - for visits 1-59
- Find
 - APIWizard search feature (Find) 1-13
- Fixtures
 - creating 53-26
- Fonts 18-18
- Foreign datum curves 68-3
 - continuity 68-6
 - evaluation function for
 - binding 68-7
 - evaluation functions for 68-4
 - example 68-7
- Foreign datum surfaces 68-2
 - importing 68-2
- Foreign programs
 - running D-4
 - multiple D-4
- Frames
 - display frame in APIWizard 1-6
 - in the APIWizard online browser 1-6
 - topic/object selection 1-6
- Free
 - external object data 66-11
 - external object references 66-13
 - text style structure 22-52
 - version stamp 2-13
- Full asynchronous mode 13-9, 13-11
- Functions
 - action 1-59
 - browsing with APIWizard 1-11
 - collect 1-56
 - comparing toolkits I-4
 - error statuses 1-19
 - filter 1-59
 - list of equivalent I-16
 - prototyping 1-19
 - using expandable arrays 1-55
 - visit 1-59

G

- Gather
 - by distance 49-17
 - by model name 49-16
 - by parameters 49-16
 - by rule 49-15
 - by simplified representation 49-18
 - by size 49-17
 - by zone 49-17
- General process steps 56-6
- General surface of revolution K-7
- Generic instances 9-6
- Geometry
 - cables 58-17
 - checks 4-6
 - composite curves 5-29
 - coordinate system datums 5-27
 - cross-sectional 11-3

- datum axes 5-27
 - datum curves 5-29
 - datum planes 5-28
 - datum points 5-29
 - datum surfaces 5-29
 - equations 5-21
 - evaluating 5-18
 - feature
 - visiting 5-4
 - measurement of 5-32
 - NURBS 5-34
 - objects 5-2
 - visiting 5-3
 - of solid edges 5-22
 - of surfaces 5-25
 - quilts 5-28
 - representations K-1
 - solid
 - visiting 5-6
 - terms J-2
 - traversal J-1
 - Graphics
 - color and line styles 18-10
 - color map 18-13
 - displaying 18-15
 - line styles
 - setting 18-15
 - surviving a repaint 18-20
 - Groups 4-20
 - drawing symbol 51-87
 - identifying symbol definition 51-88
 - Identifying symbol instance 51-88
 - local 4-19
 - manipulating symbols 51-89
 - read access 4-20
 - gtol
 - additional text 23-17
 - creation 23-7
 - deleting 23-16
 - editing 23-15
 - backup made by ProGtolEditInit 23-15
 - cancellation 23-15
 - live in Pro/ENGINEER session 23-15
 - save
 - changed data with ProGtolEditCommit 23-15
 - handle
 - type 23-2
 - layout 23-16
 - parameters 23-19
 - placement
 - set 23-9
 - prefix 23-18
 - ProGtolleader defined 23-9
 - reading 23-4
 - reference
 - set 23-9
 - required function sequence 23-8, 23-16
 - return type 23-4
 - set functions return ProGtoldatastatus
 - type 23-9
 - suffix 23-18
 - text style properties 23-18
 - visiting 23-2
- ## H
- handle
 - type for gtol 23-2
 - Handles
 - data 1-51
 - description 1-49
 - object 1-49
 - Pro/DEVELOP versus Pro/TOOLKIT I-7
 - workspace 1-52
 - Hardware type
 - setting 1-65
 - Harnesses
 - clearance 58-17
 - connectors 58-9
 - Height
 - text 18-18
 - Highlight 19-9
 - definition Glossary-3
 - menu buttons 15-57
 - Highlighting
 - functions described 19-9
 - Holemaking
 - required parameters 53-38
 - tool parameters 53-12
 - Holes
 - code example 1-60
 - Hot spots 66-10
- ## I
- Identifiers
 - cable 58-14
 - external object 66-5
 - postfix 3-18
 - recycling 66-4
 - Identifying
 - symbol groups
 - definition 51-88
 - instance 51-88
 - Identity matrix
 - used with external objects 66-8
 - Import
 - 2D Models 26-48
 - 3D Models 26-48
 - files 26-46
 - foreign datum surfaces 68-2
 - Parameter Files 26-46

- Include files 1-41
- Incomplete features
 - description 4-5
- Information window
 - definition Glossary-3
 - display function 12-4
- Initialize
 - assembly component paths 45-5
 - family tables 9-2
 - process step features 56-2
 - single animation 21-4
 - tool 53-13
- Input
 - default values 14-9
 - keyboard 14-9
 - mouse 18-20
- Inquiry
 - features 4-3
- Insert
 - datum points 5-31
- Insert mode 4-8
- Install
 - Pro/TOOLKIT 1-20
 - See Pro/TOOLKIT Installing 1-20
- Installation
 - See Pro/TOOLKIT Installing 1-20
- Installing, sample applications F-2
- Instances
 - identifying symbol groups 51-88
- Interactive selection 19-5
- Interchange domain 45-16
- Interface
 - tools 28-1
- Interference 5-35
- Internal buffer
 - writing a message to 14-8
- Internet Explorer
 - Java2 plug in 1-3, 1-5
 - requirements 1-3, 1-5
- Inverse evaluation 5-19
- ISO/DIN Tolerance, use 22-35

J

- Java2 plugin
 - not recommended for APIWizard with Netscape Navigator 1-5
 - required for APIWizard with Internet Explorer 1-3, 1-5, 1-6

K

- Keyboard
 - input 15-73
 - default values 14-9
 - getting 14-9
 - macros

- execution rules 15-70

L

- Launch
 - Synchronous J-Link Applications 69-17
 - Toolkit DLL Functions 69-8
- Layers 2-13
 - creating 2-14
 - deleting 2-14
 - getting 2-14
 - items 2-15
 - view dependency 2-16
 - visiting 2-14
- Layouts 2-20
- Leaders
 - definition in glossary Glossary-3
 - patterns 4-14
- Legacy Encoding
 - definition B-2
- Libraries
 - alternate E-2
 - MT, for Windows E-2
 - PIC, for UNIX E-3
 - standard E-2
- License data 12-3
- Light sources 18-25
- Line
 - styles
 - of external objects 66-9
- Lines
 - code example 18-16
 - drawing 18-16
 - representation K-17
 - styles 18-10
 - setting 18-15
 - types 18-15
- Linking 1-32
- Load directory
 - definition Glossary-3
- Loads
 - Bearing 61-30
 - Centrifugal 61-32
 - Force and Moment 61-29
 - Gravity 61-30
 - Heat 61-37
 - Mechanica 61-20
 - Pressure 61-30
 - Temperature 61-32
 - External 61-36
 - Global 61-34
 - MEC/T 61-35
 - Mechanica 61-33
 - Structural 61-33
- Loads and Constraints

- Mechanica 61-15
- Local groups 4-19
- Locations
 - of assembly components 45-11
 - routing 58-16
- Locks 9-6

M

- Macro keys
 - definition Glossary-3
- main()
 - user-supplied 1-48
- Makefiles
 - description of 1-32
- Management
 - Argument 69-2
 - Memory 69-4
- Manipulating symbol groups 51-89
- Manufacturing
 - analyzing a model 53-3
 - components
 - roles 53-5
 - traversing 53-5
 - features
 - analyzing 53-45
 - creating 53-25
 - fixtures 53-26
 - models
 - create 53-3
 - types 53-2
 - NC sequences 53-36
 - objects
 - creating 53-9
 - operations
 - creating 53-33
 - parameters 53-16
 - roles 53-5
 - storage solids 53-3
 - identifying 53-6
 - tools
 - creating 53-9
 - types 53-8
 - visiting 53-8
 - workcells
 - creating 53-27
- Mass elements
 - getting 63-11
- Mass properties 3-18
 - definition Glossary-4
 - example 11-8
 - of cross sections 11-7
- Materials
 - data
 - accessing 3-22

- files 3-27
- properties 3-21
- removal volumes
 - creating 53-40
- Matrix
 - definition Glossary-4
 - pan and zoom 18-6
 - view 18-6
 - window 18-6
- Maximum length
 - of files 15-42
- Measurement 5-32
- Mechanica
 - Accessing
 - AutoGEM Maximum Element Size Mesh Control Data 61-115
 - Displacement Coordinate System Data 61-118
 - Edge Distribution Mesh Control Data 61-116
 - Mesh Control Element Size Data 61-119
 - Mesh Control Hard Point Data 61-119
 - Mesh Control ID Offset Data 61-120
 - Mesh Control Numbering Data 61-120
 - Mesh Control Shell Coordinate System Data 61-119
 - Suppressed Mesh Control Data 61-121
 - AutoGEM Edge Distribution 61-113
 - Beam Orientations 61-68
 - Beam Releases 61-70
 - Beam Section
 - General 61-66
 - Sketched 61-65
 - Beam Sections 61-60
 - Beams 61-56
 - Constraint Sets 61-56
 - Constraints 61-40
 - Environment, Entering 61-3
 - Functions 61-12
 - Gaps 61-106
 - Geometric References 61-8
 - Interfaces 61-99
 - Load Sets 61-39
 - Loads 61-20
 - Mass
 - Properties 61-77
 - Mass Items 61-74
 - Material Orientations 61-78
 - Mesh Control 61-108
 - Minimum Edge Mesh Control Data 61-113
 - Shell Pairs 61-96
 - Shell Properties 61-88
 - Shells 61-82
 - Spring Items 61-71

- Spring Property Items 61-73
- Welds 61-121
- Y-directions 61-11
- Mechanica Items
 - Accessing 61-6
 - Selection 61-4
- Memory
 - Management
 - Task Library Functions 69-4
 - maximum allocated by ProArrayAlloc 1-54
- Menu Bars
 - pushbutton
 - adding 15-5
- Menu bars
 - adding a menu 15-15
 - adding buttons 15-8
 - buttons 15-2
 - code example 15-17
 - menus 15-2
- Menus
 - adding to menu bar 15-15
 - buttons
 - accessibility 15-58
 - adding 15-44
 - example 15-46
 - locations 15-56
 - setting 15-57
 - compound
 - creating 15-53
 - creating
 - exit actions 15-47
 - for selecting a single value 15-51
 - preempting existing commands 15-53
 - data menus 15-56
 - definition Glossary-4
 - exit actions 15-47
 - files
 - definition Glossary-4
 - names and contents 15-42
 - purpose 15-42
 - sample 15-43
 - submenus 15-55
 - syntax 15-42
 - toolkit I-7
 - main
 - definition Glossary-3
 - manipulating 15-56
 - accessibility of buttons 15-58
 - data menus 15-56
 - setting buttons 15-57
 - mode-specific 15-41
 - new
 - creating 15-47
 - defining 15-48
 - using 15-49
 - pushing and popping 15-58
 - run-time 15-59
 - submenus 15-55
 - definition Glossary-6
- Merge 45-15
- Merge Feature 33-11
 - ProMerge.h 33-12
- Mesh controls
 - getting 63-5
- Message
 - classification of 14-7
- Message file
 - definition Glossary-4
 - restrictions 14-5
- Messages
 - classification of
 - critical 14-8
 - error 14-8
 - info 14-7
 - prompt 14-7
 - warning 14-8
 - file 14-5
 - files
 - toolkit I-7
- Milling tools
 - parameters 53-11
- Model coordinates
 - definition Glossary-4
- Model items
 - definition Glossary-4
 - description 2-8
 - from family table items 9-7
 - names 2-10
- ModelCHECK 12-7
 - running 12-7
- Models 2-2
 - definition Glossary-4
 - drawing 51-47
 - file management operations 2-7
 - finding the handle
 - example 2-5
 - identifying 2-4
 - in session 2-6
 - names
 - gathering by 49-16
 - orientation 18-6
 - section 40-2
- Modes 1-30
 - complement
 - definition Glossary-1
 - description 2-2
 - set
 - definition Glossary-6
- Mode-specific menus 15-41

- Modify
 - colors
 - example 18-13
 - cross sections 11-7
 - simplified representations 49-14
- Mouse
 - input 18-20
 - positions
 - example 18-21
- Movies
 - animation 21-5
- Multibyte String
 - definition B-2
- Multiprocess mode
 - overview 1-30
 - registry file for 1-35
- Multi-Threaded (MT) DLL Libraries E-2
- Multivalued element 29-5

N

- Names
 - of foreign datum surfaces 68-2
 - simplified representations 49-9
- Naming conventions 1-17
- Native Encoding
 - definition B-2
- NC sequences
 - auxiliary and custom 55-2
 - creating 53-36
 - determining 4-5
- Netscape Navigator
 - requirements 1-3
 - Swing required 1-3
- Notes 22-50
 - detail items 51-50
 - Properties 22-50
 - text styles 22-52
 - visiting 22-52
- Notify
 - classes 65-2
 - definition Glossary-5
- NURBS 5-34
 - representation K-18
 - surface K-13

O

- Object
 - ProDtlitem 51-50
- Object handle
 - ProDtlentity 51-50
- Object handles
 - ProDtlentity 51-50
 - ProDtlgroup 51-51
 - ProDtlnote 51-50
 - ProDtlsymdef 51-51

- ProDtlsyminst 51-51
- Object, ProDimension 22-22
- Objects
 - browsing with APIWizard 1-10
 - definition Glossary-5
 - displaying 18-10
 - external 66-2
 - handles 1-49
 - naming conventions 1-17
 - ProSelection
 - defined 19-2
 - selecting 19-2
- OHandles
 - converting to DHandles 1-52
 - description 1-50
 - ProView 51-16
- Opaque pointers 1-50
- Operations
 - creating 53-33
 - example 53-34
- Options
 - configuration file
 - getting and setting 12-2
- Orientation
 - model 18-6
- Outlines
 - code example 3-9, 3-19
 - of solids 3-8
- Overlay views
 - definition Glossary-5
- Owners
 - external object 66-4
 - window 18-5

P

- Pan and zoom matrix 18-6
- Parameters
 - accessing 7-4
 - code example 7-8
 - connector 58-9
 - conventional milling
 - required 53-38
 - creating a parameter tree 53-18, 53-22
 - definition Glossary-5
 - designating for Pro/INTRALINK 7-10
 - diagram 58-4
 - external objects 66-5
 - for auxiliary tools 53-11
 - for contouring tools 53-11
 - for face milling 53-38
 - for holmaking tools 53-12
 - for milling tools 53-11
 - for turning tools 53-10
 - from family table items 9-7
 - gathering by 49-16

- holemaking
 - required 53-38
- manufacturing 53-16
- utility functions 7-3
- values 7-3
- Parent
 - definition Glossary-5
- Parse
 - file names 12-6
- Parts 3-19
 - conversion paths I-11
 - density 3-19
 - interference checking 5-35
 - material properties 3-21
 - postfix identifiers 3-18
 - traversing 5-3
- Paths
 - feature element 29-11
- Pattern
 - axis patterns 39-15
- Patterns 4-13
 - creating 39-2
 - example 39-23
 - curve patterns 39-17
 - dimension patterns 39-6
 - direction patterns 39-12
 - element tree 39-2
 - getting 39-22
 - fill patterns 39-10
 - Fill type 4-13
 - leaders 4-14
 - manipulating 4-13
 - point patterns 39-18
 - reference patterns 39-6
 - regeneration types 39-5
 - table-driven
 - creating 39-8
 - types 39-5
- patterns
 - table patterns 39-8
- Placing
 - toolbar 15-22
 - toolbar button 15-22
- Placing toolbar button 15-22
- Planes K-4
- Play
 - single animation 21-4
- Plot driver 28-1
- Points
 - datum 5-14
- Polygons 18-15
- Polylines 18-16
- Popup Menu
 - Adding to the Graphics Window 15-31
 - Checking access state 15-34
 - Using Trail files to determine names 15-32
- Popup Menus 15-31
 - Accessing 15-33
 - Creating commands for new buttons 15-34
 - Registering Notifications to create and
 - destroy menus 15-33
- Popup menus
 - Adding 15-35
- Position Independent Code (PIC) Libraries
 - E-3
- Postfix identifiers 3-18
- Preempt
 - Pro/ENGINEER commands 15-53
- Pro/DEVELOP
 - conversion paths I-8
 - converting from I-2
 - converting to Pro/TOOLKIT I-2
 - enumerated types I-12
 - functions
 - compared to Pro/TOOLKIT I-4
 - list of equivalent functions I-16
 - relationship with Pro/TOOLKIT I-2
- Pro/ENGINEER
 - commands
 - entering 15-69
 - preempting 15-53
 - connecting to a process 13-5
 - license data 12-3
 - process status 13-9
 - starting and stopping 13-4
- Pro/INTRALINK
 - designating parameters for 7-10
- Pro/MESH
 - functionality 63-2
- Pro/TOOLKIT
 - actions 1-17
 - application structure I-7
 - converting from Pro/DEVELOP I-2
 - coordinate systems 8-2
 - core applications 1-43
 - documentation
 - online (APIWizard) 1-2
 - PDF format 1-2
 - expandable arrays 1-55
 - functions
 - compared to Pro/DEVELOP I-4
 - include files 1-41
 - Installation 1-23
 - installing 1-20
 - add or update 1-23
 - loadpoint directories 1-20
 - test of 1-23
 - version compatibility 1-35
 - models 2-2
 - objects 1-17
 - registry file

- compared to Pro/DEVELOP I-7
 - terminology
 - contrasted with Pro/DEVELOP I-4
 - using a batch session 1-46
 - utility functions 12-1
- Pro/Toolkit DLL Functions
 - Launch 69-8
- pro_change_dir() function
 - equivalent I-22
- pro_clear_window() function
 - equivalent I-18
- pro_close_object_window() function
 - equivalent I-18
- pro_compute_clearance() function 5-35
 - assembly members 45-15
 - used in a code example 5-35
- pro_compute_global_interference() function
 - description 5-35
- pro_compute_interference() function
 - description 5-35
- pro_compute_volume() function
 - used in a code example 5-35
- pro_create_2d_disp_list() function
 - equivalent I-19
- pro_create_3d_disp_list() function
 - description I-19
- pro_delete_2d_disp_list() function
 - description I-19
- pro_delete_3d_disp_list() function
 - equivalent I-20
- pro_display_2d_disp_list() function
 - description I-19
- pro_display_3d_disp_list() function
 - description I-19
- pro_display_intf_volume() function
 - used in a code example 5-35
- pro_dist_manifolds() function
 - description 5-35
- PRO_DRAWING_WELD_GROUPIDS_GET
 - weld symbol notification type 60-3
- PRO_DRAWING_WELD_SYMPATH_GET
 - weld symbol notification type 60-3
- PRO_DRAWING_WELD_SYMTEXT_GET
 - weld symbol notification type 60-3
- pro_dtmpntarr_add_pnt() function
 - description 5-31
- pro_dtmpntarr_delete_pnt() function
 - description 5-31
- pro_dtmpntarr_insert_pnt() function
 - description 5-31
- PRO_E_CURVE_CONTINUITY element 68-6
- pro_edge_length() function
 - equivalent I-24
- pro_element_info() function
 - equivalent I-21
- pro_eval_uv_edge() function
 - equivalent I-23
- pro_eval_xyz_edge() function
 - equivalent I-23
- pro_eval_xyz_entity() function
 - equivalent I-23
- pro_eval_xyz_face() function
 - equivalent I-23
- pro_export_fea_mesh() function
 - description 63-5
 - equivalent I-22
- pro_export_file_from_pro() function
 - equivalent I-22
- pro_export_plot_file() function
 - equivalent I-22
- pro_face_area() function
 - equivalent I-24
- pro_face_extremes() function
 - equivalent I-24
- PRO_FEAT_ASSEM_CUT 45-2
- PRO_FEAT_COMPONENT 45-2
- Pro_fem_bar_element structure 63-10
- Pro_fem_con_value structure 63-10
- Pro_fem_constraint structure 63-8
- Pro_fem_contact structure 63-10
- Pro_fem_mass_element structure 63-11
- Pro_fem_mass_props structure 63-12
- Pro_fem_mesh_control structure 63-5
- Pro_fem_region structure 63-6
- Pro_fem_shell_pair structure 63-12
- pro_get_config() function
 - equivalent I-22
- pro_get_cur_window_matrix() function
 - equivalent I-18
- pro_get_current_directory() function
 - equivalent I-22
- pro_get_current_mode() function
 - equivalent I-20
- pro_get_current_object() function
 - equivalent I-20
- pro_get_current_window() function
 - equivalent I-18
- pro_get_curve_type_geom() function
 - equivalent I-23
- pro_get_edge_param() function
 - equivalent I-23
- pro_get_entity_param() function
 - equivalent I-23
- pro_get_face_params() function
 - equivalent I-23
- pro_get_light_sources() function
 - description 18-25
- pro_get_prodevdat_info() function
 - equivalent I-21
- pro_get_selection() function
 - equivalent I-21
 - guidelines I-15

pro_get_text_attributes() function
 equivalent I-19
 pro_get_view_matrix() function
 equivalent I-18
 pro_get_view_names() function
 equivalent I-18
 pro_getbox() function
 equivalent I-19
 pro_getenvironment() function
 equivalent I-22
 pro_interference_volume_release() function
 description 5-35
 pro_is_option_ordered() function
 equivalent I-22
 Pro_light structure 18-26
 pro_on_regenerate_end() function
 equivalent I-26
 pro_open_object_window() function
 equivalent I-18
 Pro_Pattern structure
 declaration 4-15
 Pro_pattern_dim structure
 declaration 4-17
 Pro_pattern_dir structure
 declaration 4-16
 Pro_pattern_option enum
 declaration 4-15
 Pro_pattern_type enum
 declaration 4-15
 pro_point_on_geom() function
 equivalent I-23
 pro_ray_x_model() function
 equivalent I-21
 pro_read_file_to_pro() function
 equivalent I-22
 pro_refresh_window() function
 equivalent I-18
 pro_regenerate() function
 equivalent I-21
 pro_regenerate_object() function
 and pattern tables 4-19
 equivalent I-21
 pro_release_memb_id_tab() function
 description 58-9
 pro_reset_view() function
 equivalent I-18
 pro_retrieve_view() function
 equivalent I-18
 pro_rotate_view() function
 equivalent I-18
 pro_sample_xy() function
 equivalent I-19
 pro_select() function
 equivalent I-21
 guidelines I-15
 options I-14
 pro_select_strings() function
 equivalent I-17
 pro_set_and_get_selection() function
 equivalent I-21
 guidelines I-15
 pro_set_config() function
 equivalent I-22
 pro_set_current_window() function
 equivalent I-18
 pro_set_light_sources() function
 description 18-25
 pro_set_text_attributes() function
 equivalent I-19
 pro_set_view_matrix() function
 equivalent I-18
 pro_show_file() function
 equivalent I-21
 pro_show_info_window() function
 equivalent I-21
 pro_show_select() function
 equivalent I-18
 guidelines I-15
 pro_solid_id_from_ptr() function
 equivalent I-20
 pro_solid_ptr_from_id() function
 equivalent I-20
 pro_store_view() function
 equivalent I-18
 pro_str_to_wstr() function
 equivalent I-21
 Pro_surf_props structure 18-24
 pro_term() function
 equivalent I-16
 pro_vectors_to_transf() function
 equivalent I-26
 pro_view_repaint() function
 equivalent I-18
 pro_wchar_t_check() function
 equivalent I-21
 pro_wstr_to_str() function
 equivalent I-21
 Pro2dExport function
 description 26-6
 Pro2dExportdataAlloc() function
 description 26-12
 Pro2dExportdataFree() function
 description 26-12
 Pro2dExportdataModelspacesheetSet()
 function
 description 26-12
 Pro2dExportdataSheetoptionSet() function
 description 26-12
 Pro2dExportdataSheetsSet() function
 description 26-12
 Pro2dImportAppend function
 description 26-48

Pro2dImportCreate function
 description 26-48
 ProAccessorywindowWithTreeCreate()
 function
 description 18-5
 ProAnalysisAttrIsSet() function
 defined 67-27
 ProAnalysisAttrSet() function
 defined 67-27
 ProAnalysisInfoGet function
 defined 67-8
 ProAnalysisInfoSet function
 defined 67-8
 ProAnalysisNameGet function
 defined 67-28
 ProAnalysisTypeGet function
 defined 67-8
 ProAnalysisTypeRegister function
 call from user initialize 67-5
 defined 67-5
 ProAnimframeCreate() function
 description 21-3
 used in a code example 21-8
 ProAnimframeDelete() function
 description 21-4
 ProAnimframeObjAdd() function
 description 21-3
 used in a code example 21-8
 ProAnimframeObjRemove() function
 description 21-4
 ProAnimframeObjVisit() function
 description 21-4
 ProAnimmovieCreate() function
 description 21-5
 used in a code example 21-8
 ProAnimmovieDelete() function
 description 21-5
 ProAnimmovieFrameAdd() function
 description 21-5
 used in a code example 21-8
 ProAnimmovieFrameRemove() function
 description 21-5
 ProAnimmovieFrameVisit() function
 description 21-5
 ProAnimobjectCreate() function
 description 21-3
 used in a code example 21-8
 ProAnimobjectDelete() function
 description 21-3
 ProAnnotationAttachmentUpdate() function
 description 22-20
 ProAnnotationDisplay() function
 description 22-16
 ProAnnotationelemAnnotationGet() function
 description 22-6
 ProAnnotationelemAnnotationSet() function

 description 22-10
 ProAnnotationelemArrayReferencesSet()
 function
 description 22-10
 ProAnnotationelemAutopropagate() function
 description 22-12
 ProAnnotationelemCopyGet() function
 description 22-6
 ProAnnotationelemCopySet() function
 description 22-10
 ProAnnotationelemDependencySet() function
 description 22-10
 ProAnnotationElementGet() function
 description 22-15
 ProAnnotationelemFeatureGet() function
 description 22-7
 ProAnnotationelemFilterAction() function
 description 22-6
 ProAnnotationelemHasMissingrefs() function
 description 22-9
 ProAnnotationelemIsDependent() function
 description 22-7
 ProAnnotationelemIsIncomplete() function
 description 22-7
 ProAnnotationelemQuiltreferenceSurfacesCol
 lect() function
 description 22-7
 ProAnnotationelemReferenceAdd() function
 description 22-10
 ProAnnotationelemReferenceAutopropagateG
 et() function
 description 22-8
 ProAnnotationelemReferenceAutopropagateS
 et 22-10
 ProAnnotationelemReferenceDescriptionGet()
 function
 description 22-8
 ProAnnotationelemReferenceIsStrong()
 function
 description 22-9
 ProAnnotationelemReferenceRemove()
 function
 description 22-10
 ProAnnotationelemReferencesCollect()
 function
 description 22-7
 ProAnnotationelemReferencesSet() function
 description 22-10
 ProAnnotationelemReferenceStrengthen()
 function
 description 22-10
 ProAnnotationelemReferenceWeaken()
 function
 description 22-10
 ProAnnotationelemTypeGet() function
 description 22-8

ProAnnotationErase() function
 description 22-16
 ProAnnotationfeatCreate() function
 description 22-4
 ProAnnotationfeatElementAdd() function
 description 22-5
 ProAnnotationfeatElementArrayAdd()
 function
 description 22-5
 ProAnnotationfeatElementDelete() function
 description 22-5, 22-47
 ProAnnotationIsAssociative() function
 description 22-20
 ProAnnotationIsInactive() function
 description 22-16
 ProAnnotationIsShown() function
 description 22-16
 ProAnnotationplaneAngleGet() function
 description 22-19
 ProAnnotationplaneCreate() function
 description 22-17
 ProAnnotationplaneFlatToScreenCreate()
 function
 description 22-18
 ProAnnotationplaneForcetoplaneflagGet()
 function
 description 22-19
 ProAnnotationplaneForcetoplaneflagSet()
 function
 description 22-19
 ProAnnotationplaneFromViewCreate()
 function
 description 22-17
 ProAnnotationplaneFrozenGet() function
 description 22-18
 ProAnnotationplaneFrozenSet() function
 description 22-18
 ProAnnotationplanePlaneGet() function
 description 22-18
 ProAnnotationplaneReferenceGet() function
 description 22-18
 ProAnnotationplaneTypeGet() function
 description 22-18
 ProAnnotationplaneVectorGet() function
 description 22-19
 ProAnnotationplaneViewnameGet() function
 description 22-19
 ProAnnotationPositionUpdate() function
 description 22-20
 ProAnnotationRotate() function
 description 22-19
 ProAnnotationShow() function
 description 22-16
 ProAnnotationUndisplay() function
 description 22-17
 ProAnnotationUpdate() function
 description 22-16
 ProAnnotatonDisplay() 22-16
 ProAppData data type
 declaration 1-59
 proappdata_create_data() function
 equivalent I-22
 proappdata_delete_data() function
 equivalent I-22
 proappdata_list_classes() function
 equivalent I-22
 proappdata_list_data_in_class() function
 equivalent I-22
 proappdata_read_data() function
 equivalent I-22
 proappdata_register_class() function
 equivalent I-22
 proappdata_write_data() function
 equivalent I-22
 ProArcdataInit() function
 description 27-10
 ProArgument
 Argument Management 69-2
 description 69-2
 ProArgumentByLabelGet() function
 description 69-2
 ProArgumentProarrayFree() function
 description 69-3
 ProArrayAlloc
 maximum allocated memory 1-54
 ProArrayAlloc() function
 description 1-54
 used with material data 3-22
 with element paths 29-12
 ProArrayFree() function
 description 1-54
 ProArrayObjectAdd() function
 description 1-55
 ProArrayObjectRemove() function
 description 1-55
 ProArraySizeGet() function
 description 1-54
 ProArraySizeSet() function
 description 1-54
 ProAsmcomp object
 description 45-2
 ProAsmcompActivatePostAction callback
 65-6
 ProAsmcompActivatePreAction callback 65-6
 ProAsmcompAssemble() function
 description 47-2
 ProAsmcompconstaraintAlloc() function
 description 47-3
 ProAsmcompconstraintArrayFree() function
 description 47-6
 ProAsmcompconstraintAsmreferenceGet()
 function

description 47-5	description 45-8
ProAsmcompconstraintAsmreferenceSet() function	ProAsmcompMdlGet() function
description 47-5	description 45-6
ProAsmcompconstraintAttributesGet() description 47-6	ProAsmcompMdlNameGet() function
ProAsmcompconstraintAttributesSet() description 47-6	description 45-5
ProAsmcompconstraintCompreferenceGet() function	ProAsmcompPackageMove() function
description 47-5	description 47-16
ProAsmcompconstraintCompreferenceSet() function	ProAsmcomppath structure
description 47-6	declaration 45-3
ProAsmcompconstraintFree() description 47-6	ProAsmcomppathDispCsysVisit() function description 2-21
ProAsmcompconstraintOffsetGet() function	ProAsmcomppathDispCurveVisit() function description 2-21
description 47-6	ProAsmcomppathDispPointVisit() function description 2-21
ProAsmcompconstraintOffsetSet() function	ProAsmcomppathDispQuiltVisit() function description 2-21
description 47-6	ProAsmcomppathInit() function
ProAsmcompConstraintsGet() functions	description 45-5
description 47-2	ProAsmcomppathMdlGet() function
ProAsmcompConstraintsSet() functions	description 45-6
description 47-2	ProAsmcomppathTrfGet() function
ProAsmcompconstraintTypeSet() function	assembly components 45-11
description 47-5	description 8-12
ProAsmcompconstraintUserDataGet() function	ProAsmcomppathTrfSet() function 45-11
description 47-6	ProAsmcompPositionSet() description 47-2
ProAsmcompconstraintUserDataSet() function	ProAsmcompRegenerate() function
description 47-6	description 47-3
ProAsmcompConstrRedefUI() function	ProAsmcompRmvUniqueSubasm() function
description 47-16	description 45-8
ProAsmcompConstrRemove() function	ProAsmcompSetPlaced() function
description 47-3	description 45-8
ProAsmcompCreateCopy() function	ProAsmcompTypeGet() function
description 47-2	description 45-6, 47-4
ProAsmcompFillFromMdl function	used with manufacturing components 53-5
description 45-8	ProAsmcompTypeSet() function
ProAsmcompGetPosition() description 47-2	description 45-8
ProAsmcompIsBulkitem() function	ProAsmcompVisibilityGet() function
description 45-7	description 45-8
ProAsmcompIsFrozen() function	ProAssembly object
description 45-7	description 45-2
ProAsmcompIsPackaged() function	ProAssemblyDynPosGet() function 45-11
description 45-7	ProAssemblyDynPosSet() function 45-11
ProAsmcompIsPlaced() function	ProAssemblyExplode() function
description 45-7	description 45-14
ProAsmcompIsSubstitute() function	ProAssemblyIsExploded() function
description 45-7	description 45-14
ProAsmCompIsUnderconstrained() function	ProAssemblyUnexplode() function
description 45-7	description 45-14
ProAsmcompIsUnplaced() function	ProAsynchronousEventLoop() function
description 45-7	description 13-10
ProAsmcompMakeUniqueSubasm() function	ProAsynchronousEventLoopInterrupt() function
	description 13-10
	ProAxis object

- declaration 1-50
- ProAxisDataGet() function
 - description 5-27
- ProAxisIdGet() function
 - description 5-10
- ProAxisInit() function 5-10
- ProAxisSurfaceGet() function
 - description 5-10
- ProBatchAnimationStart() function
 - used in a code example 21-8
- ProBrepRepresentationAlloc function
 - description 26-25
- ProBrepRepresentationFlagsSet function
 - description 26-25
- ProBrepRepresentationFree function
 - description 26-25
- ProBrowserAuthenticate() function
 - description 25-3, 25-5
- ProBsplinedataInit() function
 - description 27-11
- ProCableSectionsDelete() function
 - description 58-19
- ProCablesFromLogicalCreate()function
 - description 58-14
- prodbl_cable_clearance() function
 - description 58-17
- prodbl_designate_connector() function
 - description 58-11
- prodbl_get_assy_connectors() function
 - description 58-9
- prodbl_get_cable_geom() function
 - description 58-17
- prodbl_get_cable_id() function
 - description 58-14
- prodbl_get_cable_logical_ends() function
 - description 58-15
- prodbl_get_cable_subtype() function
 - description 58-14
- prodbl_get_connector_entry_ports() function
 - description 58-9
- prodbl_get_connector_parameters() function
 - description 58-10
- prodbl_get_harness_locations() function
 - description 58-16
- prodbl_get_location_cables() function
 - description 58-16
- prodbl_get_location_pnt() function
 - description 58-16
- prodbl_get_location_type() function
 - description 58-16
- prodbl_get_wire_length() function
 - description 58-17
- prodbl_route_thru_location() function
 - description 58-18
- prodbl_routing_end() function
 - description 58-18
- prodbl_routing_start() function
 - description 58-18
- prodbl_undesignate_connector() function
 - description 58-11
- Process steps
 - access 56-2
 - creating 56-3
 - feature elements 56-4
 - optional elements 56-6
 - types of 56-5
 - visiting 56-2
- ProCheckinUIPreAction callback 65-3
- ProCheckinUIPreAction() function
 - description 25-15
- ProClcmdElemAdd() function
 - description 55-4
- ProClcmdElemCreate() function
 - description 55-4
- ProClcmdElemGet() function
 - description 55-4
- ProClcmdElemGetByIndex() function
 - description 55-4
- ProClcmdElemGetNumber() function
 - description 55-4
- ProClcmdElemRemove() function
 - description 55-4
- ProClcmdElemRemoveByIndex() function
 - description 55-4
- ProClcmdElemSet() function
 - description 55-4
- ProClCommandExpandAction callback 65-7
- ProClCommandGetLocAction callback 65-7
- ProCmdAccessFuncAdd() function
 - description 15-19
- ProCmdAccessFuncRemove() function
 - description 15-19
- ProCmdActionAdd() function
 - description 15-5, 15-20, 15-34
 - used in a code example 15-18
- ProCmdBracketFuncAdd() function
 - description 15-19
- ProCmdCmdIdFind() function
 - description 15-19, 15-34
- ProCmdDesignate() function
 - description 15-21
- ProCmdIconSet() function
 - description 15-21
- ProCmdOptionAdd() function
 - description 15-20, 15-34
- ProCollectionTypeGet() function
 - description 20-7
- ProColorByTypeGet 18-12
- ProColormapGet() function
 - description 18-13
 - used in a code example 18-13
- ProColormapSet() function

- description 18-13, 18-14
- used in a code example 18-14
- ProColorTextModify() function
 - description 18-11
- ProCompoundmenuCreate() function
 - description 15-53
- ProConfigoptArrayGet() function
 - description 12-2
- ProConfigoptSet() function
 - description 12-2
- ProContainingContourFind() function
 - description 5-7
- ProContourdataAlloc() function
 - description 27-8
- ProContourdataEdgeIdArraySet() function
 - description 27-8
- ProContourdataInit() function
 - description 27-8
- ProContourEdgeVisit() function
 - description 5-7
 - used with datum surfaces 5-13
- ProContourTraversalGet() function
 - description 5-7
- ProContourUvpntVerify() function
 - description 5-19
- ProCrvcollectionAlloc() function
 - description 20-10
- ProCrvcollectionCopy() function
 - description 20-10
- ProCrvcollectionInstructionAdd() function
 - description 20-10
- ProCrvcollectionInstructionRemove() function
 - description 20-10
- ProCrvcollectionInstructionsGet() function
 - description 20-8
- ProCrvcollectionRegenerate() function
 - description 20-11
- ProCrvcollinstrAlloc() function
 - description 20-10
- ProCrvcollinstrArrayFree() function
 - description 20-9
- ProCrvcollinstrAttributeIsSet() function
 - description 20-9
- ProCrvcollinstrAttributeUnset() function
 - description 20-10
- ProCrvcollinstrFree() function
 - description 20-9
- ProCrvcollinstrReferenceAdd() function
 - description 20-11
- ProCrvcollinstrReferenceRemove() function
 - description 20-11
- ProCrvcollinstrReferencesGet() function
 - description 20-9
- ProCrvcollinstrTypeGet() function
 - description 20-8
- ProCrvcollinstrValueGet() function
 - description 20-9
- ProCrvcollinstrValueSet() function
 - description 20-11
- ProCsys object
 - declaration 1-50
- ProCsysDataGet() function
 - description 5-27, 8-12
- ProCsysIdGet() function
 - description 5-11
- ProCsysInit() function
 - description 5-11
- ProCurrentWorkspaceExport() function
 - description 25-20
- ProCurrentWorkspaceImpexMessagesGet() function
 - description 25-20
- ProCurrentWorkspaceImport() function
 - description 25-20, H-6
- ProCurve object
 - declaration 1-50
 - description 5-13
- ProCurveCompVisit() function
 - description 5-14
 - for geometry 5-29
- ProCurvedata objects
 - used for external objects 66-7
- ProCurvedataAlloc() function
 - description 27-10, 51-55
- ProCurvedataFree function
 - description 27-9
- ProCurveDataGet() function
 - description 5-29
- ProCurveIdGet() function
 - description 5-13
- ProCurveInit() function
 - description 5-13
- ProCurveLengthEval() function
 - description 5-32
- ProCurveLengthT1T2Eval() function
 - description 5-33
- ProCurveParamByLengthEval() function
 - description 5-33
- ProCurveParamEval() function
 - description 5-20
- ProCurvePersistentColorGet() function
 - description 5-13
- ProCurvePersistentColorSet() function
 - description 5-13
- ProCurvePersistentLinestyleGet() function
 - description 5-13
- ProCurvePersistentLinestyleSet() function
 - description 5-13
- ProCurvesCollect() function
 - description 20-3
- ProCurveTessellationGet function
 - function description 5-15

ProCurveToNURBS() function
 description 5-34
 ProCurveTypeGet() function
 description 5-29
 to distinguish curve type 5-14
 ProCurveXyzdataEval() function
 description 5-19
 ProDatasharingfeatureIsQuerydriven()
 function
 description 50-11
 ProDatasharingfeatureQueryUpdate()
 function
 description 50-11
 ProDatumdataAlloc() function
 description 27-13
 ProDatumdataInit() function
 description 27-13
 prodb_add_parameters() function
 equivalent I-24
 prodb_assemble_by_transform() function
 equivalent I-26
 prodb_assemble_component() function
 equivalent I-26
 prodb_auto_interchange() function
 description 45-16
 prodb_backup_object() function
 equivalent I-20
 prodb_compute_outline() function
 equivalent I-24
 prodb_containing_contour() function
 equivalent I-23
 prodb_contour_traversal() function
 equivalent I-23
 prodb_copy_object() function
 equivalent I-20
 prodb_create_obj() function
 equivalent I-20
 prodb_create_object() function
 equivalent I-20
 prodb_create_parallel_xsec() function
 used in a code example 11-7
 prodb_declare_layout() function 2-20
 prodb_delete_feature() function
 equivalent I-25
 prodb_delete_parameters() function
 equivalent I-24
 prodb_delete_xsec() function
 description 11-7
 used in a code example 11-11
 prodb_designate_param() function
 equivalent I-24
 prodb_edge_data() function
 equivalent I-23
 prodb_edge_direction() function
 equivalent I-23
 prodb_edge_tessellation() function
 equivalent I-23
 prodb_edge_to_nurbs() function
 equivalent I-24
 prodb_edit_file() function
 equivalent I-21
 prodb_entity_to_nurbs() function
 equivalent I-24
 prodb_erase_object() function
 equivalent I-20
 prodb_eval_xyz_dtm_point() function
 equivalent I-25
 prodb_explode_assembly() function
 equivalent I-26
 prodb_find_csys() function
 equivalent I-25
 prodb_find_declared_objects() function
 equivalent I-20
 prodb_find_nobject_depend() function
 equivalent I-20
 prodb_first_cntr_edge() function
 equivalent I-22
 prodb_first_datum() function
 equivalent I-25
 prodb_first_face_contour() function
 equivalent I-22
 prodb_first_member() function
 equivalent I-25
 prodb_first_name_in_list() function
 equivalent I-20
 prodb_first_part_face() function
 equivalent I-22
 prodb_first_xsec() function
 description 11-2
 prodb_get_asm_constraints() function
 equivalent I-26
 prodb_get_asm_transform() function
 equivalent I-26
 prodb_get_comp_role() function
 equivalent I-27
 prodb_get_curve_type() function
 equivalent I-23
 prodb_get_datum_curves() function
 equivalent I-23
 prodb_get_datum_surfaces() function
 equivalent I-23
 prodb_get_edge_curve() function
 equivalent I-23
 prodb_get_edge_type() function
 equivalent I-23
 prodb_get_edge_uv_points() function
 equivalent I-23
 prodb_get_element_name() function
 equivalent I-24
 used with diagrams 58-2
 prodb_get_envelope() function
 equivalent I-24

- prodb_get_face_type() function
equivalent I-23
- prodb_get_feat_parent_child() function
equivalent I-24
- prodb_get_feat_type() function
equivalent I-24
- prodb_get_feature_dtm_points() function
equivalent I-25
- prodb_get_feature_ids() function
equivalent I-24
- prodb_get_feature_surfaces() function
equivalent I-25
- prodb_get_first_axis() function
equivalent I-25
- prodb_get_material_name() function
equivalent I-22
- prodb_get_material_props() function
equivalent I-22
- prodb_get_model_accuracy() function
equivalent I-21
- prodb_get_next_axis() function
equivalent I-25
- prodb_get_object_info() function
equivalent I-20
- prodb_get_object_ptr() function
equivalent I-20
- prodb_get_parameters() function
equivalent I-24
- prodb_get_simplfd_rep_info() function
equivalent I-21
- prodb_get_simplfd_rep_list() function
equivalent I-21
- prodb_get_solid_surfaces() function
equivalent I-22
- prodb_get_solid_volumes() function
equivalent I-23
- prodb_get_surface() function
equivalent I-23
- prodb_get_surface_feature() function
equivalent I-25
- prodb_get_surface_props() function
description 18-23, 18-25
- prodb_get_surface_quilt() function
equivalent I-25
- prodb_interchange_domain() function
description 45-16
- prodb_is_asm_exploded() function
equivalent I-26
- prodb_mass_prop() function
equivalent I-21
- prodb_mass_prop_xsec() function
used in a code example 11-10
- prodb_measure() function
equivalent I-24
- prodb_member_to_object() function
equivalent I-25

- prodb_member_transform() function
equivalent I-26
- prodb_merge_members() function
description 45-15
- prodb_next_cntr_edge() function
equivalent I-23
- prodb_next_datum() function
equivalent I-25
- prodb_next_face_contour() function
equivalent I-22
- prodb_next_member() function
equivalent I-25
- prodb_next_name_in_list() function
equivalent I-20
- prodb_next_part_face() function
equivalent I-22
- prodb_next_xsec() function
description 11-2
- prodb_obj_from_assem_rel() function
equivalent I-26
- prodb_regen_xsec() function
to modify cross sections 11-7
- prodb_rename_object() function
equivalent I-20
- prodb_reset_parameters() function
equivalent I-24
- prodb_resume_feature() function
equivalent I-25
- prodb_retrieve_object() function
equivalent I-20
- prodb_retrieve_simplfd_rep() function
equivalent I-21
- prodb_rls_edge_curve() function
equivalent I-23
- prodb_rls_surface() function
equivalent I-24
- prodb_save_object() function
equivalent I-20
- prodb_set_element_name() function
equivalent I-24
used with diagrams 58-2
- prodb_set_material_name() function
equivalent I-22
- prodb_set_member_transform() function
equivalent I-26
- prodb_set_model_accuracy() function
equivalent I-21
- prodb_set_parameters() function
equivalent I-24
- prodb_set_surface_props() function
description 18-23
- prodb_suppress_feature() function
equivalent I-25
- prodb_surface_tessellation() function
surface properties 18-25
- prodb_surface_to_nurbs() function

- equivalent I-24
- prodb_undeclare_layout() function 2-20
- prodb_unpack_csyes() function
 - equivalent I-25
- prodb_unset_surface_props() function
 - description 18-23
- prodb_uv_in_face_domain() function
 - equivalent I-23
- prodb_vertex_data() function
 - equivalent I-23
- prodb_was_object_modified() function
 - equivalent I-20
- prodb_xsec_name() function
 - description 11-2
- prodev_get_proe_status() function
 - equivalent I-16
- prodev_handle_interrupt() function
 - equivalent I-16
- prodev_main() function
 - equivalent I-16
- prodev_notify() function
 - equivalent I-26
- prodev_set_interrupt_func() function
 - equivalent I-16
- prodev_set_proe_term_func() function
 - equivalent I-16
- prodev_start_proengineer() function
 - equivalent I-16
- prodgm_get_ids() function
 - description 58-2
- prodgm_get_node_connections() function
 - used in a code example 58-6
- prodgm_get_nodes() function
 - description 58-3
- prodgm_get_parameters() function
 - description 58-2
- prodgm_get_wire_nodes() function
 - description 58-5
- prodgm_get_wire_spool() function
 - description 58-5
- Prodgm_item structure 58-3
- Prodgm_node_ref structure 58-6
- prodgm_set_parameters() function
 - description 58-4
- prodgm_set_wire_spool() function
 - description 58-5
- prodim_drawing_dim_view_id() function
 - 51-22
- prodim_set_dim() function
 - used with cross-sections 11-7
- ProDimensionAnnotationelemCreate()
 - function
 - description 22-45
- ProDimensionAnnotationelemDelete()
 - function
 - description 22-45

- ProDimensionAttachmentSet() function
 - description 22-47
- ProDimensionAttachmentsGet() function
 - description 22-47
- ProDimensionBasicSet() function
 - description 22-29
- ProDimensionBoundGet() function
 - description 22-25
- ProDimensionBoundSet() function
 - description 22-29
- ProDimensionCanRegen() function
 - description 22-47
- ProDimensionCreate() function
 - description 22-46
- ProDimensionDecimalsGet() function
 - description 22-24
- ProDimensionDecimalsSet() function
 - description 22-28
- ProDimensionDelete() function
 - description 22-47
- ProDimensionDenominatorGet() function
 - description 22-24
- ProDimensionDenominatorSet() function
 - description 22-28
- ProDimensionDimensionReset() function
 - description 22-29
- ProDimensionDisplayedToleranceGet()
 - function
 - description 22-32
- ProDimensionDisplayRoundedValueSet()
 - function
 - description 22-24
- ProDimensionElbowlengthSet() function
 - description 22-29
- ProDimensionFeatureGet() function
 - description 22-25
- ProDimensionInspectionSet() function
 - description 22-29
- ProDimensionIsAccessibleInModel() function
 - description 22-25
- ProDimensionIsDisplayRoundedValue()
 - function
 - description 22-23
- ProDimensionIsDriving() function
 - description 22-46
- ProDimensionIsFractional() function
 - description 22-24
- ProDimensionIsRegenednegative() function
 - description 22-24
- ProDimensionIsReldriven() function
 - description 22-24
- ProDimensionIsToleranceDisplayed() function
 - description 22-33
- ProDimensionLocationGet() function
 - description 22-36
- ProDimensionNomvalueGet() function

- description 22-23
- ProDimensionPlaneGet() function
 - description 22-44
- ProDimensionPlaneSet() function
 - description 22-44
- ProDimensionSymbolGet() function
 - description 22-23
- ProDimensionSymbolSet() function
 - description 22-28
- ProDimensionTextGet() function
 - description 22-35, 22-38
- ProDimensionTextSet() function
 - description 22-35
- ProDimensionTextstyleGet() function
 - description 22-36
- ProDimensionTextstyleSet() function
 - description 22-36
- ProDimensionToleranceGet() function
 - description 22-34
- ProDimensionToleranceSet() function
 - description 22-34
- ProDimensionTollabelGet() function
 - description 22-35
- ProDimensionTollabelSet() function
 - description 22-35
- ProDimensionToltypeGet() function
 - description 22-33
- ProDimensionToltypeSet() function
 - description 22-33
- ProDimensionTypeGet() function
 - description 22-23
- ProDimensionValueGet() function
 - description 22-23
- ProDimensionValueSet() function
 - description 22-28
- ProDimlocationArrowsGet() function
 - description 22-37, 22-41
- ProDimlocationArrowtypesGet() function
 - description 22-37
- ProDimlocationFree() function
 - description 22-36
- ProDimlocationNormalGet() function
 - description 22-38
- ProDimlocationTextGet() function
 - description 22-37
- ProDimlocationWitnesslinesGet() function
 - description 22-37
- ProDimlocationZExtensionlinesGet() function
 - description 22-37
- ProDimModifyValuePreAction callback 65-5
- ProDirectoryChange() function
 - description 12-3
- ProDirectoryChangePostAction callback 65-6
- ProDirectoryChoose() function
 - description 12-6
- ProDirectoryCurrentGet() function

- description 12-3
- ProDispdatAlloc() function
 - description 66-7
 - used in a code example 66-21
- ProDispdatColorGet() function
 - description 66-9
- ProDispdatColorSet() function
 - description 66-9
 - used in a code example 66-21
- ProDispdatEntsGet() function
 - description 66-7
- ProDispdatEntsSet() function
 - description 66-7
 - used in a code example 66-21
- ProDispdatLinestyleGet() function
 - description 66-9
- ProDispdatLinestyleSet() function
 - description 66-9
 - used in a code example 66-21
- ProDispdatPropsGet() function
 - description 66-8
- ProDispdatPropsSet() function
 - description 66-8
- ProDispdatScaleGet() function
 - description 66-10
- ProDispdatScaleSet() function
 - description 66-10
 - used in a code example 66-21
- ProDispdatTrfGet() function
 - description 66-8
- ProDispdatTrfSet() function
 - description 66-8
 - used in a code example 66-21
- ProDisplaymodeGet() function
 - description 22-21
- ProDisplaymodeSet() function
 - description 22-21
- ProDisplist2dCreate() function 18-19
- ProDisplist2dDelete() function 18-19
- ProDisplist2dDisplay() function 18-19
- ProDisplist3dCreate() function 18-19
- ProDisplist3dDelete() function 18-19
- ProDisplist3dDisplay() function 18-19
- ProDisplistInvalidate() function
 - description 12-2
- ProDltsymgroupLevelIsExclusive() function
 - description 51-89
- ProDrawingAsmsimprepAdd() function
 - description 51-48
- ProDrawingAsmsimprepDelete() function
 - description 51-48
- ProDrawingContextGet()
 - description 51-6
- ProDrawingContextSet()
 - description 51-6
- ProDrawingCurrentSheetGet() function

description 51-12
 ProDrawingCurrentSheetSet() function
 description 51-12
 ProDrawingDimCreate() function
 description 19-4
 ProDrawingDimensionIsDisplayed() function
 description 51-115
 ProDrawingDimensionVisit() function
 description 22-23
 ProDrawingDtlentitiesCollect() function
 description 51-51
 ProDrawingDtlentityVisit() function
 description 51-51
 ProDrawingDtlgroupsCollect() function
 description 51-52
 ProDrawingDtlgroupVisit() function
 description 51-52
 ProDrawingDtlnotesCollect() function
 description 51-51
 ProDrawingDtlnoteVisit() function
 description 51-51
 ProDrawingDtlsymdefRetrieve() function
 description 51-80
 ProDrawingDtlsymdefsCollect() function
 description 51-51
 ProDrawingDtlsymdefVisit() function
 description 51-51
 ProDrawingDtlsyminstsCollect() function
 description 51-52
 ProDrawingDtlsyminstVisit() function
 description 51-52
 ProDrawingedgedisplayColorGet() function
 description 51-91
 ProDrawingedgedisplayColorSet() function
 description 51-91
 ProDrawingedgedisplayFontGet() function
 description 51-91
 ProDrawingedgedisplayFontSet() function
 description 51-91
 ProDrawingedgedisplayFree() function
 description 51-91
 ProDrawingEdgeDisplayGet() function
 description 51-90
 ProDrawingedgedisplayGlobalSet() function
 description 51-91
 ProDrawingedgedisplayIsGlobal() function
 description 51-91
 ProDrawingEdgeDisplaySet() function
 description 51-91
 ProDrawingedgedisplayWidthGet() function
 description 51-91
 ProDrawingedgedisplayWidthSet() function
 description 51-91
 ProDrawingFormatAdd() function
 description 51-13
 ProDrawingFormatGet() function

description 51-13
 ProDrawingMerge() function
 description 51-11
 ProDrawingOLEobjectApplicationtypeGet()
 function
 description 51-60
 ProDrawingOLEobjectOutlineGet() function
 description 51-60
 ProDrawingOLEobjectPathGet() function
 description 51-60
 ProDrawingOLEobjectSheetGet() function
 description 51-60
 ProDrawingOLEobjectsVisit() function
 description 51-52
 ProDrawingPosToLocgrid() function
 description 51-7
 ProDrawingScaleGet() function
 description 51-19
 ProDrawingSetupOptionGet() function
 description 51-6
 ProDrawingSetupOptionSet() function
 description 51-6
 ProDrawingSheetCopy() function
 description 51-13
 ProDrawingSheetCreate() function
 description 51-12
 ProDrawingSheetDelete() function
 description 51-12
 ProDrawingSheetFormatIsBlanked() function
 description 51-13
 ProDrawingSheetOrientationGet() function
 description 51-12
 ProDrawingSheetscount() function
 description 51-12
 ProDrawingSheetSizeGet() function
 description 51-12
 ProDrawingSheetsReorder() function
 description 51-13
 ProDrawingSheetTrfGet()
 used in a code example 8-12
 ProDrawingSheetTrfGet() function
 description 8-9, 51-12
 ProDrawingSheetUnitsGet() function
 description 51-12
 ProDrawingSimpredsCollect() function
 description 51-48
 ProDrawingSolidAdd() function
 description 51-47
 ProDrawingSolidDelete() function
 description 51-48
 ProDrawingSolidReplace() function
 description 51-48
 ProDrawingSolidsCollect() function
 description 51-47
 ProDrawingSolidsVisit() function
 description 51-47

ProDrawingSystemDtlsymdefRetrieve()	description 51-23
function	ProDrawingViewDetailBoundaryGet()
description 51-80	function
ProDrawingTableCreate() function	description 51-29
description 51-93	ProDrawingViewDetailBoundarySet()
ProDrawingTablesCollect() function	function
description 51-93	description 51-29
ProDrawingTablesUpdate() function	ProDrawingViewDetailCreate() function
description 51-103	description 51-28
ProDrawingTableVisit() function	ProDrawingViewDetailCurvedataGet()
description 51-93	function
ProDrawingtreeCollapse() function	description 51-29
description 51-8	ProDrawingViewDetailCurvedataSet()
ProDrawingtreeExpand() function	function
dewcription 51-8	description 51-29
ProDrawingtreeRefresh() function	ProDrawingViewDetailReferenceGet()
description 51-8	function
ProDrawingView2DSectionGet() function	description 51-28
description 51-35	ProDrawingViewDetailReferenceSet()
ProDrawingView2DSectionNumberGet()	function
function	description 51-28
description 51-36	ProDrawingViewDisplayGet() function
ProDrawingView2DSectionSet() function	described 51-19
description 51-35	ProDrawingViewDisplaySet() function
ProDrawingView3DSectionGet() function	description 51-24
description 51-36	ProDrawingViewErase() function
ProDrawingView3DSectionSet() function	description 51-23
description 51-36	ProDrawingViewExplodedGet() function
ProDrawingViewAlignmentGet() function	description 51-37
description 51-18	ProDrawingViewExplodedSet() function
ProDrawingViewAlignmentSet() function	description 51-37
description 51-23	ProDrawingViewFlagGet() function
ProDrawingViewAuxiliaryCreate() function	description 51-19
description 51-29	ProDrawingViewFlagSet() function
ProDrawingViewAuxiliaryInfoGet() function	description 51-23
description 51-30	ProDrawingViewFullVisibleAreaSet()
ProDrawingViewAuxiliarySet() function	function
description 51-30	description 51-32
ProDrawingViewBrokenNumberGet()	ProDrawingViewHalfVisibleAreaGet()
function	function
description 51-34	description 51-32
ProDrawingViewBrokenVisibleAreaGet()	ProDrawingViewHalfVisibleAreaSet()
function	function
description 51-33	description 51-32
ProDrawingViewBrokenVisibleAreaSet()	ProDrawingViewIsErased() function
function	description 51-20
description 51-33	ProDrawingViewIsOverlay() function
ProDrawingViewChildrenGet() function	description 51-28
description 51-18	ProDrawingViewMove() function
ProDrawingViewCollect() function	description 51-22
description 51-18	ProDrawingViewNameGet function
ProDrawingViewColorSourceGet() function	description 51-20
description 51-19	ProDrawingViewNameSet() function
ProDrawingViewDatumdisplayGet() function	description 51-24
description 51-20	ProDrawingViewNeedsRegen() function
ProDrawingViewDelete() function	description 51-20

ProDrawingViewOrientationFromAngleSet()	function
description	51-31
ProDrawingViewOrientationFromNameSet()	function
description	51-31
ProDrawingViewOrientationFromReferenceSet()	function
description	51-31
ProDrawingViewOriginGet()	function
description	51-18
ProDrawingViewOriginSet()	function
description	51-23
ProDrawingViewOutlineGet()	function
described	51-18
ProDrawingViewParentGet()	function
description	51-18
ProDrawingViewPartialVisibleAreaGet()	function
description	51-33
ProDrawingViewPartialVisibleAreaSet()	function
description	51-33
ProDrawingViewPerspectiveScaleGet()	function
description	51-19
ProDrawingViewPipingdisplayGet()	function
description	51-20
ProDrawingViewPipingdisplaySet()	function
description	51-24
ProDrawingViewProjectionSet()	function
description	51-23
ProDrawingViewResume()	function
description	51-23
ProDrawingViewRevolveCreate()	function
description	51-30
ProDrawingViewRevolveInfoGet()	function
description	51-30
ProDrawingViewScaleGet()	function
described	51-19
ProDrawingViewScaleIsUserdefined	function
described	51-18
ProDrawingViewScaleSet()	function
description	51-23
ProDrawingViewSectionTypeGet()	function
description	51-35
ProDrawingViewSheetGet()	function
description	51-18
ProDrawingViewSimplifiedGet()	function
description	51-37
ProDrawingViewSimplifiedSet()	function
description	51-37
ProDrawingViewSinglepartSectionGet()	function
description	51-36
ProDrawingViewSinglepartSectionSet()	function
description	51-36
ProDrawingViewSolidGet()	function
description	51-19
ProDrawingViewTransformGet()	function
description	51-19
ProDrawingViewTypeGet()	function
description	51-18
ProDrawingViewVisibleareaTypeGet()	function
description	51-32
ProDrawingViewVisit()	function
description	51-18
ProDrawingViewZclippingGet	function
description	51-20
ProDrawingViewZclippingSet	function
description	51-24
prodrw_get_overlay_view_id()	function
51-27	
Prodtl_attach	structure
51-91	
prodtl_create()	function
detail items	51-49
prodtl_delete()	function
51-49	
prodtl_display()	function
51-49	
prodtl_get_ids()	function
51-49	
prodtl_get_item()	function
51-49	
Prodtl_leader	structure
51-91	
prodtl_modify()	function
51-49	
ProDtlattachAlloc()	function
description	51-64
ProDtlattachArrowtypeGet()	function
description	51-66
ProDtlattachFree()	function
description	51-66
ProDtlattachGet()	function
description	51-65
ProDtlentity	
object handle	51-50
ProDtlentityCreate()	function
description	51-53
ProDtlentitydataAlloc()	function
description	51-57
ProDtlentitydataColorGet()	function
description	51-57
ProDtlentitydataColorSet()	function
description	51-57
ProDtlentitydataConstructionSet()	function
description	51-58
ProDtlentitydataCurveGet()	function
description	51-57
ProDtlentitydataCurveSet()	function
defined	51-57
ProDtlentitydataFontGet()	function
description	51-57
ProDtlentitydataFontSet()	function
description	51-57
ProDtlentitydataFree()	function

- defined 51-57
- ProDtlentityDataGet() function
 - description 51-53
- ProDtlentitydataIsConstruction() function
 - description 51-58
- ProDtlentitydataViewGet() function
 - description 51-58
- ProDtlentitydataViewSet() function
 - description 51-58
- ProDtlentitydataWidthGet() function
 - description 51-58
- ProDtlentitydataWidthSet() function
 - description 51-58
- ProDtlentityDelete() function
 - description 51-53
- ProDtlentityDraw() function
 - description 51-52
- ProDtlentityErase() function
 - description 51-52
- ProDtlentityIsOLEObject() function
 - description 51-55
- ProDtlentityModify() function
 - description 51-53
- ProDtlgroup
 - object handle 51-51
- ProDtlgroupCreate() function
 - description 51-54
- ProDtlgroupdataDisplayedSet() function
 - description 51-85
- ProDtlgroupDataGet() function
 - description 51-54
- ProDtlgroupdataIsDisplayed() function
 - description 51-85
- ProDtlgroupDelete() function
 - description 51-54
- ProDtlgroupModify() function
 - description 51-54
- ProDtlitem
 - object 51-50
- ProDtlnote
 - object handle 51-50
- ProDtlnoteCreate() function
 - description 51-53
- ProDtlnotedataAlloc() function
 - description 51-67
- ProDtlnotedataAngleGet() function
 - description 51-68
- ProDtlnotedataAngleSet() function
 - description 51-68
- ProDtlnotedataAttachmentGet() function
 - description 51-68
- ProDtlnotedataAttachmentSet() function
 - description 51-68
- ProDtlnotedataColorGet() function
 - description 51-68
- ProDtlnotedataColorSet() function
 - description 51-68
- ProDtlnotedataDisplayedSet() function
 - description 51-68
- ProDtlnotedataElbowlengthGet() function
 - description 51-68
- ProDtlnotedataElbowlengthSet() function
 - description 51-68
- ProDtlnotedataFree() function
 - description 51-67
- ProDtlnoteDataGet() function
 - description 51-53, 51-54
- ProDtlnotedataIdGet() function
 - description 51-67
- ProDtlnotedataIsDisplayed() function
 - description 51-68
- ProDtlnotedataIsMirrored() function
 - description 51-67
- ProDtlnotedataJustifGet() function
 - description 51-68
- ProDtlnotedataJustifSet() function
 - description 51-68
- ProDtlnotedataLeaderAdd() function
 - description 51-68
- ProDtlnotedataLeadersCollect() function
 - description 51-68
- ProDtlnotedataLeadersSet() function
 - description 51-68
- ProDtlnotedataLineAdd() function
 - description 51-67
- ProDtlnotedataLinesCollect() function
 - description 51-67
- ProDtlnotedataMirrorSet() function
 - description 51-67
- ProDtlnoteDelete() function
 - description 51-53
- ProDtlnoteDraw() function
 - description 51-52
- ProDtlnoteDtlsyminstsCollect() function
 - description 51-68
- ProDtlnoteErase() function
 - description 51-52
- ProDtlnotedataLinesSet() function
 - description 51-67
- ProDtlnotelineAlloc() function
 - described 51-62
- ProDtlnoteLineEnvelopeGet() function
 - description 51-53
- ProDtlnotelineFree() Function
 - described 51-62
- ProDtlnotelineFree() function
 - described 51-62
- ProDtlnotelineTextAdd() function
 - described 51-62
- ProDtlnotelineTextsCollect() function
 - description 51-62
- ProDtlnotelineTextsSet() function

- description 51-62
- ProDtlnoteModify() function
 - description 51-53
- ProDtlnoteRemove() function
 - description 51-52
- ProDtlnoteShow() function
 - description 51-52
- ProDtlnotetextAlloc() function
 - description 51-61
- ProDtlnotetextFontGet() function
 - description 51-62
- ProDtlnotetextFontSet() function
 - description 51-62
- ProDtlnotetextFree() function
 - description 51-61
- ProDtlnotetextHeightGet() function
 - description 51-61
- ProDtlnotetextHeightSet() function
 - description 51-61
- ProDtlnotetextSlantGet() function
 - description 51-61
- ProDtlnotetextSlantSet() function
 - description 51-61
- ProDtlnotetextStringGet() function
 - description 51-62
- ProDtlnotetextStringSet() function
 - description 51-62
- ProDtlnotetextThicknessGet() function
 - description 51-62
- ProDtlnotetextThicknessSet() function
 - description 51-62
- ProDtlnotetextUlineGet() function
 - description 51-62
- ProDtlnotetextUlineSet() function
 - description 51-62
- ProDtlnotetextWidthGet() function
 - description 51-61
- ProDtlnotetextWidthSet() function
 - description 51-61
- ProDtlsymdef
 - object handle 51-51
- ProDtlsymdefCreate() function
 - description 51-53
- ProDtlsymdefDataGet() function
 - description 51-53
- ProDtlsymdefDelete() function
 - description 51-53
- ProDtlsymdefModify() function
 - description 51-53
- ProDtlsymdefToModelCopy() function
 - description 51-53
- ProDtlsymgroupdataAlloc() function
 - description 51-89
- ProDtlsymgroupdataFree() function
 - description 51-89
- ProDtlsymgroupDataGet() function

- description 51-88
- ProDtlsymgroupdataItemAdd() function
 - description 51-89
- ProDtlsymgroupdataItemsCollect() function
 - description 51-88
- ProDtlsymgroupdataItemsSet() function
 - description 51-89
- ProDtlsymgroupdataNameGet() function
 - description 51-88
- ProDtlsymgroupdataNameSet() function
 - description 51-89
- ProDtlsymgroupDelete() function
 - description 51-90
- ProDtlsymgroupLevelExclusiveSet() function
 - description 51-90
- ProDtlsymgroupModify() function
 - description 51-89
- ProDtlsymgroupParentGet() function
 - description 51-88
- ProDtlsymgroupSubgroupCreate() function
 - description 51-89
- ProDtlsymgroupSubgroupsCollect() function
 - description 51-88
- ProDtlsyminst
 - object handle 51-51
- ProDtlsyminstCreate() function
 - description 22-61, 51-53
- ProDtlsyminstdataDisplayedSet() function
 - description 51-83
- ProDtlsyminstDataGet() function
 - defined 51-54
- ProDtlsyminstDataGet() function
 - description 22-61, 51-53
- ProDtlsyminstdataGroupoptionsSet() function
 - description 51-83
- ProDtlsyminstdataIsDisplayed() function
 - description 51-83
- ProDtlsyminstdataPlaneGet() function
 - description 22-61
- ProDtlsyminstdataPlaneSet() function
 - description 22-61
- ProDtlsyminstdataScaledheightGet() function
 - description 51-82
- ProDtlsyminstdataScaledheightSet() function
 - description 51-82
- ProDtlsyminstDelete() function
 - description 51-54
- ProDtlsyminstDraw() function
 - description 51-52
- ProDtlsyminstErase() function
 - description 51-52
- ProDtlsyminstModify() function
 - description 22-61, 51-54
- ProDtlsyminstRemove() function
 - description 51-52
- ProDtlsyminstShow() function

description 51-52	description 51-96
ProDtlSymInstSymGroupsCollect() function	ProDwgTableErase() function
description 51-88	description 51-96
ProDtmcsysTransformfileRead() function	ProDwgTableInfoGet() function
description 31-76	description 51-93
ProDwgLayerDisplaystatusGet() function	ProDwgTableIsFromFormat() function
description 2-16	description 51-94
ProDwgLayerDisplaystatusSet() function	ProDwgTableRetrieve() function
description 2-16	description 51-94
ProDwgLayerItemInit() function	ProDwgTableRotate() function
description 2-15	description 51-96
ProDwgSheetRegenerate() function	ProDwgTableRowAdd() function
description 51-13	description 51-95
ProDwgTableCellComponentGet() function	ProDwgTableRowDelete() function
description 51-103	description 51-95
ProDwgTableCellsComment() function	ProDwgTableRowDeletePostAction callback
description 51-103	65-6
ProDwgTableCellNoteGet() function	ProDwgTableRowDeletePostAction()function
description 51-94	description 51-97
ProDwgTableCellRefmodelGet() function	ProDwgTableRowDeletePreAction callback
description 51-103	65-6
ProDwgTableCellsMerge() function	ProDwgTableRowDeletePreAction()function
description 51-96	description 51-97
ProDwgTableCellsRemesh() function	ProDwgTableRowHeightSet() function
description 51-96	description 51-95
ProDwgTableCelltextGet() function	ProDwgTableRowsCount() function
description 51-94	description 51-94
ProDwgTableCelltextWrap() function	ProDwgTableRowSizeGet() function
description 51-96	description 51-94
ProDwgTableColumnAdd() function	ProDwgTableSave() function
description 51-95	description 51-96
ProDwgTableColumnDelete() function	ProDwgTableSegCount() function
description 51-95	description 51-103
ProDwgTableColumnsCount() function	ProDwgTableSegmentOriginSet() function
description 51-94	description 51-103
ProDwgTableColumnSizeGet() function	ProDwgTableSegMove() function
description 51-94	description 51-103
ProDwgTableColumnWidthSet() function	ProDwgTableSegSheetGet() function
description 51-95	description 51-103
ProDwgTabledataAlloc() function	ProDwgTableTextEnter() function
description 51-92	description 51-93
ProDwgTabledataColumnsSet() function	ProDwgViewRegenerate() function
description 51-92	description 51-24
ProDwgTabledataOriginSet() function	ProEdge object
description 51-92	declaration 1-50
ProDwgTabledataRowsSet() function	ProEdgedataAlloc() function
description 51-92	description 27-9
ProDwgTabledataSizetypeSet() function	ProEdgeDataGet() function
description 51-92	description 5-23
ProDwgTableDelete() function	ProEdgedataInit() function
description 51-96	description 27-9
ProDwgTableDeletePostAction callback 65-6	ProEdgeDirGet() function
ProDwgTableDeletePostAction() function	description 5-7
description 51-96	ProEdgeLengthEval() function
ProDwgTableDeletePreAction callback 65-6	description 5-32
ProDwgTableDeletePreAction() function	ProEdgeLengthT1T2Eval() function

- description 5-33
- ProEdgeNeighborsGet function
 - description 5-15
- ProEdgeNeighborsGet() function
 - description 5-7
- ProEdgeParamByLengthEval() function
 - description 5-33
- ProEdgeParamEval() function
 - description 5-20
- ProEdgeTessellationGet() function
 - description 5-15
- ProEdgeToNURBS() function
 - description 5-34
- ProEdgeTypeGet() function
 - description 5-22
- ProEdgeUvdataEval() function
 - description 5-19
- ProEdgeVertexdataGet() function
 - description 5-7
- ProEdgeXyzdataEval() function
 - description 5-19
- ProElemdiagnosticFree() function
 - description 29-18
- ProElemdiagnosticMessageGet() function
 - description 29-18
- ProElemdiagnosticSeverityGet() function
 - description 29-18
- ProElement object
 - description 29-3
- ProElementAlloc() function
 - creating tools 53-10
 - description 29-13
 - to create NC sequences 53-38
- ProElementArrayCount() function
 - description 29-14
- ProElementArrayGet() function
 - description 29-14
- ProElementArraySet() function
 - description 29-14
- ProElementBooleanGet()function
 - description 29-17
- ProElementBooleanSet()function
 - description 29-17
- ProElementChildrenGet() function
 - description 29-14
- ProElementChildrenSet() function
 - description 29-14
- ProElementCollectionGet() function
 - description 20-15
- ProElementCollectionProcess() function
 - description 20-15
- ProElementCollectionSet() function
 - description 20-15
- ProElementDecimalsGet() function
 - description 29-16
- ProElementDecimalsSet() function

- description 29-16
- ProElementDiagnosticsCollect() function
 - description 29-18
- ProElementDoubleGet()function
 - description 29-16
- ProElementDoubleSet()function
 - description 29-16
- ProElementFree() function
 - description 29-13
- ProElementIdGet() function
 - description 29-13
- ProElementIdSet() function
 - description 29-13
- ProElementIntegerGet()function
 - description 29-16
- ProElementIntegerSet()function
 - description 29-16
- ProElementIsArray() function
 - description 29-14
- ProElementIsCompound() function
 - description 29-14
- ProElementIsMultival() function
 - description 29-14
- ProElementReferenceGet()function
 - description 29-16
- ProElementReferenceSet()function
 - description 29-16
- ProElementReferencesGet()function
 - description 29-16
- ProElementReferencesSet()function
 - description 29-16
- ProElementSpecialvalueGet()function
 - description 29-17
- ProElementSpecialvalueSet()function
 - description 29-17
- ProElementStringGet()function
 - description 29-17
- ProElementStringSet()function
 - description 29-17
- ProElementTransformGet()function
 - description 29-17
- ProElementTransformSet()function
 - description 29-17
- ProElementValueGet 29-15
- ProElementValueGet() function
 - description 29-9
- ProElementValueSet 29-15
- ProElementValueSet() function
 - description 29-9
- ProElementValuesGet() function
 - description 29-9
- ProElementValuesSet() function
 - description 29-9
- ProElementValuetypeGet()function
 - description 29-15
- ProElementWstringGet()function

- description 29-17
- ProElementWstringSet() function
 - description 29-17
- ProElementWstrOpts() function
 - description 29-17
- ProElementwstropsAlloc() function
 - description 29-16
- ProElementwstropsExpressionsSet() function
 - description 29-16
- ProElementwstropsFree() function
 - description 29-17
- ProElementwstropsSignoptionsSet() function
 - description 29-16
- ProElemId object
 - description 29-4
- ProElempath object
 - description 29-11
- ProElempathAlloc() function
 - description 29-12
- ProElempathCompare() function
 - description 29-13
- ProElempathCopy() function
 - description 29-13
- ProElempathDataGet() function
 - description 29-12
- ProElempathDataSet() function
 - description 29-12
- ProElempathFree() function
 - description 29-12
- ProElempathItem structure
 - declaration 29-12
- ProElempathSizeGet() function
 - description 29-13
- ProElemtreeElementAdd() function
 - adding manufacturing elements 53-17
 - description 29-14
- ProElemtreeElementGet() function
 - description 29-14
- ProElemtreeElementRemove() function
 - description 29-14
- ProElemtreeElementVisit() function
 - description 29-14
- ProElemtreeFromXMLCreate() function
 - description 29-29
- ProElemtreeVerify() function
 - description 29-18
- ProElemtreeWrite() function
 - description 29-29
- ProEllipsedataInit() function
 - description 27-10
- ProEngineerConnect() function
 - description 13-5
- ProEngineerConnectIdExtract() function
 - description 13-6
- ProEngineerConnectIdGet() function
 - description 13-5
- ProEngineerConnectionStart() function
 - description 13-4
- ProEngineerConnectWS() function
 - description 13-5
- ProEngineerDisconnect() function
 - description 13-5
- ProEngineerDisplaydatecodeGet() function
 - description 1-43
- ProEngineerEnd() function
 - batch session 1-46
 - description 13-4
- ProEngineerMultithreadModeEnable()
 - function
 - description 1-45
- ProEngineerStart() function
 - description 13-4
- ProEngineerStatusGet() function
 - description 13-9
- ProErritemType enum
 - declaration 29-20
- ProError return type
 - description 1-19
- ProErrorlist declaration 29-20
- ProEventProcess() function
 - description 13-9
- ProExpldstate structure
 - declaration 45-14
- ProExpldstateActivate() function
 - description 45-15
- ProExpldstateActiveGet() function
 - description 45-15
- ProExpldstateSelect() function
 - description 45-15
- ProExportVRML function
 - description 26-28
- ProExtdataClass structure
 - description 10-3
- ProExtdataClassNamesList() function
 - description 10-6
- ProExtdataClassRegister() function
 - description 10-4
- ProExtdataClassUnregister() function
 - description 10-4
- ProExtdataFree() function
 - description 10-6
- ProExtdataInit() function
 - description 10-4
- ProExtdataLoadAll() function
 - description 10-5
- ProExtdataSlot structure
 - description 10-3
- ProExtdataSlotCreate() function
 - description 10-4
- ProExtdataSlotDelete() function
 - description 10-5
- ProExtdataSlotIdsList() function

description 10-6	description 66-5
ProExtdataSlotNamesList() function	ProExtobjExttypeGet() function
description 10-6	description 66-5
ProExtdataSlotRead() function	ProExtobjExttypeSet() function
description 10-6	description 66-5
ProExtdataSlotWrite() function	ProExtobjOwnerobjGet() function
description 10-4	description 66-4
ProExtdataTerm() function	ProExtobjOwnerobjSet() function
description 10-5	description 66-4
ProExtobj object 66-2	ProExtobjRefAdd() function
ProExtobjCallbacks structure	description 66-13
declaration 66-15	used in a code example 66-23
ProExtobjCallbacksSet() function	ProExtobjRefAlloc() function
description 66-14	description 66-13
ProExtobjCBAct() function	used in a code example 66-23
description 66-14	ProExtobjRefRemove() function
ProExtobjCBEnable() function	description 66-13
description 66-14	ProExtobjRefselectionGet() function
used in a code example 66-22	description 66-13
ProExtobjClass object	ProExtobjRefselectionSet() function
description 66-3	used in a code example 66-23
ProExtobjClassCreate() function	ProExtobjReftypeGet() function
description 66-3	description 66-13
used in a code example 66-20	ProExtobjReftypeSet() function
ProExtobjClassDelete() function	description 66-13
description 66-3	ProExtobjRefVisit() function
ProExtobjClassGet() function	description 66-13
description 66-3	ProExtobjReusableClear() function
ProExtobjClassWarningDisable() function	description 66-4
description 66-16	ProExtobjReusableGet() function
ProExtobjClassWarningEnable() function	description 66-4
description 66-16	ProExtobjReusableSet() function
used in a code example 66-20	description 66-4
ProExtobjCreate() function	used in a code example 66-21
description 66-3	ProExtobjScreentrfGet() function
used in a code example 66-21	description 66-8
ProExtobjdataAdd() function	ProExtobjVisit() function
description 66-11	description 66-6
used in a code example 66-21	ProExtobjWarningDisable() function
ProExtobjdataFree() function	description 66-16
description 66-11	ProExtobjWarningEnable() function
ProExtobjdataGet() function	description 66-16
description 66-11	ProFaminstance object
ProExtobjdataRemove() function	description 9-2
description 66-11	ProFaminstanceAdd() function
ProExtobjdataSet() function	description 9-5
description 66-11	ProFaminstanceCheck() function
ProExtobjdataType enum	description 9-6
declaration 66-12	ProFaminstanceCreate() function
ProExtobjDelete() function	description 9-5
description 66-3	ProFaminstanceErase() function
ProExtobjDispprops object	description 9-6
description 66-9	ProFaminstanceFamtableItemIsDefault()
ProExtobjExtidGet() function	function
description 66-5	description 9-5
ProExtobjExtidSet() function	ProFaminstanceGenericGet() function

- description 9-6
- ProFamInstanceImmediateGenericInfoGet() function
 - description 9-6
- ProFamInstanceInit() function
 - description 9-5
- ProFamInstanceIsExtLocked() function
 - description 9-6
- ProFamInstanceIsFlatState() function
 - description 9-6, 52-4
- ProFamInstanceIsModifiable() function
 - description 9-6
- ProFamInstanceIsVerified() function
 - description 9-6
- ProFamInstanceLock() function
 - description 9-6
- ProFamInstanceMdlGet() function
 - description 9-5
- ProFamInstanceRemove() function
 - description 9-5
- ProFamInstanceRetrieve() function
 - description 9-6
- ProFamInstanceSelect() function
 - description 9-5
- ProFamInstanceValueGet() function
 - description 9-5
 - used in a code example 9-9
- ProFamInstanceValueSet() function
 - description 9-5
- ProFamtable object
 - description 9-2
- ProFamtableCheck() function
 - description 9-2
 - used in a code example 9-8
- ProFamtableEdit() function
 - description 9-2
- ProFamtableErase() function
 - description 9-2
- ProFamtableInit() function
 - description 9-2
 - used in a code example 9-10
- ProFamtableInstanceVisit() function
 - description 9-3
 - used in a code example 9-10
- ProFamtableIsModifiable() function
 - description 9-2
- ProFamtableItem object
 - description 9-2
- ProFamtableItemAdd() function
 - description 9-7
- ProFamtableItemRemove() function
 - description 9-7
- ProFamtableItemToModelitem() function
 - description 9-7
- ProFamtableItemToParameter() function
 - description 9-7

- ProFamtableItemVisit() function
 - description 9-3
 - used in a code example 9-8
- ProFamtableShow() function
 - description 9-2
- ProFeature object
 - description 4-2
- ProFeatureAnnotationelemsVisit() function
 - description 22-5
- ProFeatureChildrenGet() function
 - description 4-5
- ProFeatureCopyinfoGet() function
 - description 4-6
- ProFeatureCopyPostAction callback 65-5
- ProFeatureCreate() function
 - calling 29-18
 - description 4-5, 19-4
 - manufacturing objects 53-9
 - used to assemble components 47-17
- ProFeatureCreateOptions enum
 - description 29-19
- ProFeatureCreatePostAction callback 65-5
- ProFeatureCreatePreAction callback 65-5
- ProFeatureDelete() function
 - deleting cross sections 11-7
 - description 4-7, 45-13
- ProFeatureDeletePostAction callback 65-5
- ProFeatureDeletePreAction callback 65-5
- ProFeatureElemDimensionIdGet() function
 - description 29-4
- ProFeatureElemIsIncomplete() function
 - description 29-27
- ProFeatureElemIsVisible() function
 - description 29-27
- ProFeatureElemtreeCreate() function
 - description 29-26
- ProFeatureElemtreeExtract() function
 - description 29-27
 - with NC sequences 53-39
- ProFeatureElemtreeFree() function
 - description 29-27
- ProFeatureElemValueGet() function
 - description 29-27
- ProFeatureElemValuesGet() function
 - description 29-27
- ProFeatureGeomitemVisit() function 4-6
 - description 5-5
 - to create an operation 53-34
 - to create NC sequences 53-38
 - used with axis datums 5-10
 - used with composite datum curves 5-14
 - used with coordinate system datums 5-10
 - used with datum curves 5-13
 - used with datum planes 5-11
 - used with datum points 5-14
- ProFeatureGrppatternStatusGet() function

- description 4-14
- ProFeatureHasGeomChecks() function
 - description 4-6
- ProFeatureInit() function
 - description 4-2
- ProFeatureInsertModeActivate() function
 - description 4-8
 - used in a code example 4-9
- ProFeatureInsertModeIsActive() function 4-6
- ProFeatureInsertmodeWithoptionsActivate()
 - function
 - description 4-12
- ProFeatureIsEmbedded() function
 - description 4-6
- ProFeatureIsIncomplete() function
 - description 4-5
- ProFeatureIsNcseq() function
 - description 4-5
- ProFeatureIsReadonly() function
 - description 4-6
- ProFeatureModeCancel() function 4-6
- ProFeatureNeedsRegenGet callback 65-5
- ProFeatureNumberGet() function
 - description 4-10
 - used in a code example 4-11
- ProFeatureNumSectionsGet() function
 - description 40-26
- ProFeatureParentsGet() function
 - description 4-5
- ProFeaturePatternGet() function
 - description 4-14
- ProFeaturePatternStatusGet() function
 - description 4-14
- ProFeatureReadonlySet() function
 - description 4-9
- ProFeatureReadonlyUnset() function
 - description 4-9
- ProFeatureRedefine() function
 - description 4-8
- ProFeatureRedefinePostAction callback 65-5
- ProFeatureRedefinePreAction callback 65-5
- ProFeatureRegenFailureAction callback 65-5
- ProFeatureRegenPostAction callback 65-5
- ProFeatureRegenPreAction callback 65-5
- ProFeatureReorder() function
 - description 4-10
 - used in a code example 4-11
- ProFeatureReplacePostAction callback 65-5
- ProFeatureReplacePreAction callback 65-5
- ProFeatureReroutePostAction callback 65-5
- ProFeatureReroutePreAction callback 65-5
- ProFeatureSectionCopy() function
 - description 40-26
- ProFeatureSelectionGet() function
 - description 4-5
- ProFeatureSketchAdd() function
 - description 41-10
- ProFeatureSketchedCreate() function
 - description 41-10
- ProFeatureSolidGet() function
 - description 4-5
- ProFeatureStatusflagsGet() function
 - description 4-4
- ProFeatureStatusGet() function
 - description 5-5
 - possible status values 4-4
- ProFeatureSubtypeGet() function
 - defined 4-4
- ProFeatureSuppress() function
 - description 4-7
- ProFeatureSuppressPostAction callback 65-5
- ProFeatureSuppressPreAction callback 65-5
- ProFeatureTypeGet() function
 - description 4-3
 - to visit geometry 5-5
 - used with assemblies 45-5
- ProFeatureTypeNameGet() function
 - description 4-4
- ProFeatureVerstampGet() function
 - description 2-13
- ProFeatureVisibilityGet() function
 - description 4-2
 - visiting geometry 5-5
- ProFeatureWithoptionsCreate() function
 - description 4-12
- ProFeatureWithoptionsDelete() function
 - description 4-12
- ProFeatureWithoptionsRedefine() function
 - description 4-12
- ProFeatureWithoptionsReorder() function
 - description 4-12
- ProFeatureWithoptionsResume() function
 - description 4-12
- ProFeatureWithoptionsSuppress() function
 - description 4-12
- profem_get_bar_elements() function
 - description 63-10
- profem_get_con_case_names() function
 - description 63-5
- profem_get_constraints() function
 - description 63-7
- profem_get_contacts() function
 - description 63-10
- profem_get_mass_elements() function
 - description 63-11
- profem_get_mesh_controls() function
 - description 63-5
- profem_get_shell_pairs() function
 - description 63-12
- ProFileCopyToWS() function
 - description 25-21, H-6
- ProFileEdit() function

- description 12-6
- ProFilenameParse() function
 - description 12-6
- ProFileOpen() function
 - description 12-5
- ProFileOpenOKAction callback 65-3
- ProFileOpenRegister() function
 - description 12-5
- ProFileSave() function
 - description 12-5
- ProFileSaveRegister() function
 - description 12-5
- ProFilesList() function
 - description 12-4
- ProForeignCrvCont enum
 - description 68-6
- ProForeignCurveClassEvalSet() function
 - description 68-7
- ProForeignCurveEvalFunction() typedef
 - description 68-5
- ProGeometryAtPointFind() function
 - description 5-20
- ProGeomitem
 - declaration 1-51
 - description 2-8
 - hierarchy 5-2
 - list of types 5-2
- ProGeomitemAngleEval() function
 - description 5-33
- ProGeomitemdata structure
 - declaration 5-21
- ProGeomitemdataFree() function
 - description 5-22
- ProGeomitemdataGet() function
 - description 5-21
- ProGeomitemDiameterEval() function
 - description 5-33
- ProGeomitemDistanceEval() function
 - description 5-33, 19-4
- ProGeomitemFeatureGet() function 4-6
 - description 5-10
 - used with selection I-14
- ProGeomitemIsInactive() function
 - description 5-5
 - used with axis datums 5-10
 - used with coordinate system datums 5-10
 - used with datum curves 5-13
 - used with datum planes 5-11
 - used with datum points 5-14
 - used with quilts 5-12
- ProGeomitemSetdatumClear() function
 - description 22-58
- ProGeomitemSetdatumGet() function
 - description 22-58
- ProGeomitemSetdatumSet() function
 - description 22-58

- ProGeomitemSetdatumtagGet() function
 - description 22-57
- ProGeomSimpRepRetrieve() function
 - description 49-6
- ProGlobalInterfCalcPostAction callback 65-6
- progr_color() function
 - equivalent I-19
- progr_display_object() function
 - equivalent I-18
- progr_draw_3d() function
 - equivalent I-19
- progr_draw_polygon_2d() function
 - equivalent I-19
- progr_get_color_map() function
 - equivalent I-19
- progr_get_default_font_id() function
 - equivalent I-19
- progr_get_font_id() function
 - equivalent I-19
- progr_get_font_name() function
 - equivalent I-19
- progr_get_line_style_def() function
 - equivalent I-19
- progr_invalidate_display_list() function
 - equivalent I-22
- progr_move_3d() function
 - equivalent I-19
- progr_put_arc() function
 - equivalent I-19
- progr_put_circle() function
 - equivalent I-19
- progr_put_multi_polylines() function
 - equivalent I-19
- progr_put_polyline() function
 - equivalent I-19
- progr_put_text() function
 - equivalent I-19
- progr_set_color_map() function
 - equivalent I-19
- progr_set_draw_mode() function
 - equivalent I-19
- progr_set_line_style() function
 - equivalent I-19
- progr_text_color() function
 - equivalent I-19
- ProGraphicsArcDraw() function
 - description 18-16
- ProGraphicsCircleDraw() function
 - description 18-16
 - used in a code example 18-17
- ProGraphicsColorModify() function
 - description 18-10, 18-11
- ProGraphicsColorSet() function
 - displaying graphics 18-15
 - used in a code example 18-17
- ProGraphicsLineDraw() function

description 18-16
 used in a code example 18-17
 ProGraphicsModeSet() function 18-15
 used in a code example 18-21
 ProGraphicsMultipolylinesDraw() function
 description 18-16
 ProGraphicsPenPosition() function
 description 18-16
 used in a code example 18-17
 ProGraphicsPolygonDraw() function 18-15
 ProGraphicsPolylineDraw() function
 description 18-16
 ProGraphicsSimpRepRetrieve() function
 description 49-6
 ProGraphicsTextDisplay() function
 description 18-17
 ProGraphicWindowSizeGet() function
 description 18-2
 ProGroupUngroupPostAction callback 65-5
 ProGroupUngroupPreAction callback 65-5
 ProGtol*Set function
 description 23-8
 ProGtolCreate function
 description 23-8
 ProGtoldataAllAroundGet function
 description 23-6
 ProGtoldataAlloc function
 description 23-7
 ProGtoldataASMEY1441Get() function
 description 23-6
 ProGtoldataCompositeGet function
 description 23-5
 ProGtoldataDiameterGet function
 description 23-6
 ProGtoldataDispositionValGet() function
 description 23-6
 ProGtoldataFree function
 description 23-16
 ProGtoldataFreeStateGet function
 description 23-6
 ProGtolDataGet function
 description 23-4
 ProGtoldataGtoldatumrefGet function
 description 23-5
 ProGtoldataGtoldatumrefSet function
 description 23-9
 ProGtoldataMatCondGet function
 description 23-5
 ProGtoldataModelGet function
 description 23-4
 ProGtoldataNumdigitsGet() function
 description 23-16
 ProGtoldataNumdigitsSet() function
 description 23-16
 ProGtoldataOwnerGet function
 description 23-4
 ProGtoldataPerUnitGet function
 description 23-5
 ProGtoldataPlacementGet function
 description 23-4
 ProGtoldataPlacementSet function
 description 23-9
 ProGtoldataProfBoundaryGet function
 description 23-6
 ProGtoldataProjZoneGet function
 description 23-6
 ProGtoldataReferenceGet function
 description 23-4
 ProGtoldataStatTolGet function
 description 23-6
 ProGtoldataStatus
 defined 23-9
 ProGtoldataTangentPlaneGet function
 description 23-6
 ProGtoldataTypeGet function
 description 23-4
 ProGtoldataUnitareaStyleGet() function
 description 23-5
 ProGtoldataUnitareaStyleSet() function
 description 23-9
 ProGtoldataValueGet function
 description 23-5
 ProGtoldatumrefAlloc function
 description 23-9
 ProGtoldatumrefFree function
 description 23-9
 ProGtoldatumrefGet function
 description 23-5
 ProGtolDelete function
 description 23-16
 ProGtolEditCancel function
 description 23-15
 ProGtolEditCommit function
 description 23-15
 ProGtolEditInit function
 description 23-15
 ProGtolElbowlengthGet() function
 description 23-17
 ProGtolleader
 allocated and freed 23-9
 defined 23-9
 ProGtolleaderAlloc function
 description 23-9
 ProGtolleaderFree function
 description 23-9
 ProGtolLineEnvelopeGet() function
 description 23-17
 ProGtolPrefixGet() function
 description 23-18
 ProGtolPrefixSet() function
 description 23-18
 ProGtolRightTextGet() function

- description 23-17
- ProGtolRightTextSet() function
 - description 23-17
- ProGtolSuffixGet() function
 - description 23-19
- ProGtolSuffixSet() function
 - description 23-19
- ProGtolTextstyleGet() function
 - description 23-18
- ProGtolTextstyleSet() function
 - description 23-18
- ProGtoltextTextstyleGet() function
 - description 23-18
- ProGtoltextTextstyleSet() function
 - description 23-18
- ProGtolTopTextGet() function
 - description 23-17
- ProGtolTopTextSet() function
 - description 23-17
- ProGtolValueSet() function
 - description 23-16
- ProImportfeatCreate() function
 - description 27-3, 27-14, 27-15
- ProImportfeatDataGet() function
 - description 27-18
- ProImportfeatIdArrayCreate() function
 - description 27-17
- ProImportfeatIdArrayFree() function
 - description 27-18
- ProImportfeatIdArrayMapCount() function
 - description 27-17
- ProImportfeatIdArrayMapGet() function
 - description 27-17
- ProImportfeatItemIdToUserId() function
 - description 27-18
- ProImportfeatRedefine() function
 - redefined 27-17
- ProImportfeatRedefSourceInit() function
 - description 27-16
- ProImportfeatUserIdToItemId() function
 - description 27-18
- ProImportfeatWithProfileCreate() function
 - description 27-3, 27-15
- ProInfoWindowDisplay() function
 - description 12-6
- ProInputFileRead() function
 - description 26-46, 51-6
 - for cable parameters 58-14
 - used with cabling 58-9
- ProInsertModeCancel() function
 - description 4-8
 - used in a code example 4-9
- ProInsertModeIsActive() function
 - description 4-6
 - used in a code example 4-9
- ProInsertmodeWithoptionsCancel() function

- description 4-12
- prointerface_2d() function
 - description 28-3
- prointerface_create() function
 - description 28-3
- prointerface_load_function() function
 - description 28-3
- prointerface_object_set() function
 - description 28-3
- ProIntf3DFileWrite function
 - description 26-23
- ProIntfDataAccuracyGet() function
 - description 27-19
- ProIntfDataAccuracytypeGet() function
 - description 27-19
- ProIntfDataAlloc() function
 - description 27-4
- ProIntfDataDatumAppend() function
 - description 27-14
- ProIntfDataEdgeAppend() function
 - description 27-12
- ProIntfDataFree() function
 - description 27-15
- ProIntfDataOutlineGet() function
 - description 27-19
- ProIntfDataQuiltAppend() function
 - description 27-12
- ProIntfDataSourceInit() function
 - description 27-2, 27-14
- ProIntfDataSurfaceAppend() function
 - description 27-8
- ProIntfImportLayerFilter() function
 - description 26-51
- ProIntfimportModelCreate() function
 - description 26-50
- ProIntfimportModelWithProfileCreate() function
 - description 26-51
- ProIntfimportSourceTypeGet() function
 - description 26-50
- ProIntfSliceFileExport() function
 - description 26-28
- ProItemerror structure
 - declaration 29-20
- ProJlinkApplicationStart() function
 - description 69-17
- ProJlinkApplicationStop() function
 - description 69-18
- ProJlinkTaskExecute() function
 - description 69-18
- ProKinDragEnvironmentSet() function
 - description 48-2
- ProKinDragPerformMove() function
 - description 48-4
- ProKinDragReconnect() function
 - description 48-4

ProKinDragSessionInquire() function	ProLayerItemRemove() function
description 48-2	description 2-15
ProKinDragSnapshotsNamesGet() function	ProLayerItemsGet() function
description 48-5	description 2-14
ProKinDragStart() function	ProLayerItemsPopulate() function
description 48-2	description 2-15
ProKinDragStop() function	ProLayerViewDependencyGet() function
description 48-2	description 2-17
prolayer_add_item() function	ProLayerViewDependencySet() function
equivalent I-20	description 2-16
prolayer_create_layer() function	ProLinedataInit function
equivalent I-20	description 27-10
prolayer_delete_layer() function	ProLinedataInit() function
equivalent I-20	description 51-55
prolayer_display() function	ProLinestyledefGet() function
equivalent I-20	description 18-15
prolayer_get_display() function	ProLinestyleSet() function
equivalent I-20	description 18-15
prolayer_get_items() function	displaying graphics 18-15
equivalent I-20	proload_cmd_sequence() function
prolayer_get_names() function	equivalent I-17
equivalent I-20	ProLocalGroupCreate() function
prolayer_remove_item() function	description 4-19
equivalent I-20	ProMacroExecute() function
ProLayerCreate() function	description 15-70
description 2-14	ProMacroLoad() function
ProLayerDefLayerGet() function	description 15-69
description 2-16	ProMassProperty structure
ProLayerDefLayerSet() function	description 3-18
description 2-16	ProMaterialCreate() function
ProLayerDelete() function	description 3-22
description 2-14	ProMaterialCurrentGet() function
ProLayerDisplay enum	description 3-22
description 2-15	ProMaterialCurrentSet() function
ProLayerDisplaystatusGet() function	description 3-22
description 2-15	ProMaterialDelete() function
ProLayerDisplaystatusSave() function	description 3-22
description 2-16	ProMaterialDescriptionGet() function
ProLayerDisplaystatusSet() function	description 3-26
description 2-16	ProMaterialDescriptionSet() function
ProLayerfilterdataActionSet() function	description 3-26
description 26-51	ProMaterialfileRead() function
ProLayerfilterdataCountsGet() function	description 3-27
description 26-51	ProMaterialfileWrite() function
ProLayerfilterdataNameGet() function	description 3-27
description 26-51	ProMaterialPropertyGet() function
ProLayerfilterdataNameSet() function	description 3-25
description 26-51	ProMaterialPropertySet() function
ProLayerItemAdd() function	description 3-22, 3-25
description 2-15	ProMatrixInit() function
ProLayeritemarrayFree() function	description 8-12
description 2-15	ProMdl object
ProLayerItemInit() function	declaration 1-50
description 2-15	description 2-3
ProLayeritemLayersGet() function	ProMdlBackup() function 2-7
description 2-15	ProMdlCommonnameGet() function

- description 2-4
- ProMdlCommonnameSet() function
 - description 2-4
- ProMdlCopy() function 2-7
- ProMdlCopyPostAction callback 65-3
- ProMdlCopyPostAllAction callback 65-4
- ProMdlCopyPreAction callback 65-3
- ProMdlCreateCancelAction callback 65-4
- ProMdlCreatePostAction callback 65-3
- ProMdlCreatePreAction callback 65-3
- ProMdlCurrentGet() function
 - description 2-6
- ProMdlDataGet() function
 - description 2-5
- ProMdlDbmsFailureAction callback 65-4
- ProMdlDeclaredList() function
 - description 2-6
- ProMdlDelete() function 2-7
- ProMdlDeletePostAction callback 65-3
- ProMdlDeletePostAllAction callback 65-4
- ProMdlDeletePreAction callback 65-3
- ProMdlDependenciesList() function
 - description 2-6
- ProMdlDisplayPostAction callback 65-6
- ProMdlDisplayPreAction callback 65-6
- ProMdlErase() function 2-7
 - description 2-8
- ProMdlEraseAll() function
 - defined 2-8
- ProMdlEraseNotDisplayed() function
 - description 2-8
- ProMdlErasePostAction callback 65-3
- ProMdlErasePostAllAction callback 65-4
- ProMdlErasePreAction callback 65-3
- ProMdlFileCopy() function
 - description 2-7
- ProMdlGtolvisit function
 - description 23-2
- ProMdlIdGet() function
 - description 2-5
- ProMdlInit() function
 - description 2-5
- ProMdlIsModifiable() function
 - description 2-6
- ProMdlIsSaveAllowed() function
 - description 2-8, H-6
- ProMdlLayerGet() function
 - description 2-14
- ProMdlLayersCollect() function
 - description 2-14
- ProMdlLayerVisit() function
 - description 2-14
- ProMdlLoad() function
 - description 25-16
- ProMdlModificationVerify() function
 - description 2-6

- ProMdlMultipleRetrieve() function
 - defined 2-7
- ProMdlNameGet() function
 - description 2-4
- ProMdlNoteVisit() function
 - description 22-52
- ProMdlParamtablesetsCollect() function
 - description 7-12
- ProMdlPrincipalunitsystemGet() function
 - description 3-13
- ProMdlPrincipalunitsystemSet() function
 - description 3-17
- ProMdlPurgePostAction callback 65-3
- ProMdlPurgePreAction callback 65-3
- ProMdlRename() function 2-7
- ProMdlRenamePostAction callback 65-3
- ProMdlRenamePreAction callback 65-3
- ProMdlRetrieve() function
 - defined 2-7
 - description 2-7, 25-15
 - used in a code example 3-4
- ProMdlRetrievePostAction callback 65-3
- ProMdlRetrievePostAllAction callback 65-4
- ProMdlRetrievePreAction callback 65-3
- ProMdlSave() function 2-7
 - description 25-11
- ProMdlSavePostAction callback 65-3
- ProMdlSavePostAllAction callback 65-4
- ProMdlSavePreAction callback 65-3
- ProMdlStartAction callback 65-3
- ProMdlSubtypeGet() function
 - defined 2-5
- ProMdlToModelitem() function
 - description 2-5
 - used with parameters 7-4
- ProMdlTypeGet() function
 - description 2-4
 - used with assemblies 45-6
- ProMdlUnitsCollect() function
 - description 3-15
- ProMdlUnitsystemCreate() function
 - description 3-14
- ProMdlUnitsystemsCollect() function
 - description 3-13
- ProMdlVerstampGet() function
 - description 2-13
- ProMechadvancedmassdataAlloc() function
 - description 61-77
- ProMechadvancedmassdataCsysGet()
 - function
 - description 61-77
- ProMechadvancedmassdataCsysSet() function
 - description 61-77
- ProMechadvancedmassdataPropertiesGet()
 - function
 - description 61-77

ProMechadvancedmassdataPropertiesSet()	description 61-69
functions	ProMechbeamorientdataTypeGet() function
description 61-77	description 61-69
ProMechanicaEnter() function	ProMechbeamorientdataTypeSet() function
description 61-3	description 61-69
ProMechanicaIsActive() function	ProMechbeamorientdataVectorGet() function
description 61-3	description 61-69
ProMechanicaLeave() function	ProMechbeamorientdataVectorSet() function
description 61-3	description 61-69
ProMechbeamBeamdataGet() function	ProMechbeamorientDescriptionGet() function
description 61-58	description 61-69
ProMechbeamBeamdataSet() function	ProMechbeamorientDescriptionSet() function
description 61-58	description 61-69
ProMechbeamdataAlloc() function	ProMechbeamReferencesGet() function
description 61-58	description 61-58
ProMechbeamdataCompcrvrelGet() function	ProMechbeamreleasedataAlloc() function
description 61-59	description 61-70
ProMechbeamdataCompcrvrelSet() function	ProMechbeamreleasedataFree() function
description 61-59	description 61-71
ProMechbeamdataFree() function	ProMechbeamreleaseDataGet() function
description 61-60	description 61-70
ProMechbeamdataSidesGet() function	ProMechbeamsectionDataGet() function
description 61-59	description 61-62
ProMechbeamdataSidesSet() function	ProMechbeamsectionDataSet() function
description 61-59	description 61-62
ProMechbeamdataStressrecoveryGet()	ProMechbeamsectionDescriptionGet()
function	function
description 61-59	description 61-64
ProMechbeamdataStressrecoverySet()	ProMechbeamsectionDescriptionSet()
function	function
description 61-59	description 61-64
ProMechbeamdataXyshearGet() function	ProMechbeamsideAlloc() function
description 61-59	description 61-58
ProMechbeamdataXyshearSet() function	ProMechbeamsideFree() function
description 61-59	description 61-59
ProMechbeamdataXzshearGet() function	ProMechbeamsideOrientGet() function
description 61-59	description 61-59
ProMechbeamdataXzshearSet() function	ProMechbeamsideOrientSet() function
description 61-59	description 61-59
ProMechbeamdataYdirectionGet() function	ProMechbeamsideProarrayFree() function
description 61-60	description 61-59
ProMechbeamdataYdirectionSet() function	ProMechbeamsideReleaseGet() function
description 61-60	description 61-59
ProMechbeamMaterialGet() function	ProMechbeamsideReleaseSet() function
description 61-58	description 61-59
ProMechbeamMaterialIdSet() function	ProMechbeamsideSectionGet() function
description 61-58	description 61-59
ProMechbeamorientdataAngleGet() function	ProMechbeamsideSectionSet() function
description 61-69	description 61-59
ProMechbeamorientdataAngleSet() function	ProMechbeamTrussdataGet() function
description 61-69	description 61-58
ProMechbeamorientdataFree() function	ProMechbeamTrussdataSet() function
description 61-70	description 61-58
ProMechbeamorientDataGet() function	ProMechbeamTypeGet() function
description 61-69	description 61-58
ProMechbeamorientDataSet() function	ProMechbearingdataFree() function

description 61-31
 ProMechbearingdataValue Set() function
 description 61-31
 ProMechbearingdataValueGet() function
 description 61-31
 ProMechbondinterfacedataMergenodesGet()
 function
 description 61-102
 ProMechbondinterfacedataMergenodesSet()
 function
 description 61-102
 ProMechcentrifugaldataAccelerationGet()
 function
 description 61-32
 ProMechcentrifugaldataAccelerationSet()
 function
 description 61-32
 ProMechcentrifugaldataAlloc() function
 description 61-32
 ProMechcentrifugaldataVelocityGet() function
 description 61-32
 ProMechcentrifugaldataVelocitySet() function
 description 61-32
 ProMechcomponentmassdataAlloc() function
 description 61-77
 ProMechcomponentmassdataComponentGet()
 function
 description 61-77
 ProMechcomponentmassdataComponentSet()
 function
 description 61-77
 ProMechconstrConstrsetAssign() function
 description 61-42, 61-56
 ProMechconstrConstrsetsGet() function
 description 61-41
 ProMechconstrConvectiondataGet() function
 description 61-40
 ProMechconstrConvectiondataSet() function
 description 61-42
 ProMechconstrDisplacementdataGet()
 function
 description 61-40
 ProMechconstrDisplacementdataSet()
 function description 61-42
 ProMechconstrRadiationdataSet() function
 description 61-42
 ProMechconstrReferencesGet() function
 description 61-41
 ProMechconstrReferencesSet() function
 description 61-42
 ProMechconstrsetConstrsGet() function
 description 61-56
 ProMechconstrsetDescriptionGet() function
 description 61-56
 ProMechconstrsetDescriptionSet() function
 description 61-56

ProMechconstrsetTypeGet() function
 description 61-56
 ProMechconstrSymmetrydataGet() function
 description 61-40
 ProMechconstrSymmetrydataSet() function
 description 61-42
 ProMechconstrTemperaturedataGet()
 function
 description 61-41
 ProMechconstrTemperaturedataSet() function
 description 61-42
 ProMechconstrThermalsymmetrydataGet()
 function
 description 61-41
 ProMechconstrTypeGet() function
 description 61-40
 ProMechcontactinterfacedataAnglebetweensu
 rfacesExprSet() function
 description 61-104
 ProMechcontactinterfacedataAnglebetweensu
 rfacesGet() function
 description 61-104
 ProMechcontactinterfacedataCheckonlyplana
 rGet() function
 description 61-104
 ProMechcontactinterfacedataCheckonlyplana
 rSet() function
 description 61-104
 ProMechcontactinterfacedataCoefffrictionGet(
) function
 description 61-105
 ProMechcontactinterfacedataCoefffrictionSet(
) function
 description 61-105
 ProMechcontactinterfacedataCompatiblemes
 hGet() function
 description 61-105
 ProMechcontactinterfacedataCompatiblemes
 hSet() function
 description 61-105
 ProMechcontactinterfacedataInfiniteFriction
 Get() function
 description 61-105
 ProMechcontactinterfacedataInfiniteFriction
 Set() function
 description 61-105
 ProMechcontactinterfacedataSeparationdista
 nceExprGet() function
 description 61-103, 61-104
 ProMechcontactinterfacedataSeparationdista
 nceExprSet() function
 description 61-104
 ProMechcontactinterfacedataSeparationdista
 nceGet() function
 description 61-104
 ProMechcontactinterfacedataSeparationdista

nceSet() function
 description 61-104
 ProMechcontactinterfacedataSlippageGet()
 function
 description 61-105
 ProMechcontactinterfacedataSlippageSet()
 function
 description 61-105
 ProMechcontactinterfacedataSplitsurfacesGet()
 function
 description 61-105
 ProMechcontactinterfacedataSplitsurfacesSet()
 function
 description 61-105
 ProMechcontourAreaEval() function
 description 62-11
 ProMechcontourContainingContourGet()
 function
 description 62-11
 ProMechcontourEdgeVisit() function
 description 62-11
 ProMechcontourFilterAction() function
 description 62-10
 ProMechcontourIdGet() function
 description 62-10
 ProMechcontourSurfaceGet() function
 description 62-10
 ProMechcontourTraversalGet() function
 description 62-10
 ProMechcontourUvpntVerify() function
 description 62-11
 ProMechcontourVisitAction() function
 description 62-10
 ProMechconvectiondataAlloc() function
 description 61-43
 ProMechconvectiondataBulktempGet()
 function
 description 61-43
 ProMechconvectiondataBulktempSet()
 function
 description 61-43
 ProMechconvectiondataFilmcoefficientGet()
 function
 description 61-43
 ProMechconvectiondataFilmcoefficientSet()
 function
 description 61-43
 ProMechconvectiondataFree() function
 description 61-43
 ProMechconvectiondataTimefunctionidGet()
 function
 description 61-43
 ProMechconvectiondataTimefunctionidSet()
 function
 description 61-43
 ProMechdirvectorComponentsGet() function

description 61-18
 ProMechdirvectorComponentsSet() function
 description 61-18
 ProMechdirvectorCsysGet() function
 description 61-18
 ProMechdirvectorCsysSet() function
 description 61-18
 ProMechdirvectorVariationGet() function
 description 61-18
 ProMechdisplacementdataAlloc() function
 description 61-44
 ProMechdisplacementdataCsysGet() function
 description 61-45
 ProMechdisplacementdataCsysSet() function
 description 61-45
 ProMechdisplacementdataPinangularconstrT
 typeGet() function
 description 61-47
 ProMechdisplacementdataPinangularconstrT
 typeSet() function
 description 61-47
 ProMechdisplacementdataPinaxialconstrType
 Get() function
 description 61-47
 ProMechdisplacementdataPinaxialconstrType
 Set() function
 description 61-47
 ProMechdisplacementdataPinconstrGet()
 function
 description 61-46
 ProMechdisplacementdataPinconstrSet()
 function
 description 61-47
 ProMechdisplacementdataRotationconstrsGet()
 function
 description 61-45
 ProMechdisplacementdataRotationconstrsSet()
 function
 description 61-45
 ProMechdisplacementdataTranslationconstrs
 Get() function
 description 61-45
 ProMechdisplacementdataTranslationconstrs
 Set() function
 description 61-45
 ProMechdisplacementdataTranslationinterpr
 etinradiansflagGet() function
 description 61-46
 ProMechdisplacementdataTranslationinterpr
 etinradiansflagSet() function
 description 61-46
 ProMechdisplacementdataTypeGet() function
 description 61-45
 ProMechdisplacementdataTypeSet() function
 description 61-45
 ProMechdisplacementpinconstrAlloc()

function
 description 61-46
 ProMechdisplacementpinconstrFree()
 function
 description 61-47
 ProMechdisplacementregularconstrAlloc()
 function
 description 61-45
 ProMechdisplacementregularconstrFree()
 function
 description 61-46
 ProMechdisplacementregularconstrProarrayFree() function
 description 61-46
 ProMechdisplacementregularconstrTypeGet()
 function
 description 61-45
 ProMechdisplacementregularconstrTypeSet()
 function
 description 61-46
 ProMechdisplacementregularconstrValueGet()
) function
 description 61-46
 ProMechdisplacementregularconstrValueSet()
 function
 description 61-46
 ProMechexternalfielddataAlloc() function
 description 61-19
 ProMechexternalfielddataCsysGet() function
 description 61-19
 ProMechexternalfielddataCsysSet() function
 description 61-19
 ProMechexternalfielddataFileGet() function
 description 61-20
 ProMechexternalfielddataFileSet() function
 description 61-20
 ProMechexttmpdataAlloc() function
 description 61-37
 ProMechexttmpdataFemneutralfileGet()
 function
 description 61-37
 ProMechexttmpdataFemneutralfileSet()
 function
 description 61-37
 ProMechexttmpdataFree() function
 description 61-37
 ProMechexttmpdataReftempGet() function
 description 61-37
 ProMechexttmpdataReftempSet() function
 description 61-37
 ProMechforcedataAlloc() function
 description 61-29
 ProMechforcedataAxialForceIgnoreflagGet()
 function
 description 61-30
 ProMechforcedataAxialForceIgnoreflagSet()

function
 description 61-30
 ProMechforcedataAxialMomentIgnoreflagGet()
) function
 description 61-30
 ProMechforcedataAxialMomentIgnoreflagSet()
) function
 description 61-30
 ProMechforcedataForceGet() function
 description 61-30
 ProMechforcedataForceSet() function
 description 61-30
 ProMechforcedataFree() function
 description 61-30
 ProMechforcedataMomentGet() function
 description 61-30
 ProMechforcedataMomentSet() function
 description 61-30
 ProMechforcedataRefpntGet() function
 description 61-30
 ProMechforcedataRefpntSet() function
 description 61-30
 ProMechforcedataTypeGet() function
 description 61-29
 ProMechfreeinterfacedataSplitsurfacesGet()
 function
 description 61-106
 ProMechfreeinterfacedataSplitsurfacesSet()
 function
 description 61-106
 ProMechfuncCsysGet() function
 description 61-13
 ProMechfuncdataExpressionGet() function
 description 61-13
 ProMechfuncdataFree() function
 description 61-14
 ProMechfuncdataFuncTableGet() function
 description 61-14
 ProMechfuncdataFuncvalueinterpGet()
 function
 description 61-13
 ProMechfuncDataGet() function
 description 61-13
 ProMechfuncdataIndependentvarGet()
 function
 description 61-14
 ProMechfuncdataIndepvarinterpGet()
 function
 description 61-14
 ProMechfuncdataTypeGet() function
 description 61-13
 ProMechfuncDescriptionGet() function
 description 61-13
 ProMechgapReferencesGet() function
 description 61-107
 ProMechgapReferencesSet() function

description 61-107
 ProMechgapSimpledataGet() function
 description 61-107
 ProMechgapSimpledataSet() function
 description 61-107
 ProMechgapTypeGet() function
 description 61-107
 ProMechgeneralstndataAlloc() function
 description 61-67
 ProMechgeneralstndataAreaGet() function
 description 61-67
 ProMechgeneralstndataAreaproductGet()
 function
 description 61-67
 ProMechgeneralstndataAreaproductSet()
 function
 description 61-67
 ProMechgeneralstndataAreaSet() function
 description 61-67
 ProMechgeneralstndataFree() function
 description 61-68
 ProMechgeneralstndataMomentsGet()
 function
 description 61-67
 ProMechgeneralstndataMomentsSet()
 function
 description 61-67
 ProMechgeneralstndataShearcenterGet()
 function
 description 61-68
 ProMechgeneralstndataShearcenterSet()
 function
 description 61-68
 ProMechgeneralstndataShearfactorGet()
 function
 description 61-68
 ProMechgeneralstndataShearfactorSet()
 function
 description 61-68
 ProMechgeneralstndataStressrecoverypntsG
 et() function
 description 61-68
 ProMechgeneralstndataStressrecoverypntsS
 et() function
 description 61-68
 ProMechgeneralstndataTorsionstiffnessGet()
 function
 description 61-68
 ProMechgeneralstndataTorsionstiffnessSet()
 function
 description 61-68
 ProMechgeomrefFree() function
 description 61-11
 ProMechgeomrefIdGet() function
 description 61-10
 ProMechgeomrefIdSet() function

description 61-11
 ProMechgeomrefPathGet() function
 description 61-11
 ProMechgeomrefPathSet() function
 description 61-11
 ProMechgeomrefProarrayFree() function
 description 61-11
 ProMechgeomrefSubtypeGet() function
 description 61-9
 ProMechgeomrefTypeGet() function
 description 61-8
 ProMechglobaltempdataAlloc() function
 description 61-34
 ProMechglobaltempdataFree() function
 description 61-34
 ProMechglobaltempdataReftempGet()
 function
 description 61-34
 ProMechglobaltempdataReftempSet()
 function
 description 61-34
 ProMechglobaltempdataValueGet() function
 description 61-34
 ProMechglobaltempdataValueSet() function
 description 61-34
 ProMechgravitydataAlloc() function
 description 61-31
 ProMechgravitydataFree() function
 description 61-31
 ProMechgravitydataValueGet() function
 description 61-31
 ProMechgravitydataValueSet() function
 description 61-31
 ProMechheatdataAlloc() function
 description 61-38
 ProMechheatdataFree() function
 description 61-39
 ProMechheatdataTimefunctionidGet()
 function
 description 61-38
 ProMechheatdataTimefunctionidSet()
 function
 description 61-38
 ProMechheatdataTypeGet() function
 description 61-38
 ProMechheatdataTypeSet() function
 description 61-38
 ProMechheatdataValueGet() function
 description 61-38
 ProMechheatdataValueSet() function
 description 61-38
 ProMechinterfacebonddataAlloc() function
 description 61-101
 ProMechinterfaceBonddataGet() function
 description 61-101
 ProMechinterfaceBonddataSet() function

description 61-102	ProMechloadForcedataSet() function
ProMechinterfacecontactdataAlloc() function	description 61-22
description 61-102	ProMechloadGlobaltempdataGet() function
ProMechinterfaceContactdataGet() function	description 61-21
description 61-102	ProMechloadGlobaltempdataSet() function
ProMechinterfaceContactdataSet() function	description 61-23
description 61-102	ProMechloadGravitydataGet() function
ProMechinterfaceFreedataGet() function	description 61-21
description 61-102	ProMechloadGravitydataSet() function
ProMechinterfaceFreedataSet() function	description 61-23
description 61-102	ProMechloadHeatdataGet() function
ProMechinterfaceReferencesGet() function	description 61-21
description 61-101	ProMechloadHeatdataSet() function
ProMechinterfacethrresistdataAlloc() function	description 61-23
description 61-102	ProMechloadLoadsetAssign() function
ProMechinterfaceTypeGet() function	description 61-23, 61-39
description 61-101	ProMechloadMecttempdataGet() function
ProMechinterpolationpntAlloc() function	description 61-21
description 61-19	ProMechloadMecttempdataSet() function
ProMechinterpolationpntFree() function	description 61-23
description 61-19	ProMechloadPressuredataGet() function
ProMechinterpolationpntMagnitudeGet()	description 61-21
function	ProMechloadPressuredataSet() function
description 61-19	description 61-23
ProMechinterpolationpntMagnitudeSet()	ProMechloadReferencesGet() function
function	description 61-21
description 61-19	ProMechloadReferencesSet() function
ProMechinterpolationpntPointGet() function	description 61-23
description 61-19	ProMechloadsetDescriptionGet() function
ProMechinterpolationpntPointSet() function	description 61-39
description 61-19	ProMechloadsetDescriptionSet() function
ProMechinterpolationpntProarrayFree()	description 61-39
function	ProMechloadsetLoadsGet() function
description 61-19	description 61-39
ProMechitemFilterAction() function	ProMechloadsetTypeGet() function
description 61-7	description 61-39
ProMechitemNameGet function	ProMechloadStructtempdataGet() function
description 61-6	description 61-21
ProMechitemStatusGet() function	ProMechloadStructtempdataSet() function
description 61-7	description 61-23
ProMechitemVisitAction() function	ProMechLoadTypeGet() function
description 61-7	description 61-20
ProMechloadBearingdataGet() function	ProMechmassAdvanceddataGet() function
description 61-21	description 61-76
ProMechloadBearingdataSet() function	ProMechmassComponentdataGet() function
description 61-23	description 61-76
ProMechloadCentrifugaldataGet() function	ProMechmassComponentdistributeddataSet()
description 61-21	function
ProMechloadCentrifugaldataSet() function	description 61-76
description 61-23	ProMechmassComponentpointdataSet()
ProMechloadExttempdataGet() function	function
description 61-21	description 61-76
ProMechloadExttempdataSet() function	ProMechmassDistributionGet() function
description 61-23	description 61-75
ProMechloadForcedataGet() function	ProMechmassDistributionSet() function
description 61-21	description 61-75

ProMechmasspropsDescriptionGet() function description 61-78	ProMechmaterialorientdataFirstdirectionSet() function description 61-80
ProMechmasspropsMassGet() function description 61-78	ProMechMaterialorientDataGet() function description 61-79
ProMechmasspropsMassSet() function description 61-78	ProMechmaterialorientdataObjecttypeGet() function description 61-80
ProMechmasspropsmassSet() function description 61-78	ProMechmaterialorientdataObjecttypeSet() function description 61-80
ProMechmasspropsMomentsGet() function description 61-78	ProMechmaterialorientdataProjectiondataGet() () function description 61-80
ProMechmasspropsMomentsSet() function description 61-78	ProMechmaterialorientdataProjectiondataSet() () function description 61-80
ProMechmassReferencesGet() function description 61-75	ProMechmaterialorientdataRotationGet() function description 61-81
ProMechmassReferencesSet() function description 61-75	ProMechmaterialorientdataSeconddirectionS et() function description 61-80
ProMechmassSimplifieddataGet() function description 61-76	ProMechmaterialorientDataSet() function description 61-79
ProMechmassSimplifiedDataSet() function description 61-76	ProMechmaterialorientdataSurfacerotatG et() function description 61-81
ProMechmassTypeGet() function description 61-75	ProMechmaterialorientdataSurfacerotatU nset() function description 61-81
ProMechmaterialorientcsysCsysGet() function description 61-81	ProMechmaterialorientdataTypeGet() function description 61-80
ProMechmaterialorientcsysCsysSet() function description 61-81	ProMechmaterialorientDescriptionGet() function description 61-79
ProMechmaterialorientcsysProjectiontypeGet() () function description 61-81	ProMechmaterialorientDescriptionSet() function description 61-79
ProMechmaterialorientcsysProjectiontypeSet() () function description 61-82	ProMechmaterialorientprojPointsGet() function description 61-81
ProMechmaterialorientcsysXaxisGet() function description 61-82	ProMechmaterialorientprojTypeGet() function description 61-81
ProMechmaterialorientcsysXaxisSet() function description 61-82	ProMechmaterialorientprojXyzvectorGet() function description 61-81
ProMechmaterialorientcsysYaxisGet() function description 61-82	ProMechmaterialorientprojXyzvectorSet() function description 61-81
ProMechmaterialorientcsysYaxisSet() function description 61-82	ProMechmaterialorientReferencesGet() function description 61-80
ProMechmaterialorientcsysZaxisGet() function description 61-82	ProMechmcautogemedgedistrFree() function description 61-114
ProMechmaterialorientcsysZaxisSet() function description 61-82	
ProMechmaterialorientdataCsysdataGet() function description 61-80	
ProMechmaterialorientdataCsysdataSet() function description 61-80	

ProMechmcautogemedgeistrNodesGet() function description 61-113	axSizeGet() function description 61-117
ProMechmcautogemedgeistrNodesSet() function description 61-113	ProMechmcautogemisolateexcludeShellIsolM axSizeSet() function description 61-117
ProMechmcautogemedgeistrRatioGet() function description 61-114	ProMechmcautogemisolateexcludeShellIsolTy peGet() function description 61-118
ProMechmcautogemedgeistrRatioSet() function description 61-114	ProMechmcautogemisolateexcludeShellIsolTy peSet() function description 61-118
ProMechmcautogemedgeistrStrictGet() function description 61-114	ProMechmcautogemisolateexcludeSolidIsolM axSizeGet() function description 61-118
ProMechmcautogemedgeistrStrictSet() function description 61-114	ProMechmcautogemisolateexcludeSolidIsolM axSizeSet() function description 61-118
ProMechmcautogemedgeelencrvAlloc() function description 61-115	ProMechmcautogemisolateexcludeSolidIsolTy peGet() function description 61-118
ProMechmcautogemedgeelencrvIgnoreRadius Get() function description 61-115	ProMechmcautogemisolateexcludeSolidIsolTy peSet() function description 61-118
ProMechmcautogemedgeelencrvIgnoreRadiusS et function description 61-115	ProMechmcautogemminedgeEdgesGet() function description 61-114
ProMechmcautogemedgeelencrvMinradiusGet() function description 61-115	ProMechmcautogemminedgeEdgesSet() function description 61-114
ProMechmcautogemedgeelencrvMinradiusSet() function description 61-115	ProMechmcautogemminedgeFree() function description 61-114
ProMechmcautogemedgeelencrvRatioGet() function description 61-115	ProMechmcautogemminedgeLengthGet() function description 61-114
ProMechmcautogemedgeelencrvRatioSet() functions description 61-115	ProMechmcautogemminedgeLengthSet() function description 61-114
ProMechmcautogemelemsizeAlloc() function description 61-116	ProMechmcdisplacementcsysCsysGet() function description 61-118
ProMechmcautogemelemsizeFree() function description 61-116	ProMechmcdisplacementcsysCsysSet() function description 61-118
ProMechmcautogemelemsizeSizeGet() function description 61-116	ProMechmcedgedistrNodesGet() function description 61-116
ProMechmcautogemelemsizeSizeSet() function description 61-116	ProMechmcedgedistrNodesSet() function description 61-116
ProMechmcautogemisolateexcludeExcludeGet() function description 61-117	ProMechmcedgedistrRatioGet() function description 61-116
ProMechmcautogemisolateexcludeExcludeSet() function description 61-117	ProMechmcedgedistrRatioSet() function description 61-116
ProMechmcautogemisolateexcludeShellIsolM	ProMechmcedgedistrStrictGet() function description 61-117
	ProMechmcedgedistrStrictSet() function description 61-117
	ProMechmcelemsizeSizeGet() function

description 61-119
 ProMechmcelemsizeSizeSet() function
 description 61-119
 ProMechmchardpntNodeGet() function
 description 61-119
 ProMechmchardpntNodeSet() function
 description 61-119
 ProMechmcidoffsetOffsetGet() function
 description 61-120
 ProMechmcidoffsetOffsetSet() function
 description 61-120
 ProMechmcnumberingFirstGet() function
 description 61-120
 ProMechmcnumberingFirstSet() function
 description 61-120
 ProMechmcnumberingIncrementGet()
 function
 description 61-120
 ProMechmcnumberingIncrementSet()
 function
 description 61-120
 ProMechmcnumberingLastGet() function
 description 61-120
 ProMechmcnumberingLastSet() function
 description 61-120
 ProMechmcshellcsysCsysGet() function
 description 61-119
 ProMechmcshellcsysCsysSet() function
 description 61-119
 ProMechmcshellcsysDirectionGet() function
 description 61-119
 ProMechmcshellcsysDirectionSet() function
 description 61-119
 ProMechmcsuppressTypeGet() function
 description 61-121
 ProMechmcsuppressTypeSet() function
 description 61-121
 ProMechmecttempdataAlloc() function
 description 61-35
 ProMechmecttempdataAnalysisidGet()
 function
 description 61-35
 ProMechmecttempdataAnalysisidSet()
 function
 description 61-35
 ProMechmecttempdataDesignstudyGet 61-36
 ProMechmecttempdataDesignstudyGet()
 function
 description 61-36
 ProMechmecttempdataDesignstudySet()
 function
 description 61-36
 ProMechmecttempdataFree() function
 description 61-36
 ProMechmecttempdataLoadsetGet() function
 description 61-36

ProMechmecttempdataLoadsetidGet()
 function
 description 61-35
 ProMechmecttempdataLoadsetidSet()
 function
 description 61-35
 ProMechmecttempdataLoadsetSet() function
 description 61-36
 ProMechmecttempdataReftempGet() function
 description 61-36
 ProMechmecttempdataReftempSet() function
 description 61-36
 ProMechmecttempdataTimestepGet()
 function
 description 61-36
 ProMechmecttempdataTimestepSet() function
 description 61-36
 ProMechmeshcntrlAutogemedgedistrdataGet()
 function
 description 61-110
 ProMechmeshcntrlAutogemedgedistrdataSet()
 function
 description 61-110
 ProMechmeshcntrlAutogemedgelencrvdataGe
 t() function
 description 61-110
 ProMechmeshcntrlAutogemedgelencrvdataSe
 t() function
 description 61-110
 ProMechmeshcntrlAutogemelemsizedataGet()
 function
 description 61-110
 ProMechmeshcntrlAutogemelemsizedataSet()
 function
 description 61-110
 ProMechmeshcntrlAutogemisolateexcludedat
 aGet() function
 description 61-112
 ProMechmeshcntrlAutogemisolateexcludedat
 aSet() function
 description 61-112
 ProMechmeshcntrlAutogemminedgedataGet()
 function
 description 61-110
 ProMechmeshcntrlAutogemminedgedataSet()
 function
 description 61-110
 ProMechmeshcntrlEdgedistrdataGet()
 function
 description 61-111
 ProMechmeshcntrlEdgedistrdataSet()
 function
 description 61-111
 ProMechmeshcntrlElemsizedataGet()
 function
 description 61-111

ProMechmeshcntrlElmsizedataSet() function description 61-111	ProMechsectiondataChannelsectdataGet() function description 61-63
ProMechmeshcntrlHardpointGet() function description 61-111	ProMechsectiondataChannelsectdataSet() function description 61-63
ProMechmeshcntrlHardpointSet() function description 61-111	ProMechsectiondataCirclesectdataGet() function description 61-63
ProMechmeshcntrlIdsoffsetGet() function description 61-112	ProMechsectiondataCirclesectdataSet() function description 61-63
ProMechmeshcntrlIdsoffsetSet() function description 61-112	ProMechsectiondataDiamondsectdataGet() function description 61-63
ProMechmeshcntrlNumberingGet() function description 61-112	ProMechsectiondataDiamondsectdataSet() function description 61-63
ProMechmeshcntrlNumberingSet() function description 61-112	ProMechsectiondataEllipsesectdataGet() function description 61-64
ProMechmeshcntrlReferencesGet() function description 61-113	ProMechsectiondataEllipsesectdataSet() function description 61-64
ProMechmeshcntrlReferencesSet() function description 61-113	ProMechsectiondataFree() function description 61-65
ProMechmeshcntrlShellcsysGet() function description 61-111	ProMechsectiondataGeneralsectdataGet() function description 61-64
ProMechmeshcntrlShellcsysSet() function description 61-111	ProMechsectiondataGeneralsectdataSet() function description 61-64
ProMechmeshcntrlSuppressGet() function description 61-112	ProMechsectiondataHollowcirclesectdataGet() function description 61-64
ProMechmeshcntrlSuppressSet() function description 61-112	ProMechsectiondataHollowcirclesectdataSet() function description 61-64
ProMechmeshcntrlTypeGet() function description 61-109	ProMechsectiondataHollowellipsesectdataGet() () function description 61-64
ProMechobjectrefAlloc() function description 61-7	ProMechsectiondataHollowellipsesectdataSet() function description 61-64
ProMechobjectrefFree() function description 61-8	ProMechsectiondataHollowrectsectdataGet() function description 61-62
ProMechobjectrefIdGet() function description 61-7	ProMechsectiondataHollowrectsectdataSet() function description 61-62
ProMechobjectrefIdSet() function description 61-7	ProMechsectiondataIbeamsectdataGet() function description 61-63
ProMechobjectrefPathGet() function description 61-8	ProMechsectiondataIbeamsectdataSet() function description 61-63
ProMechobjectrefPathSet() function description 61-8	
ProMechobjectrefProarrayFree() function description 61-8	
ProMechobjectrefTypeGet() function description 61-7	
ProMechobjectrefTypeSet() function description 61-7	
ProMechpressuredataAlloc() function description 61-31	
ProMechpressuredataFree() function description 61-31	
ProMechpressuredataValueGet() function description 61-31	
ProMechpressuredataValueSet() function description 61-31	
ProMechsectiondataAlloc() function description 61-61	

ProMechsectiondataLsectionsectdataGet()	function
description	61-63
ProMechsectiondataLsectionsectdataSet()	function
description	61-63
ProMechsectiondataNonstructcogGet()	function
description	61-64
ProMechsectiondataNonstructmassGet()	function
description	61-65
ProMechsectiondataNonstructmomentGet()	function
description	61-65
ProMechsectiondataNonstructmomentSet()	function
description	61-65
ProMechsectiondataRectanglesectdataGet()	function
description	61-62
ProMechsectiondataRectanglesectdataSet()	function
description	61-62
ProMechsectiondataSectiontypeGet()	function
description	61-62
ProMechsectiondataSketchedsectdataGet()	function
description	61-62
ProMechsectiondataSketchedsectdataSet()	function
description	61-62
ProMechsectiondataSquaresectdataGet()	function
description	61-62
ProMechsectiondataSquaresectdataSet()	function
description	61-62
ProMechsectiondataWarpcoeffGet()	function
description	61-65
ProMechsectiondataWarpcoeffSet()	function
description	61-65
ProMechshellAdvanceddataGet()	function
description	61-83
ProMechshellAdvanceddataSet()	function
description	61-83
ProMechshelladvancedMaterialorientGet()	function
description	61-84
ProMechshelladvancedMaterialorientSet()	function
description	61-84
ProMechshelladvancedShellpropsGet()	function
description	61-84
ProMechshelladvancedShellpropsSet()	function
description	61-84
ProMechshellMaterialGet()	function
description	61-83
ProMechshellMaterialIdSet()	function
description	61-83
ProMechshellpairMaterialGet()	function
description	61-97
ProMechshellpairMaterialIdSet()	function
description	61-97
ProMechshellpairReferencesGet()	function
description	61-97
ProMechshellpairReferencesSet()	function
description	61-97
ProMechshellpropsDescriptionGet()	function
description	61-92
ProMechshellpropsDescriptionSet()	function
description	61-92
ProMechshellpropsHomogeneousdataGet()	function
description	61-92
ProMechshellpropsHomogeneousdataSet()	function
description	61-92
ProMechshellpropsLaminatelayupdataGet()	function
description	61-92
ProMechshellpropsLaminatelayupdataSet()	function
description	61-92
ProMechshellpropsLaminatestiffdataGet()	function
description	61-92
ProMechshellpropsLaminatestiffdataSet()	function
description	61-92
ProMechshellpropsTypeGet()	function
description	61-91
ProMechshellReferencesGet()	function
description	61-84
ProMechshellReferencesSet()	function
description	61-84
ProMechshellSimplifieddataGet()	function
description	61-83
ProMechshellSimplifieddataSet()	function
description	61-83
ProMechshellsimpleThicknessGet()	function
description	61-84
ProMechshellsimpleThicknessSet()	function
description	61-84
ProMechshellTypeGet()	function
description	61-83
ProMechshlpairrefsBottomreferencesGet()	function
description	61-98
ProMechshlpairrefsBottomreferencesSet()	function
description	61-98

function
 description 61-98
 ProMechshlpairrefsExtendAdjacentSurfacesGet()
 et() function
 description 61-98
 ProMechshlpairrefsExtendAdjacentSurfacesSet()
 et() function
 description 61-98
 ProMechshlpairrefsFree() function
 description 61-99
 ProMechshlpairrefsPlacementtypeGet()
 function
 description 61-98
 ProMechshlpairrefsPlacementtypeSet()
 function
 description 61-98
 ProMechshlpairrefsSelectedplacementGet()
 function
 description 61-98
 ProMechshlpairrefsSelectedplacementSet()
 function
 description 61-98
 ProMechshlpairrefsTopreferencesGet()
 function
 description 61-98
 ProMechshlpairrefsTopreferencesSet()
 function
 description 61-98
 ProMechshlpairrefsTypeGet() function
 description 61-97
 ProMechshlpairrefsTypeSet() function
 description 61-97
 ProMechshlprophomogeneousThicknessGet()
 function
 description 61-92
 ProMechshlprophomogeneousThicknessSet()
 function
 description 61-92
 ProMechshlproplaminatelayupLayersGet()
 function
 description 61-95
 ProMechshlproplaminatelayupLayersSet()
 function
 description 61-95
 ProMechshlproplaminatelayupTypeGet()
 function
 description 61-95
 ProMechshlproplaminatelayupTypeSet()
 function
 description 61-95
 ProMechshlproplaminatetestiffAppliedstressGet()
 function
 description 61-92
 ProMechshlproplaminatetestiffAppliedstressSet()
 function
 description 61-92

ProMechshlproplaminatetestiffBendingstiffness
 Get() function
 description 61-93
 ProMechshlproplaminatetestiffBendingstiffness
 Set() function
 description 61-93
 ProMechshlproplaminatetestiffCouplingstiffness
 Get() function
 description 61-93
 ProMechshlproplaminatetestiffCouplingstiffness
 Set() function
 description 61-93
 ProMechshlproplaminatetestiffExtensionalstiffness
 Get() function
 description 61-93
 ProMechshlproplaminatetestiffExtensionalstiffness
 Set() function
 description 61-93
 ProMechshlproplaminatetestiffIntertiaperunitarea
 Get() function
 description 61-94
 ProMechshlproplaminatetestiffMassperunitarea
 Get() function
 description 61-94
 ProMechshlproplaminatetestiffMassperunitarea
 Set() function
 description 61-94
 ProMechshlproplaminatetestiffTansverseshear
 Get() function
 description 61-94
 ProMechshlproplaminatetestiffTansverseshear
 Set() function
 description 61-94
 ProMechshlproplaminatetestiffThermalresforce
 Get() function
 description 61-94
 ProMechshlproplaminatetestiffThermalresforce
 Set() function
 description 61-94
 ProMechshlproplaminatetestiffThermalresmoment
 Get() function
 description 61-94
 ProMechshlproplaminatetestiffThermalresmoment
 Set() function
 description 61-94
 ProMechshlproplamlayuplayerMaterialGet()
 function
 description 61-95
 ProMechshlproplamlayuplayerMaterialIdSet()
 function
 description 61-95
 ProMechshlproplamlayuplayerNumberGet()
 function
 description 61-95
 ProMechshlproplamlayuplayerNumberSet()
 function

description 61-95
ProMechshlproplamlayuplayerOrientationGet()
() function
description 61-95
ProMechshlproplamlayuplayerOrientationSet()
() function
description 61-95
ProMechshlproplamlayuplayerShellpropsGet()
() function
description 61-96
ProMechshlproplamlayuplayerShellpropsSet()
() function
description 61-96
ProMechshlproplamlayuplayerThicknessGet()
() function
description 61-96
ProMechshlproplamlayuplayerThicknessSet()
() function
description 61-96
ProMechsimplegapdataAlloc() function
description 61-107
ProMechsimplegapdataAxialstiffnessGet()
() function
description 61-108
ProMechsimplegapdataAxialstiffnessSet()
() function
description 61-108
ProMechsimplegapdataClearanceGet()
() function
description 61-108
ProMechsimplegapdataClearanceSet()
() function
description 61-108
ProMechsimplegapdataDistributiontypeGet()
() function
description 61-107
ProMechsimplegapdataDistributiontypeSet()
() function
description 61-108
ProMechsimplegapdataTransversestiffnessGe
t() function
description 61-108
ProMechsimplegapdataTransversestiffnessSe
t() function
description 61-108
ProMechsimplegapdataYdirectionGet()
() function
description 61-107
ProMechsimplegapdataYdirectionSet()
() function
description 61-107
ProMechsimplemassdataAlloc() function
description 61-76
ProMechsimplemassdataMassGet() function
description 61-76
ProMechsimplemassdataMassSet() function

description 61-76
ProMechsketchedsctndataAlloc() function
description 61-65
ProMechsketchedsctndataFeatureidGet()
() function
description 61-65
ProMechsketchedsctndataFree() function
description 61-66
ProMechsketchedsctndataOrienttypeGet()
() function
description 61-65
ProMechsketchedsctndataOrienttypeSet()
() function
description 61-66
ProMechsketchedsctndataShearcenterGet()
() function
description 61-66
ProMechsketchedsctndataShearcenterSet()
() function
description 61-66
ProMechspringAdvanceddataGet() function
description 61-73
ProMechspringAdvanceddataSet() function
description 61-73
ProMechspringGrounddataGet() function
description 61-73
ProMechspringGrounddataSet() function
description 61-73
ProMechspringpropsDataGet() function
description 61-74
ProMechspringReferencesGet() function
description 61-72
ProMechspringReferencesSet() function
description 61-72
ProMechspringSimplifiedataGet() function
description 61-72
ProMechspringSimplifiedataSet() function
description 61-72
ProMechspringTypeGet() function
description 61-72
ProMechstiffmatrixEntry11Get() function
description 61-93
ProMechstiffmatrixEntry12Get() function
description 61-93
ProMechstiffmatrixEntry12Set() function
description 61-94
ProMechstiffmatrixEntry16Get() function
description 61-93
ProMechstiffmatrixEntry16Set() function
description 61-94
ProMechstiffmatrixEntry22Get() function
description 61-93
ProMechstiffmatrixEntry22Set() function
description 61-94
ProMechstiffmatrixEntry26Get() function
description 61-93

ProMechstiffmatrixEntry26Set() function description 61-94	ProMechtableentryIndependentvarGet() function description 61-14
ProMechstiffmatrixEntry66Get() function description 61-93	ProMechtableentryProarrayFree() function description 61-14
ProMechstiffmatrixEntry66Set() function description 61-94	ProMechtemperaturedataAlloc() function description 61-33
ProMechstresscalcddataCzGet() function description 61-93	ProMechtemperaturedataFree() function description 61-33
ProMechstresscalcddataCzSet() function description 61-93	ProMechtemperaturedataValueGet() function description 61-33
ProMechstresscalcddataMaterialGet() function description 61-93	ProMechtemperaturedataValueSet() function description 61-33
ProMechstresscalcddataMaterialIdSet() function description 61-93	ProMechthermalrescoeffEntry11Get() function description 61-95
ProMechstresscalcddataPlyorientationGet() function description 61-93	ProMechthermalrescoeffEntry11Set() function description 61-95
ProMechstresscalcddataPlyorientationSet() function description 61-93	ProMechthermalrescoeffEntry12Get() function description 61-95
ProMechstructtempdataFree() function description 61-34	ProMechthermalrescoeffEntry12Set() function description 61-95
ProMechstructtempdataReftempGet() function description 61-34	ProMechthermalrescoeffEntry22Get() function description 61-95
ProMechstructtempdataReftempSet() function description 61-34	ProMechthermalrescoeffEntry22Set() function description 61-95
ProMechstructtempdataValueGet() function description 61-34	ProMechthrresistinterfacedataConductivityG et() function description 61-106
ProMechstructtempdataValueSet() function description 61-34	ProMechthrresistinterfacedataConductivitySe t() function description 61-106
ProMechsymmetrydataAlloc() function description 61-55	ProMechtransverseshearEntry44Get() function description 61-94
ProMechsymmetrydataAxisGet() function description 61-55	ProMechtransverseshearEntry45Get() function description 61-94
ProMechsymmetrydataAxisSet() function description 61-55	ProMechtransverseshearEntry55Get() function description 61-94
ProMechsymmetrydataFree() function description 61-55	ProMechtrussdataAlloc() function description 61-60
ProMechsymmetrydataSide1Get() function description 61-55	ProMechtrussdataFree() function description 61-60
ProMechsymmetrydataSide1Set() function description 61-55	ProMechtrussdataSectionGet() function description 61-60
ProMechsymmetrydataSide2Get() function description 61-55	ProMechtrussdataSectionSet() function description 61-60
ProMechsymmetrydataSide2Set() function description 61-55	ProMechvalueValueGet() function description 61-18
ProMechsymmetrydataTypeGet() function description 61-55	
ProMechsymmetrydataTypeSet() function description 61-55	
ProMechtableentryFunctionvalueGet() function description 61-14	

ProMechvalueValueSet() function description 61-18	ProMechweldedgeMaterialidSet() function description 61-124
ProMechvalueVariationGet() function description 61-18	ProMechweldedgeProarrayFree() function description 61-124
ProMechvalueVariationSet() function description 61-18	ProMechweldedgeThicknessGet() function description 61-123
ProMechvariationExternalfielddataGet() function description 61-19	ProMechweldedgeThicknessSet() function description 61-123
ProMechvariationExternalfielddataSet function description 61-19	ProMechweldendAlloc() function description 61-125
ProMechvariationFunctionidGet() function description 61-19	ProMechweldEnddataGet function description 61-124
ProMechvariationFunctionidSet() function description 61-19	ProMechweldEnddataSet() function description 61-125
ProMechvariationInterpolationGet() function description 61-19	ProMechweldendExtendAdjacentSurfacesGet() function description 61-125
ProMechvariationInterpolationSet() function description 61-19	ProMechweldendExtendAdjacentSurfacesSet() function description 61-125
ProMechvariationTypeGet() function description 61-18	ProMechweldendFree() function description 61-125
ProMechvectoredvalueAlloc() function description 61-17	ProMechweldendSet() function description 61-124
ProMechvectoredvalueDirectiontypeGet() function description 61-17	ProMechweldendTypeGet function description 61-125
ProMechvectoredvalueDirectiontypeSet() function description 61-17	ProMechweldPerimeterdataGet() function description 61-123
ProMechvectoredvalueDirectionvectorGet() function description 61-17	ProMechweldPerimeterdataSet() function description 61-123
ProMechvectoredvalueDirectionvectorSet() function description 61-17	ProMechweldperimeterEdgesGet() function description 61-123
ProMechvectoredvalueFree() function description 61-18	ProMechweldperimeterEdgesSet() function description 61-123
ProMechvectoredvalueMagnitudeGet() function description 61-18	ProMechweldperimeterFree() function description 61-123, 61-124
ProMechvectoredvalueMagnitudeSet() function description 61-18	ProMechweldReferencesGet() function description 61-123
ProMechvectoredvaluePointsGet() function description 61-18	ProMechweldReferencesSet() function description 61-123
ProMechvectoredvaluePointsSet() function description 61-18	ProMechweldSpotdataGet() function description 61-124
ProMechweldedgeEdgeGet() function description 61-123	ProMechweldSpotdataSet() function description 61-124
ProMechweldedgeEdgeSet() function description 61-123	ProMechweldspotDiameterGet() function description 61-124
ProMechweldedgeFree() function description 61-124	ProMechweldspotDiameterSet() function description 61-124
ProMechweldedgeMaterialidGet() function description 61-124	ProMechweldspotFree() function description 61-124
	ProMechweldspotMaterialIdGet() function description 61-124
	ProMechweldspotMaterialIdSet() function description 61-124
	ProMechweldspotPntsSet() function description 61-124

ProMechweldspotPointsGet() function	promenu_push() function
description 61-124	equivalent I-17
ProMechweldTypeGet() function	promenu_push_command() function
description 61-123	equivalent I-17
ProMechydirectionAlloc() function	promenu_remove_item() function
description 61-11	equivalent I-17
ProMechydirectionCsysGet() function	promenu_reset_item() function
description 61-12	equivalent I-17
ProMechydirectionCsysSet() function	promenu_set_data_mode() function
description 61-12	equivalent I-17
ProMechydirectionFree() function	promenu_set_item() function
description 61-12	equivalent I-17
ProMechydirectionReferenceGet() function	promenu_set_item_location() function
description 61-12	equivalent I-17
ProMechydirectionReferenceSet() function	promenu_set_item_visible() function
description 61-12	equivalent I-17
ProMechydirectionTypeGet() function	promenu_set_mode() function
description 61-11	equivalent I-17
ProMechydirectionVectorGet() function	ProMenuAuxfileRegister() function
description 61-12	description 15-44
ProMechydirectionVectorSet() function	ProMenubarMenuAdd() function
description 61-12	used in a code example 15-18
promenu_action() function	ProMenubarmenuMenuAdd() function
equivalent I-16	description 15-15
promenu_create() function	used in a code example 15-18
equivalent I-16	ProMenubarmenuPushbuttonAdd() function
promenu_exit_action_up() function	syntax 15-8
equivalent I-17	used in a code example 15-18
promenu_exit_up() function	ProMenubarMenuRadiogrpValueGet()
equivalent I-16	function
promenu_expand() function	description 15-13
equivalent I-16	ProMenubarMenuRadiogrpValueSet()
promenu_get_pick() function	function
equivalent I-19	description 15-14
promenu_is_up() function	ProMenubuttonActionSet() function
equivalent I-17	description 15-45
promenu_load_action() function	using the final arguments 15-51
equivalent I-16	ProMenubuttonActivate() function
promenu_load_post_func() function	description 15-58
equivalent I-17	ProMenubuttonDeactivate() function
promenu_load_pre_func() function	description 15-58
equivalent I-17	ProMenubuttonGenactionSet() function 15-44
promenu_make() function	ProMenubuttonHighlight() function
equivalent I-16	description 15-58
promenu_make_compound() function	ProMenubuttonLocationSet() function
equivalent I-17	description 15-56
promenu_make_item_accessible() function	ProMenubuttonPostactionSet() function
equivalent I-17	description 15-53
promenu_make_item_inaccessible() function	ProMenubuttonPreactionSet() function
equivalent I-17	description 15-53
promenu_no_exit() function	used in a code example 15-54
equivalent I-17	ProMenubuttonUnhighlight() function
promenu_on_button() function	description 15-58
equivalent I-16	ProMenuCommandPush() function
promenu_pop() function	description 15-69, 15-73
equivalent I-17	ProMenuCreate() function

- description 15-49
- to create submenus 15-55
- ProMenuDatamodeSet() function
 - description 15-57
- ProMenuDelete() function
 - description 15-48
 - used with ProMenuProcess() 15-51
- ProMenuDeleteWithStatus() function
 - description 15-51
- ProMenuFileRegister() function
 - calling 15-45
 - description 15-45
 - used in a code example 15-54
- ProMenuFromStringsRegister() function
 - description 15-59
- ProMenuHold() function
 - description 15-48
- ProMenuModeSet() function
 - description 15-57
- ProMenuPop() function
 - description 15-59
- ProMenuProcess() function
 - description 15-50
 - returning a value from 15-51
- ProMenuPush() function
 - description 15-59
- ProMenuStringsSelect() function
 - description 15-59
- ProMenuVisibilityGet() function
 - description 15-58
- ProMessageClear() function
 - description 14-5
- ProMessageDisplay() function
 - description 14-4, 14-9, 19-5
 - used in a code example 15-55
- ProMessageDoubleRead() function
 - description 14-9
- ProMessageIntegerRead() function
 - description 14-9
- ProMessagePasswordRead() function
 - description 14-9
- ProMessageStringRead() function
 - description 14-9
 - used in a code example 15-55
- ProMessageToBuffer() function
 - description 14-8
- promfg_get_nc_type() function
 - equivalent I-27
- promfg_get_tool_ids() function
 - equivalent I-27
- promfg_get_tool_parameters() function
 - equivalent I-27
- promfg_set_tool_parameters() function
 - equivalent I-27
- ProMfgAssemGet() function
 - description 53-2
- ProMfgCreate() function
 - description 53-2
- ProMfgddbDataAlloc() function
 - description 54-2
- ProMfgddbDataDbnameAdd() function
 - description 54-3
- ProMfgddbDataSearchoptAdd() function
 - description 54-3
- ProMfgddbNameCreate() function
 - definition 54-3
- ProMfgddbRegister() function
 - description 54-3
- ProMfgddbSearchoptAllowedValueAdd()
 - function
 - description 54-3
- ProMfgddbSearchoptApplicDataAdd() function
 - description 54-3
- ProMfgddbSearchoptCreate() function
 - definition 54-3
- ProMfgFeatureOwnerGet() function
 - description 53-6
- ProMfgoperClPostAction callback 65-7
- ProMfgoperToolpathDisplay() function
 - description 53-45
- ProMfgprocitemAnnotationGet() function
 - description 57-8, 57-10
- ProMfgprocitemCreate() function
 - description 57-6, 57-7
- ProMfgprocitemDefaultfixturesetupstepGet()
 - function
 - description 57-9
- ProMfgprocitemDelete() function
 - description 57-9
- ProMfgprocitemElemtreeFree() function
 - description 57-8
- ProMfgprocitemElemtreeGet() function
 - description 57-8
- ProMfgprocitemFeatureGet() function
 - description 57-8
- ProMfgprocitemFromTemplateCreate()
 - function
 - description 57-7
- ProMfgprocitemHolesetdepthGet() function
 - description 57-8
- ProMfgprocitemHolesetdepthSet() function
 - description 57-9
- ProMfgprocitemHolesetdepththtypeGet()
 - function
 - description 57-8
- ProMfgprocitemNextitemGet() function
 - description 57-8
- ProMfgprocitemPreviousitemGet() function
 - description 57-8
- ProMfgprocitemPropertyGet() function
 - description 57-11
- ProMfgprocitemPropertySet() function

description 57-12	description 12-8
ProMfgprocitemRedefine() function	ProModelcheckRegisterFunction() function
description 57-9	description 12-10
ProMfgprocitemReorder() function	ProModelitem object
description 57-9	declaration 1-51
ProMfgprocitemReorderlimitsGet() function	description 2-8
description 57-8	ProModelitemByNameInit() function
ProMfgprocitemSubtypeGet() function	description 2-9
description 57-5	ProModelitemDefaultnameGet() function
ProMfgprocitemTemplateparamGet() function	description 2-10
description 57-10	ProModelitemHide() function
ProMfgProctableDisable() function	description 2-10
description 57-3	ProModelitemInit() function
ProMfgproctableDisplayUpdate() function	description 2-10
description 57-4	ProCurve to a ProGeomitem 5-13
ProMfgProctableEnable() function	ProPoint to a ProGeomitem 5-15
description 57-3	ProQuilt to a ProGeomitem 5-12
ProMfgproctableExtractionPostAction	ProSurface to a ProGeomitem 5-11
function	used in visits 5-10
description 57-13	used with parameters 7-4
ProMfgproctableFeaturesCreate() function	ProModelitemIsHidden() function
description 57-12	description 2-11
ProMfgproctableFeaturesDelete() function	ProModelitemMdlGet() function
description 57-12	description 2-10
ProMfgproctableItemAccessFunction function	ProModelitemNameCanChange() function
description 57-3	description 2-10
ProMfgproctableItemActionFunction function	ProModelitemNameGet() function
description 57-3	description 2-10, 22-5
ProMfgproctableMenuAdd() function	used with ProGeomitem 5-3
description 57-3	ProModelitemNameSet() function
ProMfgproctablePushbuttonAdd() function	description 2-10
description 57-3	used with ProGeomitem 5-3
ProMfgproctableSelecteditemsGet() function	ProModelitemToFamtableItem() function
description 57-4	description 9-7
ProMfgproctableSelecteditemsSet() function	ProModelitemToRelset() function
description 57-4	description 6-3
ProMfgProctableSynchronize() function	ProModelitemUnhide() function
description 57-13	description 2-10
ProMfgProctableVisit() function	ProModelitemUsernameDelete() function
description 57-6	description 2-10
ProMfgProctableWrite() function	ProModeNameCurrentGet() function
description 57-13	description 2-2
ProMfgSolidGet() function	ProMouseBoxInput() function 18-20
description 53-6	ProMousePickGet() function
ProMfgToolVisit() function	description 18-20
description 53-8	used in a code example 18-17
ProMfgTypeGet() function	ProMouseTrack() function
description 53-2	description 18-20
ProMode object	used in a code example 18-22
description 2-2	promsg_clear() function
ProModeCurrentGet() function	equivalent I-17
description 2-2	promsg_getdouble() function
used in a code example 18-9	equivalent I-18
ProModelcheckCheckFunction() function	promsg_getint() function
description 12-11	equivalent I-18
ProModelcheckExecute() function	promsg_getpasswd() function

equivalent I-18	description 22-54
promsg_getstring() function	ProNoteAttachNormtanleaderGet() function
equivalent I-18	description 22-51
promsg_print() function	ProNoteAttachOnitemGet() function
equivalent I-17	description 22-54
promsg_sprint() function	ProNoteAttachOnitemSet() function
equivalent I-18	description 22-56
ProNavigatorpaneBrowserAdd() function	ProNoteAttachPlaneGet() function
description 15-60	description 22-54
ProNavigatorpaneBrowsericonSet() function	ProNoteAttachPlaneSet() function
description 15-61	description 22-56
ProNavigatorpaneBrowserURLSet() function	ProNoteAttachRelease() function
description 15-61	description 22-55
ProNcseqClIPostAction callback 65-7	ProNoteDelete() function
ProNcseqCutTimeGet() function	description 22-50
description 53-45	ProNoteElbowlengthGet() function
ProNcseqElemHolesetAdd() function	description 22-51
description 53-40	ProNoteElbowlengthSet() function
ProNcseqElemHolesetAxisAdd() function	description 22-51
description 53-40	ProNoteLeaderstyleGet() function
ProNcseqElemHolesetDepthBySet() function	description 22-51
description 53-40	ProNoteLeaderstyleSet() function
ProNcseqElemHolesetDepthSet() function	description 22-51
description 53-40	ProNoteLineEnvelopeGet() function
ProNcseqElemHolesetDepthTypeSet()	description 22-51
function	ProNoteOwnerGet() function
description 53-40	description 22-51
ProNcseqElemHolesetDirectionSet() function	ProNotePlacementGet() function
description 53-40	description 22-54
ProNcseqElemHolesetEndSet() function	ProNotePlacementSet() function
description 53-40	description 22-55, 22-56
ProNcseqElemHolesetStartSet() function	ProNoteTextStyleGet() function
description 53-40	description 22-52
ProNcseqElemMillsurfSet() function	ProNoteTextStyleSet() function
description 53-39	description 22-52
ProNcseqElemSurfaceAdd() function	ProNoteURLGet() function
description 53-39	description 22-51
ProNcseqElemSurfaceflipSet() function	ProNoteURLSet() function
description 53-39	description 22-51
ProNcseqNumGet() function	ProNotificationSet() function
description 53-45	description 65-2
ProNcseqRemovedVolGet() function	ProNotificationUnset() function
description 53-45	description 65-2
ProNcseqToolpathDisplay() function	ProObjectwindowCreate() function
description 53-45	description 18-4
ProNoteAttachAlloc() function	used in a code example 3-4
description 22-55	ProOptionOrderedVerify() function
ProNoteAttachFreeGet() function	description 12-3
description 22-54	ProOutputAssemblyConfigurationIsSupporte
ProNoteAttachFreeSet() function	d function
description 22-55	description 26-25
ProNoteAttachLeaderAdd() function	ProOutputFileWrite() function
description 22-56	description 26-2, 51-6
ProNoteAttachLeaderRemove() function	for cable parameters 58-14
description 22-56	for cabling 26-3, 58-10
ProNoteAttachLeadersGet() function	used with cabling 58-9

ProOutputInclusionAlloc function
 description 26-25
 ProOutputInclusionFacetparamsSet()
 function
 description 26-26
 ProOutputInclusionFlagsSet function
 description 26-25
 ProOutputInclusionFree function
 description 26-27
 ProOutputLayerOptionsAlloc function
 description 26-27
 ProOutputLayerOptionsAutoidSet function
 description 26-27
 ProOutputLayerOptionsSetupfileSet function
 description 26-27, 26-28
 ProParameter object
 description 7-2
 ProParameter structure
 declaration 7-2
 ProParameterCreate() function
 description 7-5
 ProParameterCreatePostAction callback 65-5
 ProParameterCreatePostAction() function
 description 7-8
 ProParameterCreatePreAction callback 65-5
 ProParameterCreatePreAction() function
 description 7-7
 ProParameterDelete() function
 description 7-5
 ProParameterDeletePostAction callback 65-6
 ProParameterDeletePostAction() function
 description 7-8
 ProParameterDeletePreAction callback 65-6
 ProParameterDeletePreAction() function
 description 7-7
 ProParameterDescriptionGet() function
 description 7-7
 ProParameterDescriptionSet() function
 description 7-7
 ProParameterDesignationAdd() function
 description 7-10
 ProParameterDesignationRemove() function
 description 7-10
 ProParameterDesignationVerify() function
 description 7-10
 ProParameterDriverGet() function
 description 7-13
 ProParameterDrivingFunctionGet() function
 description 7-13
 ProParameterDrivingFunctionSet() function
 description 7-13
 ProParameterDrivingparamGet() function
 description 7-13
 ProParameterDrivingparamSet() function
 description 7-13
 ProParameterInit() function
 description 7-4
 ProParameterIsEnumerated() function
 description 7-10, 22-11
 ProParameterIsModified() function
 description 7-5
 ProParameterModifyPostAction callback 65-5
 ProParameterModifyPostAction() function
 description 7-8
 ProParameterModifyPreAction callback 65-5
 ProParameterModifyPreAction() function
 description 7-7
 ProParameterRangeGet() function
 description 7-11, 22-11
 ProParameterReorder() function
 description 7-6
 ProParameterScaledvalueGet() function
 description 7-7
 ProParameterScaledvalueSet() function
 description 7-7
 ProParameterSelect() function
 description 7-5
 used with family table items 7-6, 9-7
 ProParameterTableExport() function
 description 7-6
 ProParameterTablesetGet() function
 description 7-11
 ProParameterToFamtableItem() function
 description 7-6, 9-7
 ProParameterUnitsGet() function
 description 7-7
 ProParameterValueGet() function
 description 7-4
 ProParameterValueReset() function
 description 7-5
 ProParameterValueSet() function
 description 7-5
 ProParameterVisit() function
 description 7-6
 ProParameterWithUnitsCreate() function
 description 7-7
 ProParamfrom enum
 declaration 7-2
 ProParamowner structure
 declaration 7-2
 ProParamtableentryNameGet() function
 description 7-12
 ProParamtableentryProarrayFree() function
 description 7-12
 ProParamtableentryRangeGet() function
 description 7-12
 ProParamtableentryValueGet() function
 description 7-12
 ProParamtablesetApply() function
 description 7-12
 ProParamtablesetEntriesGet() function
 description 7-12

ProParamtablesetFree() function
 description 7-12
 ProParamtablesetLabelGet() function
 description 7-12
 ProParamtablesetProarrayFree() function
 description 7-12
 ProParamtablesetTablepathGet() function
 description 7-12
 ProParamvalue structure
 declaration 7-3
 ProParamvalueSet() function
 description 7-3
 ProParamvalueType structure
 declaration 7-3
 ProParamvalueTypeGet() function
 description 7-3
 ProParamvalueValue structure
 declaration 7-3
 ProParamvalueValueGet() function
 description 7-4
 ProPartDensityGet() function
 description 3-19
 ProPartDensitySet() function
 description 3-19
 ProPartMaterialsGet() function
 description 3-22
 ProPartSimpRepRetrieve() function
 description 49-6
 ProPartTessellate()
 function description 5-18
 ProPartTessellationFree()function
 description 5-18
 ProPartToProIntfData() function
 description 27-19
 ProPathCreate() function
 description 12-6
 ProPattern object
 description 39-2
 ProPatternClass object
 description 39-2
 ProPatternCreate() function
 description 39-23
 ProPatternDelete() function
 description 4-14
 ProPatternElemntreeCreate() function
 description 39-22
 ProPatternLeaderGet() function
 description 4-14
 ProPatternMembersGet() function
 description 39-23
 ProPatternMemberVisit() function
 description 39-23
 ProPDFExport() function
 description 26-45
 ProPDFOptionsAlloc() function
 description 26-37
 ProPDFOptionsBoolpropertySet() function
 description 26-43
 ProPDFOptionsDoublepropertySet() function
 description 26-45
 ProPDFOptionsFree() function
 description 26-46
 ProPDFOptionsIntpropertySet() function
 description 26-37
 ProPDFOptionsStringpropertySet() function
 description 26-44
 Properties
 Loads and Constraints 61-15
 material 3-21
 of external object display 66-8
 surface 18-23
 text style 22-52
 ProPntTrfEval() function
 description 8-5
 used with assemblies 45-11
 ProPoint object
 declaration 1-50
 ProPoint3d typedef
 declaration 8-4
 ProPoint3dIntoleranceFind()function
 description 5-20
 ProPoint3dOnsurfaceFind()function
 description 5-20
 ProPointCoordGet() function
 description 5-29
 ProPointIdGet() function
 description 5-15
 ProPointInit() function
 description 5-15
 ProPopupMenuAccessFunction() function
 description 15-34
 ProPopupMenuButtonAdd() function
 description 15-35
 ProPopupMenuCascadebuttonAdd() function
 description 15-35
 ProPopupMenuCreatePostAction callback
 65-6
 ProPopupMenuCreatePostAction() function
 description 15-33
 ProPopupMenuDestroyPreAction callback
 65-6
 ProPopupMenuDestroyPreAction() function
 description 15-33
 ProPopupMenuIdGet() function
 description 15-33
 ProPopupMenuNameGet() function
 description 15-34
 ProPostfixIdToSolid() function
 description 3-18
 ProPrintExecute() function
 description 26-15
 ProPrintMdlOptionsGet() function

- description 26-14
- ProPrintPCFOptionsGet() function
 - description 26-15
- ProPrintPlacementOptionsGet() function
 - description 26-15
- ProPrintPrinterOptionsGet() function
 - description 26-13
- ProProcstep structure
 - declaration 56-2
- ProProcstepActiveGet() function
 - description 56-2
- ProProcstepActiveSet() function
 - description 56-2
- ProProcstepInit() function
 - description 56-2
- ProProcstepNumberGet() function
 - description 56-3
- ProProcstepVisit() function
 - description 56-2
- ProProductviewexportoptsAlloc() function
 - description 26-13, 26-29
- ProProductviewexportoptsFormatSet() function
 - description 26-13, 26-29
- ProProductviewexportoptsFree() function
 - description 26-13, 26-29
- ProProductviewFormattedExport() function
 - description 26-13, 26-29
- proptntbl_get_active_table() function
 - description 4-19
- proptntbl_get_all_tbl_names() function
 - description 4-19
- proptntbl_get_inst_dim_value() function
 - description 4-19
- proptntbl_get_inst_indices() function
 - description 4-19
- proptntbl_get_lead_pat_dims() function
 - description 4-19
- proptntbl_set_active_table() function
 - description 4-19
- proptntbl_set_dimval_driven() function
 - description 4-19
- proptntbl_set_inst_dim_value() function
 - description 4-19
- ProQuickprintExecute() function
 - description 26-21
- ProQuickprintoptionsAlloc() function
 - description 26-16
- ProQuickprintoptionsFree() function
 - description 26-16
- ProQuickprintoptionsLayouttypeSet() function
 - description 26-16
- ProQuickprintoptionsOrientationSet() function
 - description 26-17

- ProQuickprintoptionsPrintFlatToScreenAnno
 - tsSet() function
 - description 26-21
- ProQuickprintoptionsProjectionsSet() function
 - description 26-19
- ProQuickprintoptionsSizeSet() function
 - description 26-17
- ProQuickprintoptionsTemplateSet() function
 - description 26-20
- ProQuickprintoptionsThreeviewlayoutSet() function
 - description 26-18
- ProQuickprintoptionsViewAdd() function
 - description 26-17
- ProQuiltdataAlloc() function
 - description 27-12
- ProQuiltdataFree() function
 - description 5-28
- ProQuiltdataGet() function
 - description 5-28
- ProQuiltdataInit() function
 - description 27-12
- ProQuiltdataMemoryFree() function
 - description 5-28
- ProQuiltdataSurfArrayGet() function
 - description 5-28
- ProQuiltdataSurfArraySet() function
 - description 5-28, 27-12
- ProQuiltIdGet() function
 - description 5-12
- ProQuiltInit() function
 - description 5-12
- ProQuiltSurfaceVisit() function
 - description 5-12
 - for geometry 5-28
- ProQuiltVolumeEval() function
 - description 5-28
- ProRasterFileWrite() function
 - description 26-28
- ProReferenceAlloc()function
 - description 30-5
- ProReferencearrayFree()function
 - description 30-5
- ProReferencearrayToSelections() function
 - description 30-4
- ProReferenceAsmcomppathGet()function
 - description 30-4
- ProReferenceDiagnosticsCollect() function
 - description 29-18
- ProReferenceFree()function
 - description 30-5
- ProReferenceIdGet()function
 - description 30-4
- ProReferenceIsLocalcopy() function
 - description 30-3

ProReferenceOriginalidGet()function description 30-4	ProRmdtMBaseSelectPreAction callback 65-7
ProReferenceOriginalownerGet()function description 30-4	ProRoundIsAutoRoundMember() function description 36-21
ProReferenceOriginalownernameGet()function n description 30-4	ProRuleEval() function description 49-16
ProReferenceOriginaltypeGet()function description 30-4	ProRuleInitDist() function description 49-17
ProReferenceOwnerGet()function description 30-4	ProRuleInitExpr() function description 49-16
ProReferenceOwnernameGet()function description 30-4	ProRuleInitName() function description 49-16
ProReferenceParamsGet()function description 30-4, 30-5	ProRuleInitRep() function description 49-18
ProReferencePointGet()function description 30-4, 30-5	ProRuleInitSize() function description 49-17
ProReferenceSet()function description 30-5	ProRuleInitZone() function description 49-17
ProReferenceStatusGet() function description 30-3	ProSecdimCreate() function description 40-13
ProReferenceToSelection()function description 30-4	ProSecdimDelete() function 40-13
ProReferenceTypeGet function description 30-3	ProSecdimDiamClear function definition 40-15
ProRelationArgscheckFunction() function description 6-5	ProSecdimDiamInquire function definition 40-15
ProRelationEval() function description 6-4	ProSecdimDiamSet function definition 40-15
ProRelationFunctionRegister() function description 6-4	ProSecdimIsLocked() function description 40-16
ProRelationReadFunction() function description 6-5	ProSecdimLockSet() function description 40-16
ProRelationWriteFunction() function description 6-5	ProSecdimStrengthen() function description 40-15
ProRelset object description 6-2	ProSecdimValueGet() function description 40-9
ProRelsetConstraintsGet() function description 7-12	ProSecdimValueSet() function description 40-9
ProRelsetCreate() function description 6-3	ProSecerrorAdd() function description 40-25
ProRelsetDelete() function description 6-3	ProSecerrorAlloc() function description 40-25
ProRelsetRegenerate() function description 6-3	ProSecerrorCount() function description 40-25 used in a code example 40-25
ProRelsetRelationsGet() function description 6-4	ProSecerrorFree() function used in a code example 40-26
ProRelsetRelationsSet() function description 6-4	ProSecerrorItemGet() function description 40-25
ProRelsetToModelitem() function description 6-3	ProSecerrorMsgGet() function description 40-25 used in a code example 40-26
ProRmdtBoundingBoxPreAction callback 65-7	ProSection2DAlloc() function description 40-3 used in a code example 40-4
ProRmdtCavLayoutPreAction callback 65-7	ProSection2dAlloc() function description 41-4
ProRmdtCreateImmPreAction callback 65-7	ProSectionAutodim() function
ProRmdtCreateWpPreAction callback 65-7	
ProRmdtMaterialPreAction callback 65-7	

- description 40-9
- ProSectionConstraintCreate() function
 - description 40-6
- ProSectionConstraintDelete() function
 - description 40-6
- ProSectionConstraintDeny() function
 - description 40-5
- ProSectionConstraintsGet() function
 - description 40-5
- ProSectionConstraintsIdsGet() function
 - description 40-5
- ProSectionEntityAdd() function
 - description 40-9
 - used in a code example 40-10
- ProSectionEntityDelete() function 40-10
 - description 40-9
- ProSectionEntityFromProjection() function
 - description 41-8
- ProSectionEntityIsProjection() function
 - description 41-8
- ProSectionEntityReferenceGet() function
 - description 41-9
- ProSectionEntityReplace() function
 - description 40-9
- ProSectionEntityUseEdge() function
 - description 41-9
- ProSectionEntityUseEdgeChain() function
 - description 41-9
- ProSectionEntityUseEdgeLoop() function
 - description 41-9
- ProSectionEpsilonGet() function
 - description 40-8
- ProSectionEpsilonSet() function
 - description 40-8
- ProSectionFree() function
 - description 40-4
- ProSectionIsActive() function
 - description 2-2
- ProSectionLocationGet() function
 - description 8-12, 41-7
- ProSectionNameGet() function 40-3
- ProSectionNameSet() function
 - description 40-3
 - used in a code example 40-4
- ProSectionRegenerate() function
 - description 40-7
- ProSectionShapeGet function
 - description 41-5
- ProSectionSolve() function
 - description 40-7
- ProSectionSolveRigid() function
 - description 40-8
- ProSelbox object
 - description 66-11
- ProSelbufferClear() function
 - description 19-11

- ProSelbufferCollectionAdd() function
 - description 19-12, 20-7
- ProSelbufferSelectionAdd() function
 - description 19-12
- ProSelbufferSelectionRemove() function
 - description 19-12
- ProSelbufferSelectionsGet() function
 - description 19-11, 19-12
- ProSeldataAlloc() function
 - description 66-10
- ProSeldataSelboxesGet() function
 - description 66-10
- ProSeldataSelboxesSet() function
 - description 66-10
- ProSelect() function
 - arguments 19-7
 - description 19-4, 19-5
 - guidelines I-15
 - selecting a datum point 5-31
- ProSelection 19-2
 - compared to Select3d I-15
 - information contained in 19-3
 - object 19-2
- ProSelectionAlloc() function I-13
 - description 19-5
- ProSelectionarrayToReferences()function
 - description 30-5
- ProSelectionAsmcompmpathGet() function 19-2
- ProSelectionCollectionGet() function
 - description 19-11, 20-6
- ProSelectionCopy() function
 - description 19-5
- ProSelectionDepthGet() function
 - description 5-30
- ProSelectionDisplay() function
 - description 19-10
- ProSelectionDwgtblcellGet() function
 - description 51-92
- ProSelectionFree() function
 - description 19-5
- ProSelectionHighlight() function
 - description 19-4, 19-9
 - guidelines I-15
- ProSelectionModelitemGet
 - to return gtol handle 23-2
- ProSelectionModelitemGet() function 19-2
 - used with family table items 9-7
- ProSelectionPoint3dGet() function 19-2
- ProSelectionSet() function
 - description 19-5, I-13
- ProSelectionToReference()function
 - description 30-4
- ProSelectionUnhighlight() function
 - and ProSelection 19-4
 - description 19-10
- ProSelectionUvParamSet() function

- description 19-2, I-13
- ProSelectionVerify() function
 - description 19-5, I-13
- ProSelectionViewGet() function 19-2
 - description 51-28
- ProServerActivate() function
 - description 25-5
- ProServerAliasedURLToModelName()
 - function
 - description 25-26
- ProServerAliaseURLToURL() function
 - description 25-26
- ProServerAliasGet() function
 - description 25-5
- ProServercheckinoptsAlloc() function
 - description 25-13
- ProServercheckinoptsAutoresolveSet()
 - function
 - description 25-14
- ProServerCheckinoptsBaselineSet() function
 - description 25-14
- ProServerCheckinoptsDeflocationSet()
 - function
 - description 25-13
- ProServercheckinoptsFree() function
 - description 25-15
- ProServerCheckinoptsKeepcheckedoutSet()
 - function
 - description 25-14
- ProServerCheckinoptsLocationSet() function
 - description 25-13
- ProServercheckinoptsUploadonlySet()
 - function
 - description 25-15
- ProServercheckoutoptsAlloc() function
 - description 25-17
- ProServercheckoutoptsDependencySet()
 - function
 - description 25-17
- ProServercheckoutoptsDownloadSet()
 - function
 - description 25-18
- ProServercheckoutoptsFree() function
 - description 25-18
- ProServercheckoutoptsIncludeinstancesSet()
 - function
 - description 25-17
- ProServercheckoutoptsReadOnlySet() function
 - description 25-18
- ProServercheckoutoptsVersionSet() function
 - description 25-18
- ProServerClassGet() function
 - description 25-4
- ProServerconflictsDescriptionGet() function
 - description 25-25
- ProServerconflictsFree() function

- description 25-26
- ProServerContextGet() function
 - description 25-5
- ProServerContextsCollect() function
 - description 25-4
- ProServerLocationGet() function
 - description 25-5
- ProServerlockConflictArrayFree() function
 - description 25-24
- ProServerlockConflictinfoGet() function
 - description 25-23
- ProServerlockStatusinfoGet() function
 - description 25-24
- ProServerModelNameToAliasedURL()
 - function
 - description 25-26
- ProServerMultiobjectsCheckout() function
 - description 25-17
- ProServerObjectIsLocked() function
 - description 25-24
- ProServerObjectsAreLocked() function
 - description 25-24
- ProServerObjectsCheckin() function
 - description 25-13
- ProServerObjectsCheckout() function
 - description 25-16
- ProServerObjectsLock() function
 - description 25-23
- ProServerObjectsRemove() function
 - description 25-25
- ProServerObjectsUndocheckout() function
 - description 25-19
- ProServerObjectsUnlock() functions
 - description 25-24
- ProServerObjectsUpload() function
 - description 25-12
- ProServerRegister() function
 - description 25-5
- ProServersCollect() function
 - description 25-5
- ProServerSynchronizationstateGet() function
 - description 25-12
- ProServerSynchronize() function
 - description 25-12
- ProServerTargetSet() function
 - description 25-11
- ProServerUnregister() function
 - description 25-5
- ProServerVersionGet() function
 - description 25-4
- ProServerWorkspaceCreate() function
 - description 25-6, 25-7
- ProServerworkspacedataAlloc() function
 - description 25-6
- ProServerworkspacedataContextGet()
 - function

description 25-7	ProShrinkwrapoptionsIgnorequiltsSet
ProServerworkspacedataFree() function	function
description 25-7	description 26-33
ProServerworkspacedataNameGet() function	ProShrinkwrapoptionsIgnoreskeletonSet
description 25-6	function
ProServerworkspacedataProarrayFree()	description 26-33
function	ProShrinkwrapoptionsIgnoresmallsurfsSet
description 25-7	function
ProServerWorkspaceGet() function	description 26-33
description 25-7	ProSimprep object
ProServerWorkspacesCollect() function	description 49-2
description 25-4	ProSimprepActionInit() function
ProServerWorkspaceSet() function	description 49-14
description 25-7	used in a code example 49-8
ProSessionMdlList() function	ProSimprepActivate() function
description 2-6	description 49-4
ProSetdatumtagAttachmentGet() function	used in a code example 49-9
description 22-57	ProSimprepActiveGet() function
ProSetdatumtagAttachmentSet() function	description 49-4
description 22-57	ProSimprepCreate() function
ProSetdatumtagCreate() function	used in a code example 49-8
description 22-57	ProSimprepdata structure
ProSetdatumtagPlaneGet() function	description 49-2
description 22-58	ProSimprepdataAlloc() function
ProSetdatumtagPlaneSet() function	used in a code example 49-8
description 22-58	ProSimprepdataDefltGet() function
ProSetdatumtagReferenceGet() function	description 49-9
description 22-58	ProSimprepdataDefltSet() function
ProSetdatumtagTextstyleGet() function	description 49-14
description 22-58	ProSimprepdataFree() function
ProSetdatumtagTextstyleSet() function	description 49-9
description 22-58	used in a code example 49-9
ProShrinkwrapoptionsAdditionalcomponents	ProSimprepdataGet() function
Set function	description 49-9
description 26-34	ProSimprepdataitemAdd() function
ProShrinkwrapoptionsAdditional SurfacesSet	description 49-14
function	used in a code example 49-8
description 26-33	ProSimprepdataitemDelete() function
ProShrinkwrapoptionsAlloc function	description 49-14
description 26-32	ProSimprepdataitemInit() function
ProShrinkwrapoptionsAssignmasspropsSet	description 49-14
function	used in a code example 49-8
description 26-33	ProSimprepdataitemsVisit() function
ProShrinkwrapoptionsAutoholefillingSet	description 49-10
function	ProSimprepdataNameGet() function
description 26-33	description 49-9
ProShrinkwrapoptionsDatumrefsSet function	ProSimprepdataNameSet() function
description 26-33	description 49-14
ProShrinkwrapoptionsFacetedformatSet	ProSimprepdataSet() function
function	description 49-14
description 26-34	used in a code example 49-8
ProShrinkwrapoptionsFramesFileSet	ProSimprepdataTmpvalGet() function
function	description 49-9
description 26-34	ProSimprepdataTmpvalSet() function
ProShrinkwrapoptionsFree function	description 49-14
description 26-32, 26-33	ProSimprepDelete() function

- description 49-6
- ProSimpRepInit() function
 - description 49-4
- ProSimpRepIsDefault() function
 - description 49-5
- ProSimpRepItem object
 - description 49-2
- ProSimpRepSelect() function
 - description 49-4
- ProSimpRepSubstitutionNameGet() function
 - description 49-10
- ProSingleAnimationClear() function
 - description 21-4
- ProSingleAnimationInit() function
 - description 21-4
- ProSingleAnimationPlay() function
 - description 21-4
- ProSldsSurfaceVolumesFind() function
 - description 5-26
- ProSmtBendEdgeChildGet() function
 - description 52-4
- ProSmtBendEdgeParentGet() function
 - description 52-4
- ProSmtBendSrfChildGet() function
 - description 52-4
- ProSmtBendSrfParentGet() function
 - description 52-3
- ProSmtDevldimDrivenSet() function
 - description 52-9
- ProSmtDevldimIsDriven() function
 - description 52-9
- ProSmtEdgeContourGet() function
 - description 52-3
- ProSmtFeatureBendTableApply() function
 - description 52-9
- ProSmtFeatureBendTableRemove() function
 - description 52-9
- ProSmtFeatureDevldimsGet() feature
 - defined 52-9
- ProSmtMdlIsFlatStateInstance() function
 - description 52-4
- ProSmtOppEdgeGet() function
 - description 52-3
- ProSmtOppSurfGet() function
 - description 52-3
- ProSmtPartBendTableApply() function
 - description 52-9
- ProSmtPartBendTableRemove() function
 - defined 52-9
- ProSmtPartThicknessGet() function
 - description 52-2
- ProSmtSurfaceTypeGet() function
 - description 52-3
- ProSnapshotAllowedInDrawingGet() function
 - description 48-9
- ProSnapshotAllowedInDrawingSet() function

- description 48-9
- ProSnapshotApply() function
 - description 48-5
- ProSnapshotConstraintAdd() function
 - description 48-8
- ProSnapshotConstraintDelete() function
 - description 48-8
- ProSnapshotConstraintEvaluate() function
 - description 48-8
- ProSnapshotConstraintsGet() function
 - description 48-8
- ProSnapshotConstraintUpdate() function
 - description 48-8
- ProSnapshotCreate() function
 - description 48-5
- ProSnapshotDelete() function
 - description 48-5
- ProSnapshotRename() function
 - description 48-5
- ProSnapshotTrfsGet() function
 - description 48-9
- ProSnapshotTrfsUpdate() function
 - description 48-9
- ProSnapshotUpdate() function
 - description 48-5
- ProSolid object
 - declaration 1-50
- ProSolidAccuracyGet() function
 - description 3-10
- ProSolidAccuracySet() function
 - description 3-10
- ProSolidAnalysisVisit function
 - defined 67-28
- ProSolidAnnotationElementsVisit() function
 - description 22-6
- ProSolidAxisVisit() function
 - description 5-10
 - overview 3-3
- ProSolidCreate() function
 - description 3-2
- ProSolidCsysVisit() function
 - description 5-10
 - fundamentals 3-3
- ProSolidDefaultTextHeightGet() function
 - description 22-53
- ProSolidDimensionVisit() function
 - description 22-23
- ProSolidDispCompVisit()
 - function description 2-20
- ProSolidDisplay() function
 - description 3-4
 - display lists 18-20
 - used in a code example 3-4
 - used with family table instances 9-5
- ProSolidDtlSymDefEntityVisit() function
 - description 22-62

ProSolidDtlSymdefNoteVisit() function	ProSolidQuiltVisit() function
description 22-62	description 5-12
ProSolidDtlSymdefRetrieve() function	fundamentals 3-3
description 22-60, 22-62	ProSolidRayIntersectionCompute() function
ProSolidDtlSymdefVisit() function	and ProSelection 19-4
description 22-62	description 5-30
ProSolidDtlSyminstVisit() function	ProSolidRegenerate() function
description 22-62	description 3-5
ProSolidExpldstateVisit() function	ProSolidRegeneratePostAction callback 65-6
description 45-14	ProSolidRegeneratePreAction callback 65-6
ProSolidFailedFeatsList() function	ProSolidRegenerationstatusGet() function
description 3-3	description 3-7
ProSolidFailedfeaturesList() function	ProSolidRelsetVisit() function
description 3-3	description 6-3
ProSolidFamtableCheck	ProSolidRetrievalErrorsGet() function
function definition 9-2	description 2-7
ProSolidFeatstatusflagsGet() function	ProSolidShrinkwrapCreate function
description 3-3	description 26-32
ProSolidFeatstatusGet() function	ProSolidSimprepVisit() function
description 3-3, 4-4	description 49-3
ProSolidFeatstatusSet() function	ProSolidSurfaceVisit() function
description 3-3	description 5-7
ProSolidFeatstatusWithoptionsSet() function	fundamentals 3-2
description 3-3	ProSolidSurffinishVisit() function
ProSolidFeatureVisit() function	description 22-59
defined 67-28	ProSolidTolclassLoad() function
ProSolidFeatVisit() function	description 22-35
description 1-60, 22-5, 53-2	ProSolidToleranceGet() function
explanation of feature visits 5-5	description 22-33
fundamentals 3-2	ProSolidToleranceSet() function
to visit workcells 53-6	description 22-33
used with assemblies 45-2	ProSolidToPostfixId() function
used with axis datums 5-10	description 3-18
used with composite datum curves 5-14	ProSolidUnitConvertPostAction callback 65-6
used with coordinate system datums 5-10	ProSolidUnitConvertPreAction callback 65-6
used with datum curves 5-13	ProSplinedataInit() function
used with datum planes 5-11	description 27-11
used with datum points 5-14	prosrfl_bind_srf_class() function 68-2
ProSolidMassPropertyGet() function	ProSrfcollectionAlloc() function
description 3-18	description 20-14
ProSolidMechitemVisit() function	ProSrfcollectionCopy() function
description 61-6	description 20-14
ProSolidModelclassGet() function	ProSrfcollectionInstructionAdd() function
description 22-35	description 20-14
ProSolidModelclassSet() function	ProSrfcollectionInstructionRemove() function
description 22-35	description 20-14
ProSolidNoteCreate() function	ProSrfcollectionInstructionsGet() function
description 22-50	description 20-11
ProSolidOutlineCompute() function	ProSrfcollectionRegenerate() function
description 3-8	description 20-14
ProSolidOutlineGet() function	ProSrfcollinstrAlloc() function
description 3-8, 22-54	description 20-14
ProSolidPrincSysUnitsRenamedPostAction	ProSrfcollinstrArrayFree() function
callback 65-6	description 20-12
ProSolidProjectPoint() function	ProSrfcollinstrFree() function
description 5-20	description 20-12

ProSrffcollinstrIncludeGet() function
 description 20-12
 ProSrffcollinstrIncludeSet() function
 description 20-14
 ProSrffcollinstrReferenceAdd() function
 description 20-14
 ProSrffcollinstrReferenceRemove() function
 description 20-14
 ProSrffcollinstrReferencesGet() function
 description 20-12
 ProSrffcollinstrTypeGet() function
 description 20-11
 ProSrffcollrefArrayFree() function
 description 20-13
 ProSrffcollreffree() function
 description 20-13
 ProSrffcollrefItemGet() function
 description 20-13
 ProSrffcollrefTypeGet() function
 description 20-12
 ProStringarrayFree() function
 description 12-7
 ProStringFree() function
 description 12-7
 ProStringproarrayFree() function
 description 12-7
 ProStringToWString() function
 description 1-64
 ProStringToWstring() function
 used in a code example 15-55
 ProStringVerstampGet() function
 description 2-13
 ProSurfaceAreaEval() function
 description 5-32
 ProSurfaceContourVisit() function
 description 5-7
 used with a datum surface 5-13
 ProSurfacedataAlloc() function
 description 27-5
 ProSurfacedataContourArraySet() function
 description 27-7
 ProSurfaceDataGet() function
 description 5-25
 used with datum planes 5-28
 used with datum surfaces 5-29
 ProSurfacedataInit() function
 description 27-5, 27-6
 ProSurfaceDiameterEval() function
 description 5-33
 ProSurfaceExtremesEval() function
 description 5-32
 ProSurfaceIdGet() function
 description 5-11
 ProSurfaceInit() function
 description 5-11
 ProSurfaceIsDatumPlane() function
 description 5-26
 ProSurfaceParamEval() function
 description 5-20
 ProSurfacePrincipalCrvtEval() function
 description 5-32
 ProSurfaceSameSrfsFind() function
 description 5-26
 ProSurfacesCollect() function
 description 20-4
 ProSurfaceTessellationGet() function
 description 5-16
 ProSurfacetessellationinputAlloc() function
 description 5-16
 ProSurfacetessellationinputAnglecontrolSet()
 function
 description 5-16
 ProSurfacetessellationinputChordheightSet()
 function
 description 5-16
 ProSurfacetessellationinputFree() function
 description 5-16
 ProSurfacetessellationinputUvprojectionSet()
 function
 description 5-17
 ProSurfaceToNURBS() function
 description 5-34
 ProSurfaceTypeGet() function
 description 5-25
 ProSurfaceUvpntVerify() function
 description 5-19
 ProSurfaceXyzdataEval() function
 description 5-18
 ProSurffinishCreate() function
 description 22-59
 ProSurffinishDataGet() function
 description 22-60
 ProSurffinishDelete() function
 description 22-61
 ProSurffinishModify() function
 description 22-60
 ProSurffinishNameGet() function
 description 22-60
 ProSurffinishNameSet() function
 description 22-60
 ProSurffinishReferencesGet() function
 description 22-60
 ProSurffinishValueGet() function
 description 22-60
 ProSurffinishValueSet() function
 description 22-61
 ProSymbolDesignate() function
 description 22-21
 ProSymbolDesignationVerify() function
 description 22-21
 ProSymbolUndesignate() function
 description 22-21

ProTermFuncSet() function	ProTexturePathGet() function
description 13-9	description 12-6
ProTerminationAction() function	ProTextWidthFactorCurrentSet() function
description 13-10	description 18-18
ProTessellationFacetsGet() function	protk.dat file 1-28
description 5-16	ProToleranceDefaultGet() function
ProTessellationFree() function	description 22-32
description 5-16	ProToolAttributesGet() function
ProTessellationNormalsGet() function	description 53-8
description 5-16	ProToolCreate() function
ProTessellationParamsGet() function	description 53-13
description 5-16	ProToolElemModelSet() function
ProTessellationVerticesGet() function	description 53-12
description 5-16	ProToolElemParamAdd() function
ProTextAttributesCurrentGet() function	description 53-10
description 18-18	ProToolInit() function
ProTextColorModify() function	description 53-13
description 18-10	ProToolinputAlloc() function
ProTextFontDefaultIdGet() function	description 53-9
description 18-18	ProToolinputElemAdd() function
ProTextFontIdCurrentSet() function	description 53-10
description 18-18	ProToolinputFree() function
ProTextFontIdGet() function	description 53-13
description 18-19	ProToolinputTypeSet() function
ProTextFontNameToId() function	description 53-9
description 18-19	ProToolkitApplExecPathGet() function
ProTextHeightCurrentSet() function	description 12-2
description 18-18	ProToolkitApplTextPathGet() function
ProTextRotationAngleCurrentSet() function	description 12-3
description 18-18	ProToolkitDllHandleGet() function
ProTextSlantAngleCurrentSet() function	description 69-9
description 18-18	ProToolkitDllIdGet() function
ProTextStyleAlloc() function	description 69-9
description 22-52	ProToolkitDllLoad() function
ProTextStyleAngleGet() function 22-53	description 69-8
ProTextStyleAngleSet() function 22-53	ProToolkitDllUnload() function
ProTextstyleColorGet() function	description 69-8
description 22-53	ProToolkitMain() function
ProTextstyleColorSet() function	description 1-48
description 22-53	ProToolkitTaskExecute() function
ProTextStyleFree() function	defined 69-8
description 22-52	ProToolkitUnicodeencodingDisable() function
ProTextStyleHeightGet() function 22-53	description B-23
ProTextStyleHeightSet() function 22-53	ProToolkitUnicodeencodingIsEnabled()
ProTextstyleJustificationGet() function	function
description 22-53	description B-23
ProTextstyleJustificationSet() function	ProToolModelGet() function
description 22-53	description 53-8
ProTextStyleMirrorGet() function 22-53	ProToolParamGet() function
ProTextStyleMirrorSet() function 22-53	description 53-8
ProTextStyleThicknessGet() function 22-53	ProToolTypeGet() function
ProTextStyleThicknessSet() function 22-53	description 53-8
ProTextStyleUnderlineGet() function 22-53	ProToolVerify() function
ProTextStyleUnderlineSet() function 22-53	description 53-8
ProTextStyleWidthGet() function 22-53	Prototyping 1-19
ProTextStyleWidthSet() function 22-53	ProTrailfileCommentWrite() function

description 12-3	description 16-11
ProTreetoolRefresh() function	ProUIDialogComponentsCollect() function
description 18-3	description 16-11
ProUdfLibraryCompletePostAction callback	ProUIDialogCreate() function
65-3	description 16-11
ProUIDashboardBitmapSet() function	ProUIDialogDestroy() function
description 17-4	description 16-12
ProUIDashboardDestroy() function	ProUIDialogExit() function
description 17-4	description 16-12
ProUIDashboardpageClose() function	ProUIDialogHide() function
description 17-7	description 16-12
ProUIDashboardpageComponentnameGet()	ProUIDialogMinimumsizeGet() function
function	description 16-11
description 17-7	ProUIDialogPositionReset() function
ProUIDashboardpageDashboardGet()	description 16-11
function	ProUIDialogReconfigure() function
description 17-6	description 16-11
ProUIDashboardpageDevicenameGet()	ProUIDialogScreenpositionGet() function
function	description 16-11
description 17-7	ProUIDialogShow() function
ProUIDashboardpageForegroundcolorSet()	description 16-12
function	ProUIDialogSizeGet() function
description 17-6	description 16-11
ProUIDashboardPageGet() function	ProUILayoutIsMapped() function
description 17-6	description 16-35
ProUIDashboardpageNameGet() function	ProUILayoutMappedSet() function
description 17-6	description 16-35
ProUIDashboardpageoptionsAlloc() function	ProUILayoutMappedUnset() function
description 17-5	description 16-35
ProUIDashboardpageoptionsFree() function	ProUIMessageDialogDisplay()function
description 17-6	description 14-2
ProUIDashboardpageoptionsNotificationSet()	ProUITranslationFilesEnable() function
function	description 16-112
description 17-5	ProUnitByExpressionInit() function
ProUIDashboardpageTitleSet() function	description 3-25
description 17-6	ProUnitConversionGet() function
ProUIDashboardpageUserdataGet() function	description 3-15
description 17-6	ProUnitCreate() function
ProUIDashboardShow() function	description 3-17
description 17-3	ProUnitDelete() function
ProUIDashboardshowoptionsAlloc() function	description 3-17
description 17-3	ProUnitExpressionGet() function
ProUIDashboardshowoptionsFree() function	description 3-16, 3-25
description 17-4	ProUnitInit() function
ProUIDashboardshowoptionsNotificationSet()	description 3-15
function	ProUnitInitByExpression() function
description 17-3	description 3-16
ProUIDashboardStdlayoutGet() function	ProUnitIsStandard() function
description 17-4	description 3-16
ProUIDashboardUserdataGet() function	ProUnitModify() function
description 17-4	description 3-16
ProUIDialogAboveactivewindowGet() function	ProUnitRename() function
description 16-11	description 3-17
ProUIDialogAboveactivewindowSet() function	ProUnitssystemDelete() function
description 16-11	description 3-14
ProUIDialogActivate() function	ProUnitssystemRename() function

- description 3-13
- ProUnitsystemIsStandard() function
 - description 3-14
- ProUnitsystemTypeGet() function
 - description 3-14
- ProUnitsystemUnitGet() function
 - description 3-14
- ProUnitTypeGet() function
 - description 3-15
- ProValue object
 - description 29-9
- ProValueAlloc() function
 - description 29-11
- ProValueArrayToWstringArray() function
 - description 29-11
- ProValueData structure
 - declaration 29-10
- ProValueDataGet() function
 - description 29-10
- ProValueDataSet() function
 - description 29-11
- ProValuedataStringSet() function
 - description 69-3
- ProValuedataTransformGet() function
 - description 29-11, 69-3
- ProValuedataTransformSet() function
 - description 29-11, 69-3
- ProValueDataType enum
 - declaration 29-10
- ProValuedataWstringSet() function
 - description 69-3
- ProValueFree() function
 - description 29-11
- ProVector typedef
 - declaration 8-4
- ProVectorTrfEval() function
 - description 8-5
 - used with assemblies 45-11
- ProVerstampAlloc() function 2-12
- ProVerstampEqual() function
 - description 2-13
- ProVerstampFree() function
 - description 2-13
- ProVerstampStringFree() function 2-12
- ProVerstampStringGet() function
 - description 2-13
- ProView
 - drawing views and models 51-16
- ProViewMatrixGet() function
 - description 8-6
 - fundamental 18-7
- ProViewMatrixSet() function
 - description 8-6, 18-7
 - used in a code example 18-9
- ProViewNamesGet() function
 - description 18-7
- ProViewReset() function 18-6
- ProViewRetrieve() function
 - description 18-7
- ProViewRotate() function 18-6
- ProViewStore() function
 - description 18-7
 - used in a code example 18-9
- ProWcharSizeVerify() function
 - description 1-64
- ProWeldCompoundGet
 - function defined 60-2
- ProWeldFilletDataGet
 - function defined 60-2
- ProWeldGrooveDataGet
 - function defined 60-2
- ProWeldInfoGet
 - function defined 60-2
- ProWeldIntermittenceGet
 - function defined 60-2
- ProWeldPlugDataGet
 - function defined 60-2
- ProWeldRodGet
 - function defined 60-2
- ProWeldRodNameGet
 - function defined 60-2
- ProWeldSequenceIDGet
 - function defined 60-2
- ProWeldSlotdataGet
 - function defined 60-2
- ProWeldSpotdataGet
 - function defined 60-2
- ProWeldTypeGet
 - function defined 60-2
- ProWExtobjdata object
 - description 66-6
- ProWExtobjRef object
 - description 66-13
- ProWindowActivate() function
 - description 18-6
- ProWindowChangePostAction callback 65-6
- ProWindowClear() function
 - description 18-3
- ProWindowCurrentClose() function
 - description 18-4
- ProWindowCurrentGet() function
 - description 18-4
- ProWindowCurrentMatrixGet() function
 - description 8-8, 18-6
- ProWindowCurrentSet() function
 - description 18-4
 - displaying graphics 18-15
- ProWindowDelete() function
 - description 18-4
- ProWindowMdlGet() function
 - description 18-5
- ProWindowOccupyPostAction callback 65-6

- ProWindowOccupyPostAction() function
 - description 15-69
- ProWindowPanZoomMatrixSet() function
 - description 8-8, 18-6
- ProWindowRefresh() function
 - and display lists 18-20
 - description 18-3
 - used in a code example 18-14
- ProWindowRepaint() function
 - description 18-3
 - to see an external object 66-7
- ProWindowSizeGet() function
 - description 18-2
- ProWindowsVisit() function
 - description 18-5
- ProWindowVacatePreAction callback 65-6
- ProWindowVacatePreAction() function
 - description 15-69
- ProWorkspaceDelete() function
 - description 25-7
- ProWsexportSecondarycontentoptionSet()
 - function
 - description 25-21
- ProWsimpexFolderoptionSet() function
 - description 25-19
- ProWsimpexmessageArrayFree() function
 - description 25-21
- ProWsimpexmessageDataGet() function
 - description 25-20
- ProWstringarrayFree() function
 - description 12-7
- ProWstringArrayToValueArray() function
 - description 29-11
- ProWstringCompare() function
 - description 1-67
- ProWstringConcatenate() function
 - description 1-67
- ProWstringCopy() function
 - description 1-67
- ProWstringFree() function
 - description 12-7
- ProWstringLengthGet() function
 - description 1-66
- ProWstringproarrayFree() function
 - description 12-7
- ProWstringToString() function
 - description 1-64
 - ensuring portability 1-66
- ProXsecGeometryArrayFree() function
 - description 11-4
- ProXsecGeometryCollect() function
 - description 11-3
- ProXsecGeometryFree() function
 - description 11-4
- ProXsecGeometryRetrieve function
 - description 11-3

- ProXsecPlaneGet() function
 - description 11-4
- Pushbutton
 - adding to menus 15-5
- Pushing and popping menus 15-58

Q

- Query
 - external databases 54-1
 - features 4-3
- Quilts
 - geometry 5-28
 - visiting 5-12

R

- Rays 5-30
- Read access
 - to weld features 30-2, 60-2
- Read status 4-6
- Recycle
 - object identifiers 66-4
- Redefine
 - features 4-8
- Reference-driven patterns 39-6
- References
 - external
 - defined 46-4
 - external object 66-12
- Refresh
 - window 18-3
- Regenerate
 - 2-D sections 40-7
 - assembly components 47-3
 - relation sets 6-3
 - solid objects 3-5
- Regeneration types 39-5
- Register
 - applications 1-33
 - external object classes 66-3
- Registry files 1-28
 - examples D-4
 - fields of D-2
 - functions 12-2
 - toolkit I-7
- Relations 6-2
- Remove
 - animation frames from a movie 21-5
 - animation objects from frames 21-4
 - external object data 66-11
 - external object references 66-13
 - family table items 9-7
 - highlighting 19-10
 - windows 18-4
- Rename
 - models 2-7

- surfaces
 - code example 2-11
- Reorder 4-10
- Repaint
 - windows 18-3
- Replace
 - section entities 40-9
 - tweak surface features 34-1
- Reporting errors
 - for sections 40-24
- Reposition process steps 56-6
- Reset
 - view 18-6
- Resolution
 - specifying 40-8
- Restart
 - applications 1-37
- Retrieve
 - 2-D sections 40-26
 - external data 10-5
 - family table instances 9-6
 - geometry of a simplified representation 49-6
 - graphics of a simplified representation 49-6
 - models 2-7
 - simplified representations 49-5
 - views 18-7
- Reusable identifiers 66-4
- Roles
 - of manufacturing components 53-5
- Rotate
 - views 18-6
- Rotation angle
 - text 18-18
- Routing 58-18
 - locations 58-16
- Ruled surfaces K-8
- Rules
 - simplified representation 49-15
- Run-time menus 15-59

S

- Sample applications, installing F-2
- Save
 - 2-D sections 40-26
 - models 2-7
- Scale
 - external objects 66-10
 - view 51-18
- Scope control
 - defined 46-4
- Screen coordinate system 8-2
- Scroll
 - messages 14-4
- Search

- APIWizard search feature (Find) 1-13
- Sections
 - 2-D
 - adding entities 40-9
 - saving 40-26
 - allocating 40-3
 - automatic dimensioning 40-9
 - constraints 40-5
 - creating
 - 2-D 40-2
 - example 40-26
 - models 40-2
 - definition 40-1
 - dimensions 40-12
 - entities 40-9
 - errors 40-24
 - example 40-26
 - regenerating 40-7
 - retrieving 40-26
 - saving 40-26
 - solving 40-7
- Select3d
 - compared to ProSelection I-15
 - converting I-13
 - mapping sel_type I-14
- Selection 19-2
 - boxes 66-10
 - size of 66-11
 - data
 - for external objects 66-10
 - explode states 45-15
 - family table instances 9-5
 - feature 4-5
 - interactive 19-5
 - MechanicalItems 61-4
- Session
 - simplified representations 49-3
- Set mode
 - definition Glossary-6
- Sheetmetal
 - flange wall feature
 - access 52-43
 - create 52-42
 - feature element tree 52-30
 - redefine 52-43
 - flat wall feature
 - access 52-17
 - create 52-16
 - feature element tree 52-10
 - introduction 52-10
 - redefine 52-16
- Sheets
 - drawing 51-12
- Shell pairs 63-12
- Simple asynchronous mode 13-7

- Simplified representations
 - adding items 49-14
 - creating 49-6
 - deleting 49-6
 - items 49-14
 - extracting information from 49-9
 - modifying 49-14
 - retrieving
 - geometry 49-6
 - graphics 49-6
 - rules 49-15
 - session 49-3
 - zones 49-17
- Single animation 21-4
- Single-valued element 29-4
- Size
 - gathering by 49-17
- Skeleton Model Functions 46-5
- Sketched features
 - create 41-4
 - create with 2D sections 41-5
 - creating features with 3D sections 41-6
 - element tree 41-2
 - overview 41-2
 - reference entities and use edge 41-8
 - reusing existing sketches 41-10
- Slant angle
 - text 18-18
- Slots 10-2
- Solids
 - accuracy 3-10
 - contents of 3-2
 - coordinate system 8-2
 - creating 3-2
 - displaying 3-4
 - geometry
 - visiting 5-6
 - mass properties 3-18
 - orientation 18-6
 - outline 3-8
 - regenerating 3-5
 - transforming coordinates 8-6
 - units 3-11
 - accessing individual units 3-15
 - accessing systems of units 3-14
 - conversion of models to a new unit system 3-17
 - creating a new system of units 3-14
 - creation of a new unit 3-17
 - modifying systems of units 3-13
 - modifying units 3-16
 - retrieving systems of units 3-13
- Solve
 - sections 40-7
- Sources
 - light 18-25
- Splines
 - cylindrical spline surface K-14
 - representation K-19
 - surface K-11
- Spools 58-5
- Start
 - batch animation 21-5
- Status
 - feature 4-4
- Statuses
 - of a Pro/ENGINEER process 13-9
 - read 4-6
- Stop
 - applications 1-37
- Storage solids 53-3
 - identifying 53-6
- Store
 - external data 10-4
 - view 18-7
- Stream
 - type of slot 10-2
- Strings
 - wide 1-64
 - functions for 12-6
- Structure
 - of applications I-7
- Submenus 15-55
 - definition Glossary-6
- Support
 - third-party tool manager 54-2
- Suppress
 - feature 4-7
- Surfaces
 - cylindrical spline K-14
 - data structures K-3
 - definition Glossary-6
 - fillet K-10
 - general surface of revolution K-7
 - manufacturing code example 53-39
 - NURBS K-13
 - parametric equations 5-25
 - properties of 18-23
 - renaming
 - code examples 2-11
 - replacement features 34-1
 - ruled K-8
 - spline K-11
 - traversing J-2
 - types 5-25
- Sweeps
 - code example 43-5, 43-14
 - creating 43-1
 - element tree 43-2
- Swing
 - class path

- NT 1-4
- UNIX 1-4
- download Java foundation class archive 1-4
- in APIWizard 1-3
- required for APIWizard with Netscape 1-3
- required Java Foundation Class for APIWizard 1-3
- Symbol definitions 51-50
- Symbol instances 51-50
- Symbols
 - designating 22-21
 - manipulating groups 51-89
- Synchronous J-Link Applications 69-17
- Synchronous mode
 - definition Glossary-7

T

- Tables
 - drawing 51-91
 - family
 - code example 9-7
 - objects 9-2
 - operations on instances 9-4
 - operations on items 9-7
 - utilities 9-2
 - visiting 9-3
- Tabulated cylinders K-9
- Task Library Functions
 - Memory Management 69-4
- Tessellation 5-15
 - definition Glossary-7
 - surface 18-23
- Text
 - attributes 18-18
 - displaying 18-17
 - fonts 18-18
 - message files 14-5
 - message window
 - identifier 18-1
 - note 22-50
 - style properties 22-52
 - styles
 - of notes 22-52
- The Feature Element Tree for Fill feature in Pro/ENGINEER 33-8
- The Feature Element Tree for Mirror feature in Pro/ENGINEER 33-3
- The Feature Element Tree for Move feature in Pro/ENGINEER 33-6
- The Feature Element Tree for Solidify feature in Pro/ENGINEER 33-24
- The Feature Element Tree for Thicken feature in Pro/ENGINEER 33-22
- The Feature Element Tree for Trim feature in

- Pro/ENGINEER 33-15
- Toolbar
 - placing 15-22
- Toolkit DLL
 - Create 69-3
 - Functions 69-8
 - Task Libraries 69-3
- Tools
 - auxiliary 53-11
 - contouring 53-11
 - creating 53-9
 - holemaking
 - parameters 53-12
 - milling
 - parameters for 53-11
 - table
 - creating 53-28
 - turning
 - parameters for 53-10
 - types 53-8
 - visiting 53-8
- Torus K-7
- Trail files 12-3
- Transformations
 - coordinates of an assembly member 8-12
 - coordinates of sketched entities 8-12
 - definition Glossary-7
 - drawing view to screen coordinates 8-9
 - external object 66-8
 - screen to drawing coordinates 8-9
 - screen to window coordinates 8-8
 - solid to screen coordinates 8-5
 - in a drawing 8-9
 - to coordinate system datum coordinates 8-12
- Traversal
 - assembly 45-5
 - geometry J-1
 - manufacturing components 53-5
 - part 5-3
- Turning tools
 - parameters 53-10

U

- UDFs 4-20
- uiCmdPriority enum
 - declaration 15-6
- Unicode Ecoding
 - acronym B-2
- Unicode Encoding B-2
 - Byte Order Mark B-3
 - definition B-2
 - External Interface Handling B-15
 - Legacy Encoding Mode B-23
 - mapping methods B-2

- Necessity B-4
- Pro/ENGINEER Wildfire 4.0 B-2
- Programming Considerations B-11
- registry file option B-23
- Support for Legacy Encoding B-21
- Transcoding B-3
- Unlock
 - messages 1-40
 - toolkit
 - comparison I-7
 - toolkit application 1-39
- use same version as ProENGINEER. 1-23
- User attributes
 - definition Glossary-7
- User Interface
 - dialog components
 - dialogs with menubars 16-119
 - programming dialog components
 - table inquiry functions 16-134
 - resource files
 - syntax 16-112
- user_change_directory() function
 - equivalent I-27
- user_change_window_post() function
 - equivalent I-27
- user_dbms_copy_post_all() function
 - equivalent I-27
- user_dbms_delete_post_all() function
 - equivalent I-27
- user_dbms_erase_post_all() function
 - equivalent I-27
- user_dbms_failure_function() function
 - equivalent I-27
- user_dbms_retrieve_post_all() function
 - equivalent I-27
- user_dbms_save_post_all() function
 - equivalent I-27
- user_eval_srf() function 68-2
- user_graphics_object_output_post() function
 - equivalent I-27
- user_graphics_object_output_pre() function
 - equivalent I-27
- user_init_srf() function 68-2
- user_initialize() function
 - adding a new menu button 15-44
 - changing the color map 18-13
 - description 1-42
- user_intf_arc() function 28-2
- user_intf_filled_poly() function 28-2
- user_intf_line() function 28-2
- user_intf_polyline() function 28-2
- user_intf_text() function 28-2
- user_menu_dbms_copy_post() function
 - equivalent I-26
- user_menu_dbms_copy_pre() function
 - equivalent I-26
- user_menu_dbms_create_post() function
 - equivalent I-27
- user_menu_dbms_create_pre() function
 - equivalent I-26
- user_menu_dbms_delete_post() function
 - equivalent I-26
- user_menu_dbms_delete_pre() function
 - equivalent I-26
- user_menu_dbms_erase_post() function
 - equivalent I-26
- user_menu_dbms_erase_pre() function
 - equivalent I-26
- user_menu_dbms_purge_post() function
 - equivalent I-26
- user_menu_dbms_purge_pre() function
 - equivalent I-26
- user_menu_dbms_rename_post() function
 - equivalent I-26
- user_menu_dbms_rename_pre() function
 - equivalent I-26
- user_menu_dbms_retrieve_post() function
 - equivalent I-27
- user_menu_dbms_retrieve_pre() function
 - equivalent I-27
- user_menu_dbms_save_post() function
 - equivalent I-26
- user_menu_dbms_save_pre() function
 - equivalent I-26
- user_mfg_feat_cl_file_post() function
 - equivalent I-27
- user_mfg_oper_cl_file_post() function
 - equivalent I-27
- user_proe_term_func() function
 - equivalent I-16
- user_rmov_srf() function 68-2
- user_terminate() function
 - defined 1-42
 - description 1-43
- User's Guide
 - browsing with APIWizard 1-11
 - documentation
 - online 1-2
 - online format 1-2
 - PDF format 1-2
- user-initialize()
 - described 1-44
- User-supplied main 1-48
- UsrDisppointsCollect()
 - function description 2-21
- Utilities 12-1
 - family table 9-2

V

- Values
 - feature element 29-5

- of feature elements 29-9
 - parameter 7-3
- Vectors
 - definition Glossary-7
- Version compatibility 1-35
- Version stamps 2-12
 - definition Glossary-7
- View matrix 18-6
- Views
 - definition Glossary-7
 - drawing 51-16
 - creating 51-24
 - listing
 - example 51-20
 - modifying 51-22
 - orientation 18-6
 - overlay
 - definition Glossary-5
 - scale 51-18
 - storing 18-7
- Visibility
 - of assembly components 45-8
 - of menu buttons 15-58
- Visit
 - animation frames 21-4
 - animation movie frames 21-5
 - assembly components 45-5
 - composite datum curves 5-14
 - coordinate system datums 5-10
 - datum axes 5-9
 - datum curves 5-13
 - datum planes 5-11
 - datum points 5-14
 - datum surfaces 5-12
 - explode states 45-14
 - external object references 66-13
 - external objects 66-6
 - family tables 9-3
 - feature geometry 5-4
 - features 4-2
 - functions
 - description 1-59
 - geometry objects 5-3
 - layers 2-14
 - manufacturing tools 53-8
 - notes 22-52
 - process step features 56-2
 - quilts 5-12
 - relation sets 6-3
 - simplified representations 49-10
 - solid geometry 5-6
 - windows 18-5
- Visiting
 - cross sections 11-6
 - displayed entities 2-20
 - gtols 23-2

W

- Warnings 66-16
- wchar_t 1-64
- Weld
 - drawing symbols 60-3
 - notification callbacks 60-3
 - symbol notification types 60-3
 - PRO_DRAWING_WELD_GROUPIDS_
 - GET 60-3
 - PRO_DRAWING_WELD_SYMPATH_G
 - ET 60-3
 - PRO_DRAWING_WELD_SYMTEXT_G
 - ET 60-3
- Weld features
 - read access to 30-2, 60-2
- WHandles
 - description 1-52
- Wide strings 1-64
 - checking your declaration 1-65
 - definition Glossary-7
 - functions 12-6
 - manipulating 1-66
- Width factor
 - text 18-18
- Window coordinate system 8-3
- Window matrix 18-6
- Windows
 - activating 18-6
 - closing 18-4
 - creating 18-4
 - current 18-4
 - definition Glossary-7
 - deleting 18-4
 - getting the owner 18-5
 - identifiers 18-1
 - information
 - definition Glossary-3
 - manipulating 18-2
 - matrix 18-6
 - orientation 18-6
 - pan and zoom matrix 18-6
 - repainting 18-3
 - visiting 18-5
- Wires
 - spools 58-5
- Workcells
 - creating 53-27
 - example 53-29
 - features
 - code example 53-6
- Workspace handles
 - description 1-52
- Write
 - a message to the Message Window 14-4
 - family table to a file 9-7

message to an internal buffer 14-8

Z

Zones 49-17

