Pro/ENGINEER® Wildfire

Pro/TOOLKIT®
User’s Guide
Marcus Meißner, is redistributed under the GNU General Public License. This license is from the original copyright holder and the Applet is provided WITHOUT WARRANTY OF ANY KIND. You may obtain a copy of the source code for the Applet at http://www.mud.de/se/ta (for a charge of no more than the cost of physically performing the source distribution), by sending an e-mail to leo@mud.de or marcus@mud.de—you are allowed to choose either distribution method. The source code is likewise provided under the GNU General Public License.

GTK+ The GIMP Toolkit are licensed under the GNU LPGL. You may obtain a copy of the source code at http://www.gtk.org/, which is likewise provided under the GNU LPGL.

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), is 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.

Parametric Technology Corporation, 140 Kendrick Street, Needham, MA 02494 USA
About This Guide

Purpose
Audience and Prerequisite Experience
Contents
Documentation
Conventions

Chapter 1: Fundamentals

Introduction to Pro/TOOLKIT
Online Documentation — Pro/TOOLKIT APIWizard
To Install the APIWizard
To Run the APIWizard
Web Browser Environments
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
SGI Hardware Platforms
Troubleshooting
Automatic Index Tree Updating
Netscape on NT
Internet Explorer on NT
The APIWizard Interface
Topic/Object Selection Frame
Display Frame
Navigating the Topic/Object Selection Tree
Browsing with the Pro/TOOLKIT APIWizard
OHandles .................................................................................................. 1-47
DHandles .................................................................................................. 1-48
Workspace Handles .................................................................................. 1-49
Expandable Arrays .................................................................................... 1-50
Example 2: Expandable Arrays .................................................................. 1-53
Visit Functions .......................................................................................... 1-56
Example 3: Listing Holes in a Model .......................................................... 1-58
Wide Strings ............................................................................................. 1-60
Defining wchar_t ........................................................................................ 1-61
Setting the Hardware Type ........................................................................ 1-61
Checking Your Declaration of wchar_t ...................................................... 1-62
Wide String Functions .............................................................................. 1-62

Chapter 2: The Selection Object ................................................................. 2-1
The Selection Object .................................................................................. 2-2
Unpacking a ProSelection Object ............................................................... 2-3
Building a ProSelection Object ................................................................. 2-4
ProSelection Function Examples ............................................................... 2-4
Interactive Selection .................................................................................. 2-5
Highlighting ............................................................................................... 2-5

Chapter 3: Messages .................................................................................. 3-1
Writing a Message using a Popup Dialog .................................................. 3-2
Example 1: Displaying the UI Message Dialog .......................................... 3-2
Writing a Message to the Message Window ............................................... 3-4
Text Message File Format and Restrictions .............................................. 3-5
Restrictions on the Text Message File ....................................................... 3-5
Contents of the Message File .................................................................. 3-6
Message Classification ................................................................................ 3-7
Writing a Message to an Internal Buffer .................................................... 3-9
Getting Keyboard Input ............................................................................ 3-9
Using Default Values ............................................................................... 3-9
Example 2: Displaying Messages and Retrieving Keyboard Input .......... 3-10

Chapter 4: Menus ...................................................................................... 4-1
Introduction ............................................................................................. 4-2
Menu Bar Buttons and Menus ................................................................. 4-2
Using the Trail File to Determine UI Names ........................................... 4-4
Adding a PushButton to a Menu Bar Menu ............................................ 4-4
Adding an Action to Pro/ENGINEER ...................................................... 4-5
Adding a Menu Button ............................................................................ 4-8
Adding a Check Button to a Menu Bar Menu ........................................ 4-10
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manipulating Menus</td>
<td>4-39</td>
</tr>
<tr>
<td>Data Menus</td>
<td>4-40</td>
</tr>
<tr>
<td>Calling ProMenuModeSet() and ProMenuDatamodeSet()</td>
<td>4-40</td>
</tr>
<tr>
<td>Setting Menu Buttons</td>
<td>4-41</td>
</tr>
<tr>
<td>Calling ProMenubuttonHighlight() and ProMenubuttonUnhighlight()</td>
<td>4-41</td>
</tr>
<tr>
<td>Controlling Accessibility of Menu Buttons</td>
<td>4-42</td>
</tr>
<tr>
<td>Pushing and Popping Menus</td>
<td>4-42</td>
</tr>
<tr>
<td>Run-time Menus</td>
<td>4-43</td>
</tr>
<tr>
<td>Entering Pro/ENGINEER Commands</td>
<td>4-44</td>
</tr>
<tr>
<td>Execution Rules</td>
<td>4-45</td>
</tr>
<tr>
<td>Specifying Keyboard Input</td>
<td>4-48</td>
</tr>
<tr>
<td>Chapter 5: User Interface Components</td>
<td></td>
</tr>
<tr>
<td>Introduction</td>
<td>5-2</td>
</tr>
<tr>
<td>Dialog Components</td>
<td>5-2</td>
</tr>
<tr>
<td>Example 1: Dialog with All Components</td>
<td>5-5</td>
</tr>
<tr>
<td>Dialogs with Menubars</td>
<td>5-10</td>
</tr>
<tr>
<td>Resource Files</td>
<td>5-11</td>
</tr>
<tr>
<td>Syntax of Resource Files</td>
<td>5-11</td>
</tr>
<tr>
<td>Example 2: Resource File for Dialog with Four Components on 2x2 Grid</td>
<td>5-14</td>
</tr>
<tr>
<td>Example 3: Resource File for Subgrid Dialog</td>
<td>5-15</td>
</tr>
<tr>
<td>Example 4: Resource File for Subgrid Dialog with Resize</td>
<td>5-16</td>
</tr>
<tr>
<td>Common Attributes</td>
<td>5-17</td>
</tr>
<tr>
<td>Example 5: Resource File with Offsets, Attachments, and Help Text</td>
<td>5-19</td>
</tr>
<tr>
<td>The Dialog Viewer</td>
<td>5-20</td>
</tr>
<tr>
<td>Dialog Programming: Common Functions</td>
<td>5-21</td>
</tr>
<tr>
<td>Dialog Attributes</td>
<td>5-21</td>
</tr>
<tr>
<td>Example 6: Resource File with Text Question, OK and Cancel Buttons</td>
<td>5-23</td>
</tr>
<tr>
<td>Example 7: Source for Dialog with Text Question, OK and Cancel Buttons</td>
<td>5-24</td>
</tr>
<tr>
<td>Programming Dialog Components</td>
<td>5-26</td>
</tr>
<tr>
<td>Programming with CascadeButtons</td>
<td>5-27</td>
</tr>
<tr>
<td>CascadeButton Attributes</td>
<td>5-27</td>
</tr>
<tr>
<td>CascadeButton Functions</td>
<td>5-27</td>
</tr>
<tr>
<td>Programming with Check Buttons</td>
<td>5-28</td>
</tr>
<tr>
<td>CheckButton Attributes</td>
<td>5-28</td>
</tr>
<tr>
<td>Check Button Functions</td>
<td>5-28</td>
</tr>
<tr>
<td>Programming with InputPanels</td>
<td>5-30</td>
</tr>
<tr>
<td>InputPanel Attributes</td>
<td>5-30</td>
</tr>
<tr>
<td>InputPanel Functions</td>
<td>5-32</td>
</tr>
</tbody>
</table>
Programming with Labels ............................................................... 5-34
  Label Attributes ........................................................................ 5-34
  Label Functions ....................................................................... 5-35
Programming with Layouts ............................................................ 5-35
  Layout Attributes ..................................................................... 5-35
  Layout Functions ................................................................... 5-37
Programming with Lists ................................................................. 5-38
  List Attributes ....................................................................... 5-38
  List Functions ....................................................................... 5-41
  Example 8: UI List resource file .......................................... 5-43
  Example 9: To use UI List Functions ...................................... 5-44
Programming with MenuBars ........................................................ 5-49
  MenuBar Attributes ............................................................... 5-49
  MenuBar Functions ................................................................ 5-50
  Example 10: Resource File for Dialog with Menubar .......... 5-50
Programming with MenuPanes ....................................................... 5-52
  MenuPane Attributes ............................................................ 5-52
  MenuPane Functions .............................................................. 5-52
Programming with OptionMenus .................................................... 5-53
  OptionMenu Attributes ........................................................ 5-53
  OptionMenu Functions .......................................................... 5-54
Programming with ProgressBars .................................................. 5-56
  ProgressBar Attributes ........................................................ 5-56
  Example 11: Progress Bar Resource File .............................. 5-57
  ProgressBar Functions .......................................................... 5-59
  How to Program a Dialog Containing a Progress Bar .......... 5-59
Programming with Pushbuttons .................................................... 5-60
  Pushbutton Attributes ............................................................ 5-60
  Pushbutton Functions .............................................................. 5-61
  Example 12: Component Visibility resource file.................. 5-61
  Example 13: Controlling Component Visibility or Sensitivity at Runtime .. 5-63
Programming with RadioGroups .................................................... 5-65
  RadioGroup Attributes ........................................................ 5-65
  RadioGroup Functions .......................................................... 5-66
Programming with Separators ....................................................... 5-67
  Separator Attributes ............................................................... 5-67
  Separator Functions ............................................................... 5-67
Programming with Sliders .............................................................. 5-67
  Slider Attributes ................................................................... 5-67
  Slider Functions .................................................................. 5-69
Chapter 7: Solids and Parts

Solid Objects ........................................................................................................... 7-2
  Creating a Solid .................................................................................................... 7-2
  Contents of a Solid ............................................................................................. 7-2
  Displaying a Solid .............................................................................................. 7-3
    Example 1: Loading and Displaying a Solid ............................................. 7-3
  Regenerating a Solid ......................................................................................... 7-4
    Example 2: Combining Regeneration Flags ............................................. 7-5
  Solid Outline ...................................................................................................... 7-5
    Example 3: Computing the Outline of a Solid ........................................ 7-6
  Solid Accuracy ................................................................................................... 7-7
  Solid Units ......................................................................................................... 7-7
  Mass Properties ................................................................................................ 7-8
Part Objects ............................................................................................................ 7-9
  Solid Postfix Identifiers .................................................................................... 7-9
  Density .............................................................................................................. 7-10
Material Properties ................................................................................................ 7-10
  Material Objects ............................................................................................... 7-10
  Accessing Material Data .................................................................................. 7-11
  Material Input and Output ............................................................................... 7-12
    Example 4: Reading Material Properties from a File ................................ 7-12
Notes ..................................................................................................................... 7-15
  Creating and Deleting Notes .......................................................................... 7-15
  Note Text .......................................................................................................... 7-15
  Visiting Notes .................................................................................................. 7-16
  Note Display .................................................................................................... 7-16
  Note Owner ..................................................................................................... 7-16
  Note Placement ............................................................................................... 7-16
  Note Text Styles .............................................................................................. 7-17
  Text Style Properties ...................................................................................... 7-18
    Example 5: Creating a Model Note ........................................................... 7-18

Chapter 8: Graphics and Object Display

Manipulating Windows .......................................................................................... 8-2
  Resizing Windows .............................................................................................. 8-2
Manipulating the Embedded Browser in Windows ............................................ 8-2
Repainting Windows .......................................................................................... 8-3
Controlling Which Window is Current ............................................................ 8-4
Creating and Removing Windows ................................................................... 8-4
Retrieving the Owner of a Window ............................................................... 8-5
Visiting Windows .............................................................................................. 8-5
Activating Windows ......................................................................................... 8-5
Solid Orientation............................................................................................... 8-6
Getting and Setting the View Matrix .............................................................. 8-6
Storing Named Views ....................................................................................... 8-7
Example 1: Saving Three Views ....................................................................... 8-7
Graphics Colors and Line Styles.................................................................... 8-10
Setting Colors to Desired Values ................................................................. 8-10
Setting Colors to Match Existing Entities ...................................................... 8-11
Example 2: Setting the Graphics Color to a Specific RGB Value .................... 8-11
Example 3: Setting The Graphics Color to Follow the Color of Pro/ENGINEER Entity ................................................................. 8-12
Modifying the Pro/ENGINEER Color Map .................................................... 8-13
Example 4: Modifying the Color of the HALF_TONE Display ......................... 8-13
Setting Line Styles for Pro/TOOLKIT Graphics .............................................. 8-14
Displaying Graphics ....................................................................................... 8-14
Example 5: Displaying Lines and Circles ....................................................... 8-16
Displaying Text................................................................................................. 8-17
Controlling Text Attributes .......................................................................... 8-17
Controlling Text Fonts .................................................................................... 8-18
Display Lists ..................................................................................................... 8-18
Getting Mouse Input ....................................................................................... 8-20
Example 6: Drawing a Rubber-Band Line ....................................................... 8-21
Cosmetic Properties ...................................................................................... 8-22
Surface Properties .......................................................................................... 8-22
Setting Light Sources ...................................................................................... 8-25

Chapter 9: Geometry ....................................................................................... 9-1
Geometry Objects ........................................................................................... 9-2
Visiting Geometry Objects ............................................................................. 9-3
Visiting Feature Geometry ............................................................................ 9-5
Visiting Solid Geometry ................................................................................ 9-7
Example 1: Finding the Surfaces Penetrated by a Hole ................................... 9-8
Visiting Axis Datums ..................................................................................... 9-10
Visiting Coordinate System Datums ............................................................ 9-11
Visiting Datum Planes .................................................................................. 9-11
Transforming from Drawing View to Screen Coordinates in a Drawing .......... 10-7
Transforming from Screen to Drawing Coordinates in a Drawing ................. 10-7
Transforming Coordinates of an Assembly Member ..................................... 10-8
Transforming to Coordinate System Datum Coordinates ................................ 10-8
Transforming Coordinates of Sketched Entities ............................................. 10-8
Example 2: Using Several Coordinate Transforms ...................................... 10-8

Chapter 11: Dimensions and Relations 11-1
The ProDimension Object .............................................................................. 11-2
Reading Dimensions ...................................................................................... 11-3
Modifying Dimensions ................................................................................... 11-4
ISO/DIN Tolerance Table Use ....................................................................... 11-5
Dimension Text ............................................................................................... 11-6
Displaying Dimensions ................................................................................. 11-6
Dimension Display Modes ............................................................................. 11-7
Designating Dimensions and Symbols ............................................................. 11-7
Dimension Tolerances .................................................................................. 11-8
Relations ......................................................................................................... 11-8

Chapter 12: Basic Access to Features 12-1
Feature Objects ............................................................................................... 12-2
Visiting Features ............................................................................................ 12-2
Feature Inquiry ............................................................................................... 12-3
Feature Geometry ............................................................................................ 12-6
Manipulating Features .................................................................................. 12-6
Example 1: Inserting a Feature ..................................................................... 12-7
Example 2: Changing the Regeneration Sequence of a Solid ....................... 12-9
Feature Dimensions ...................................................................................... 12-10
Manipulating Patterns .................................................................................. 12-10
Patterns as Features ...................................................................................... 12-10
Pro_Pattern Structure .................................................................................... 12-13
Pro_pattern_dir Structure .......................................................................... 12-14
Pro_pattern_dim Structure .......................................................................... 12-15
Table-Driven Patterns ................................................................................... 12-16
Creating Local Groups ................................................................................... 12-17
Read Access to Groups .................................................................................. 12-18
Finding Groups ............................................................................................... 12-19
Group Information ......................................................................................... 12-20
Creating Groups ............................................................................................. 12-20
The UDF Input Data Structure—ProUdfdata ............................................... 12-23
Variable Dimensions and Pattern Parameters ............................................. 12-24
UDF References............................................................................................ 12-25
Assembly Intersections .............................................................................. 12-26
Component Model Names.......................................................................... 12-27

**Chapter 13: Assemblies**  13-1

Structure of Assemblies and Assembly Objects .................................................... 13-2
Visiting Assembly Components ......................................................................... 13-5
  Properties Related to Component Purpose...................................................... 13-5
  Component Placement....................................................................................... 13-7
  Simplified Representations.............................................................................. 13-7
  Modifying Component Properties.................................................................... 13-8
Regenerating an Assembly Component ............................................................. 13-11
Locations of Assembly Components ................................................................. 13-11
Assembling Components .................................................................................. 13-13
Redefining and Rerouting Components ............................................................. 13-13
Deleting Components ....................................................................................... 13-14
  Exploded Assemblies..................................................................................... 13-14
  Explode State Objects.................................................................................. 13-14
  Visiting Explode States................................................................................ 13-15
  Explode State Access..................................................................................... 13-15
Merge and Cutout ............................................................................................ 13-15
Automatic Interchange...................................................................................... 13-16

**Chapter 14: Parameters**  14-1

Parameter Objects............................................................................................ 14-2
Parameter Values............................................................................................... 14-3
Accessing Parameters ....................................................................................... 14-4
  Example 1: Labeling a Feature with a String Parameter .......................... 14-5
Designating Parameters for Pro/INTRALINK...................................................... 14-7

**Chapter 15: Geometric Tolerancing**  15-1

Geometric Tolerance Objects........................................................................... 15-2
  ProGtol .......................................................................................................... 15-2
  ProGtoldata ................................................................................................ 15-2
Geometric Tolerance Datum References........................................................... 15-2
Visiting Geometric Tolerances.......................................................................... 15-3
Reading Geometric Tolerances........................................................................ 15-4
Creating a Geometric Tolerance....................................................................... 15-6
  Example 1: Attaching a Position GTOL to Solid Planes Parallel to a
  Datum Plane .................................................................................................. 15-8
Editing a Geometric Tolerance.......................................................................... 15-13
Deleting a Geometric Tolerance....................................................................... 15-14
Chapter 18: Creating Datum Curves 18-1

Datum Curve Features ........................................................................................................... 18-2
Common Elements .................................................................................................................. 18-2
Datum Curve Types .................................................................................................................. 18-3
  Sketched Datum Curves ....................................................................................................... 18-3
  Trim Datum Curves ............................................................................................................ 18-3
  Intersect Datum Curves ...................................................................................................... 18-4
  Wrap Datum Curves ........................................................................................................... 18-6
  Offset Datum Curves ......................................................................................................... 18-6
  Tangent Offset Datum Curves .......................................................................................... 18-7
Other Datum Curve Types ..................................................................................................... 18-8

Chapter 19: Creating Chamfer Features 19-1

Edge Chamfers in Pro/ENGINEER Wildfire ........................................................................... 19-2
  The Feature Element Tree for Edge Chamfers in Pro/ENGINEER Release 2001 ................. 19-2
Corner Chamfers .................................................................................................................. 19-11

Chapter 20: Creating Hole Features 20-1

Overview .................................................................................................................................. 20-2
The Element Tree for Hole Features ....................................................................................... 20-2
Feature Element Data Types .................................................................................................. 20-4
Common Element Values ....................................................................................................... 20-8
PRO_E_HLE_COM Values ....................................................................................................... 20-8
  Straight Hole ..................................................................................................................... 20-8
  Sketched Hole .................................................................................................................... 20-17
    Procedure Using Techniques from Creating Sketched Features ......................................... 20-17
    Sketched Hole with Conventional Approach ..................................................................... 20-18
    Example 1: Creating a Standard Sketched Hole with Linear Placement ......................... 20-18
    Procedure Using ProFeatureCreate() .............................................................................. 20-29
    Sketched Hole with ProFeatureCreate() ......................................................................... 20-29
  Standard Threaded Hole ...................................................................................................... 20-39
  Standard Threaded Hole .................................................................................................... 20-40
    Example 2: Creating a Standard Threaded Hole with Linear Placement ......................... 20-40
  Standard Clearance Hole .................................................................................................... 20-51
    Standard Clearance Hole ............................................................................................... 20-52
    Example 3: Creating a Standard Clearance Hole with Linear Placement ......................... 20-52
Valid PRO_E_HLE_COM Sub-Elements ................................................................................. 20-62
  Hole Placement Types ........................................................................................................ 20-67
  Hole Placement ................................................................................................................... 20-67
<table>
<thead>
<tr>
<th>Linear Hole on a Plane</th>
<th>20-68</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radial Hole on Plane with Radial Dimensioning</td>
<td>20-69</td>
</tr>
<tr>
<td>Radial Hole on Plane with Diameter Dimensioning</td>
<td>20-69</td>
</tr>
<tr>
<td>Radial Hole on Plane with Linear Dimensioning</td>
<td>20-70</td>
</tr>
<tr>
<td>Radial Hole on Cone or Cylinder</td>
<td>20-70</td>
</tr>
<tr>
<td>Coaxial Hole with Axis as Primary Reference</td>
<td>20-70</td>
</tr>
<tr>
<td>Coaxial Hole with Primary Reference not Axis</td>
<td>20-71</td>
</tr>
<tr>
<td>Onpoint Hole</td>
<td>20-71</td>
</tr>
<tr>
<td>Miscellaneous Information</td>
<td>20-71</td>
</tr>
<tr>
<td>Hole Parameter Files</td>
<td>20-71</td>
</tr>
<tr>
<td>Hole Diameter</td>
<td>20-71</td>
</tr>
<tr>
<td>Order of Element Specification</td>
<td>20-72</td>
</tr>
</tbody>
</table>

### Chapter 21: Creating Patterns

21-1

**Introduction** 21-2

**The Element Tree for Pattern Creation** 21-2

Reference-Driven Patterns 21-5

Dimension-Driven Patterns 21-5

Table-Driven Patterns 21-6

**Creating a Pattern** 21-8

Example 1: Creating a Pattern 21-9

**Obtaining the Element Tree for a Pattern** 21-13

### Chapter 22: Assembling Components

22-1

**Assembling Components by Functions** 22-2

Assembling a Component Parametrically 22-3

Example 1: Component Constraints 22-6

Example 2: Assembling Components 22-10

Redefining Components Interactively 22-12

Assembling Components by Feature Creation 22-13

**The Element Tree for a Component Assembly** 22-14

Example 3: Assembling a Part into an Assembly 22-15

### Chapter 23: Sections

23-1

**Overview** 23-2

**Creating Section Models** 23-2

To Create and Save a Section Model 23-2

Allocating a Two-Dimensional Section 23-3

Adding Section Entities 23-4

Adding Section Dimensions 23-5

Solving and Regenerating a Section 23-8

Automatic Section Dimensioning 23-10
Section Constraints ............................................................. 23-10
Error Reporting ................................................................. 23-12
Retrieving and Saving a Section ........................................... 23-13
Example 1: Creating a Section Model ................................. 23-14

Chapter 24: Creating Sketched Features .......................... 24-1
Overview ............................................................................. 24-2
Element Tree for Sketched Features ................................. 24-2
Creating Features Containing Sections .............................. 24-3
Creating Features with 2D Sections ................................. 24-5
Creating Features with 3D Sections ................................. 24-5
3D Section Location in the Owning Model ....................... 24-5
Reference Entities and Use Edge ........................................ 24-6
Reusing Existing Sketches .................................................. 24-8
Example 1: Creating an Extruded Protrusion Base Feature ... 24-8
Example 2: Creating a Sketched Datum Curve .................... 24-12
Example 3: Creating Sweep Protrusion Feature .................. 24-19

Chapter 25: Creating Extruded and Revolved Features .... 25-1
The Element Tree for Extruded Features ......................... 25-2
Examples: Creating Extruded Features ......................... 25-7
Conventional Approach .................................................... 25-8
Example 1: Creating an Extruded Feature ....................... 25-8
Example 2: To Create an Extruded Cut with Two-sided Thru-all Depth . 25-17
Example 3: To Create an Extruded Thin Cut ..................... 25-25
Example 4: To create an Extruded Datum Surface Feature .... 25-35
Example 5: To Create a Surface Trim Extruded Feature .... 25-44
Direct Creation Approach ............................................... 25-53
The Element Tree for Revolved Features ......................... 25-58
Examples: Creating Revolved Features ......................... 25-63
Example 6: To Create a Revolved Protrusion .................... 25-64
Example 7: To Create a Revolved Thin Cut ..................... 25-72
Example 8: To Create a Revolved Surface ...................... 25-81
The Element Tree for First Features ............................... 25-90
Example 9: Creating the First Extruded Protrusion Feature by Conventional Approach ............................. 25-92
Example 10: Creating the First Thin Revolve Protrusion Feature by Conventional Approach ...................... 25-98
Chapter 26: Creating Swept Features  26-1
  Sweeps in Pro/ENGINEER Wildfire .............................................................. 26-2
  The Element Tree for Sweeps .................................................................. 26-3
  Creating a Swept Feature ........................................................................... 26-5
    Example 1: Creating a Swept First Feature Protrusion ......................... 26-5
    Example 2: Creating a Sweep Protrusion Feature by
      Conventional Approach ....................................................................... 26-13

Chapter 27: Foreign Datum Surfaces and Curves  27-1
  Foreign Datum Surfaces ............................................................................... 27-2
  Importing a Foreign Datum Surface ............................................................ 27-2
  Foreign Datum Curves .................................................................................. 27-3
    Providing an Evaluation Function .............................................................. 27-4
    Curve Continuity ....................................................................................... 27-6
    Binding the Evaluation Function to a Class .............................................. 27-7
    Example 1: Creating a Sinusoidal Foreign Datum Curve ..................... 27-7

Chapter 28: Tweak Surface Replacement  28-1
  The Feature Element Tree ........................................................................... 28-2

Chapter 29: External Analyses and Analysis Features  29-1
  Overview ....................................................................................................... 29-2
  Interactive Creation of External Analysis .................................................... 29-3
  Interactive Creation of External Analysis Feature ....................................... 29-3
  Storage of External Analysis Feature in Pro/ENGINEER ......................... 29-4
  Registering an External Analysis with Pro/ENGINEER ............................. 29-5
  Analysis Callbacks ....................................................................................... 29-5
  Pro/TOOLKIT Analysis Information ............................................................ 29-8
  Results Data ................................................................................................ 29-8
    ProAnalysisSrfData structure .................................................................. 29-10
    Example 1: Offset Coordinate System Datum ....................................... 29-11
  Analysis Attributes ...................................................................................... 29-29
  Visiting Saved External Analyses ............................................................... 29-29
  Visiting External Analyses Features ........................................................... 29-30
  Using the Model without Pro/TOOLKIT ...................................................... 29-30

Chapter 30: Manufacturing  30-1
  Manufacturing Models .................................................................................. 30-2
  Creating a Manufacturing Model ................................................................. 30-3
  Analyzing a Manufacturing Model .............................................................. 30-3
    Traversing Manufacturing Components .................................................. 30-5
    Identifying the Storage Solid .................................................................. 30-6
    Example 1: Identifying Workcell Features of a Pro/NC Model .............. 30-6
# Chapter 34: Family Tables

- Family Table Objects ................................................................. 34-2
- Family Table Utilities...................................................................... 34-2
- Visiting Family Tables .................................................................... 34-3
- Operations on Family Table Instances ............................................ 34-4
- Operations on Family Table Items ................................................ 34-6
  - Example 1: Writing a Family Table to a File ............................ 34-6

# Chapter 35: External Data

- Introduction to External Data.......................................................... 35-2
- Storing External Data......................................................................... 35-4
- Retrieving External Data................................................................. 35-5

# Chapter 36: Notify

- Notify Classes .................................................................................. 36-2
  - Model Modification Events.......................................................... 36-4
- Using Notify ........................................................................................ 36-5
  - Interface Changes for Release 20 Forward................................. 36-5
  - Notification Events ....................................................................... 36-6
  - Canceling a Notification ............................................................... 36-8

# Chapter 37: Welds

- Read Access to Weld Features .......................................................... 37-2
- Customizing Weld Drawing Symbols .................................................. 37-3
  - Example 1: Weld Callback Notification ...................................... 37-4

# Chapter 38: Simplified Representations

- Overview ............................................................................................ 38-2
- Simplified Representations in Session .............................................. 38-3
- Retrieving Simplified Representations ............................................ 38-5
- Creating and Deleting Simplified Representations .......................... 38-6
  - Example 1: Creating a Simplified Representation ....................... 38-7
  - Example 2: Visiting the Items in a Simplified Representation ....... 38-9
- Modifying Simplified Representations .............................................. 38-13
  - Adding Items to and Deleting Items from a Simplified Representation 38-13
- Gathering Components by Rule ....................................................... 38-14
  - Gathering by Model Name .......................................................... 38-15
  - Gathering by Parameters ............................................................ 38-15
  - Gathering by Zone ........................................................................ 38-16
  - Gathering by Distance from a Point .......................................... 38-16
  - Gathering by Size ......................................................................... 38-16
  - Gathering by Simplified Representation .................................. 38-17
Chapter 39: Drawings

Creating Drawings from Templates ................................................................. 39-2
  Example 1: Drawing Creation from a Template ........................................ 39-3
Diagnosing Drawing Creation Errors ............................................................. 39-4
Drawing Setup .................................................................................................. 39-6
Access Drawing Location in Grid .................................................................... 39-7
Drawing Sheets .................................................................................................. 39-7
  Example 2: Listing Drawing Sheets ........................................................... 39-8
Drawing Views and Models ............................................................................ 39-10
  Listing Drawing Views ............................................................................. 39-10
  Example 3: Listing the Views in a Drawing .............................................. 39-11
  Modifying Views ...................................................................................... 39-11
  Example 4: Replace Drawing Model Solid with a Solid ............................ 39-14
  Drawing Models ....................................................................................... 39-14
  Example 5: Creating Drawing Views ....................................................... 39-16
  Background Views ................................................................................... 39-19
Detail Items ..................................................................................................... 39-20
  Listing Detail Items .................................................................................. 39-20
  Displaying Detail Items .......................................................................... 39-21
  Creating, Modifying and Reading Detail Items ....................................... 39-23
  Draft Entity Data ...................................................................................... 39-26
    Example 6: Create a Draft Line with Predefined Color ....................... 39-28
  Detail Note Text Data ............................................................................. 39-30
  Detail Note Line Data ............................................................................. 39-31
  Detail Attachments and Leaders .............................................................. 39-32
  Detail Note Data ...................................................................................... 39-34
    Example 7: Create Drawing Note at Specified Location with
      Leader to Surface and Surface Name .................................................. 39-36
  Read-Only Notes .................................................................................... 39-38
  Parameterized Note Text ....................................................................... 39-39
  Symbol Definition Attachments .............................................................. 39-40
  Symbol Definition Data ........................................................................... 39-40
  Creating a Symbol Definition .................................................................. 39-41
    Example 8: Create Symbol Definition ................................................ 39-42
  Retrieving a Symbol Definition from Disk .............................................. 39-45
  Symbol Instance Variable Text ............................................................... 39-46
  Symbol Instance Data ............................................................................. 39-46
    Example 9: Create Free Instance of Symbol Definition ..................... 39-49
Pro/ENGINEER Cabling Design ................................................................. 42-6
Creating a Harness ............................................................................... 42-7
Finding a Harness ............................................................................... 42-7
Finding the Cables in a Harness ............................................................... 42-7
Managing Spools .................................................................................. 42-8
Spool Parameters .................................................................................. 42-8
Finding Harness Connectors .................................................................. 42-9
Connectors Parameters .......................................................................... 42-10
Managing Cables and Bundles ................................................................. 42-13
Cable Parameters .................................................................................. 42-13
Cable Identifiers and Types .................................................................. 42-14
Cable Cosmetic Features Create ............................................................ 42-14
Cable Connectivity ............................................................................... 42-15
Cable Parameters .................................................................................. 42-15
Cable Routing Locations ...................................................................... 42-17
Cable Geometry .................................................................................. 42-18
Measuring Harness Clearance ................................................................. 42-18
Cable Routing ...................................................................................... 42-19
Deleting Cable Sections ...................................................................... 42-20

Chapter 43: Finite Element Modeling (FEM) ........................................... 43-1
Overview ............................................................................................ 43-2
Exporting an FEA Mesh ....................................................................... 43-3
Getting Constraint Case Names ............................................................ 43-5
Getting Mesh Controls ......................................................................... 43-5
       Pro_fem_region Data Structure ............................................................. 43-7
Getting Constraints ............................................................................ 43-8
       Pro_fem_con_value Data Structure ......................................................... 43-10
Getting Bar Elements, Contacts, and Runners .................................... 43-11
Getting Mass Elements ........................................................................ 43-12
       Pro_fem_mass_props Data Structure ....................................................... 43-13
Getting Shell Pairs ............................................................................... 43-13
Example 1: Using the FEM Functions ..................................................... 43-14

Chapter 44: External Objects ................................................................. 44-1
Summary of External Objects ................................................................. 44-2
External Objects and Object Classes ..................................................... 44-3
Creating External Object Classes .......................................................... 44-3
Creating External Objects .................................................................... 44-3
External Object Owners ...................................................................... 44-4
Recycling External Object Identifiers ..................................................... 44-4
External Object Parameters .................................................................. 44-5
External Types and Identifiers for External Objects ........................................ 44-5
Visiting External Objects.................................................................................. 44-6
External Object Data.............................................................................................. 44-6
Display Data for External Objects........................................................................ 44-6
  Allocating Display Data ........................................................................... 44-7
  Creating the External Object Entity ...................................................... 44-7
  Transformation of the External Object .................................................... 44-8
  External Object Display Properties ....................................................... 44-8
  External Object Color ...................................................................... 44-9
  Line Styles for External Objects .......................................................... 44-9
  External Object Scale...................................................................... 44-10
Selection Data for External Objects............................................................... 44-10
Manipulating External Object Data.................................................................. 44-11
External Object References ................................................................................. 44-12
  Creating External Object References ................................................... 44-13
  Visiting External Object References ....................................................... 44-14
Callbacks for External Objects............................................................................. 44-14
Warning Mechanism for External Objects............................................................ 44-17
  Example 1: Creating an External Object .............................................. 44-18

Chapter 45: Design Manager 45-1
Overview ........................................................................................................ 45-2
  Defining Design Intent ........................................................................ 45-2
  Defining Preliminary Product Structure ........................................... 45-2
  Introducing Skeleton Models ............................................................ 45-2
  Communicating Design Intent Throughout the Assembly Structure ....... 45-3
  Continued Population of the Assembly ............................................. 45-3
  Managing Part Interdependencies ....................................................... 45-3
Skeleton Model Functions...................................................................................... 45-5
Assembly Component Functions ........................................................................ 45-5
External Reference Control Functions ............................................................ 45-6
Feature and CopyGeom Feature Functions........................................................... 45-7
External Reference Data Gathering ................................................................... 45-9

Chapter 46: Animation 46-1
Introduction ..................................................................................................... 46-2
Animation Objects.......................................................................................... 46-3
Animation Frames.......................................................................................... 46-4
Playing Animations......................................................................................... 46-4
  Single Animation................................................................................. 46-5
  Batch Animation............................................................................... 46-5
Chapter 51: Importing Features

Creating Import Features from Files .......................................................... 51-2
Creating Import Features from Arbitrary Geometric Data ..................... 51-3
Allocating ProInterfacedata ................................................................. 51-4
Adding Surfaces ...................................................................................... 51-4
    Initializing Surface Data ................................................................. 51-5
Surfacedata Contours ............................................................................. 51-7
Appending the Surface Data to the Interface Data .............................. 51-8
Adding Edges ......................................................................................... 51-8
    Appending the Edge Data to the Interface Data ............................... 51-11
Adding Quilts ......................................................................................... 51-11
Adding Datums ....................................................................................... 51-12
    Initializing Datums ........................................................................ 51-12
    Appending the Datum data to the Interface data ............................ 51-13
Creating Features from the Interface Data ............................................. 51-13
Import Feature Attributes ....................................................................... 51-14
Redefining the Import Feature ............................................................... 51-15
Import Feature Properties ....................................................................... 51-16
Extracting Pro/ENGINEER Geometry as Interface Data....................... 51-18

Chapter 52: Asynchronous Mode

Overview .................................................................................................. 52-2
    Setting up an Asynchronous Pro/TOOLKIT Application .................. 52-3
Simple Asynchronous Mode ................................................................. 52-3
    Starting and Stopping Pro/ENGINEER ............................................ 52-4
Connecting to a Pro/ENGINEER Process .......................................... 52-5
    Example 1: Connecting to Pro/ENGINEER ..................................... 52-6
    Example 2: Retrieve, Calculate, Delete .......................................... 52-7
Status of a Pro/ENGINEER Process ..................................................... 52-8
Chapter 53: Task Based Application Libraries

ProArgument and Argument Management ............................................................... 53-2
Creating Pro/TOOLKIT DLL Task Libraries ......................................................... 53-3
Memory Management in Task Library Functions .................................................. 53-4
Launching Pro/TOOLKIT DLL Functions .............................................................. 53-8
Launching Synchronous J-Link Applications ......................................................... 53-9

Chapter 54: Help

How Help Files Work ............................................................................................. 54-2
Creating Help Files ............................................................................................... 54-3

Appendix A: Summary of Technical Changes

Updates for Pro/TOOLKIT Wildfire .......................................................................... A-2
Critical Functions.................................................................................................. A-2
External Analysis ............................................................................................... A-2
Feature Element Trees ......................................................................................... A-2
Patterns ............................................................................................................. A-6
ProAsmcompMdlNameGet() ............................................................................. A-6
ProFeatureCreatePreAction ............................................................................. A-6
Selection ........................................................................................................... A-7
System Colors .................................................................................................. A-7
Top Level Menu Manager Menus ......................................................................... A-7
Version String .................................................................................................. A-7
Pro2dImportCreate() ...................................................................................... A-7
Functions ProFemmeshExport(), pro_export_fea_mesh() ............................... A-8
ProFileOpen() ............................................................................................... A-8
ProMdlDataGet(), prodb_get_object_info() .................................................. A-8
New Functions.................................................................................................. A-8
Superseded Functions ....................................................................................... A-20
Miscellaneous Technical Changes ........................................................................ A-21
Datum Planes ..................................................................................................... A-21
Field Datum Point ............................................................................................ A-21
Sketched Datum Point ....................................................................................... A-21
Rib Feature ....................................................................................................... A-21
Notification....................................................................................................... A-21
Platform Support .............................................................................................. A-22
64-bit installations/32-bit applications ............................................................... A-22
Platform support: HPUX 11.0 support ............................................................... A-22
Drawings......................................................................................................... A-22
Simplified Representations............................................................................ A-22
Symbolic....................................................................................................... A-22
Parts.......................................................................................................... A-23
Sketcher ....................................................................................................... A-23
User interface ............................................................................................... A-23
Data Exchange ............................................................................................. A-23
  Layouts, Diagrams and Reports............................................................... A-23
  Importing MEDUSA.............................................................................. A-23
  Import Feature...................................................................................... A-23
Parameters .................................................................................................. A-24
ProSolidRegenerate .................................................................................... A-24
UDF Placement ........................................................................................... A-25

Appendix B: Pro/TOOLKIT Registry File ......................................................... B-1
  Registry File Fields.................................................................................. B-2
  Sample Registry Files ............................................................................ B-4

Appendix C: Debugging Pro/TOOLKIT Applications ...................................... C-1
  Debugging in Existing Process ............................................................... C-2
    Example 1: Debugging a Program that is Already Running............... C-2
  Debugging in a Separate Window.......................................................... C-3
    Example 2: Invoking the Debugger as a Separate Process.............. C-3
    Example 3: Debugging Applications on NT/Windows 95 System.... C-4
    Example 4: Debugging Pro/TOOLKIT DLL Applications.............. C-5

Appendix D: Pro/DEVELOP to Pro/TOOLKIT Function Mapping ..................... D-1
  The Relationship Between Pro/TOOLKIT and Pro/DEVELOP................. D-2
  Converting from Pro/DEVELOP ............................................................. D-2
    Using Pro/DEVELOP Applications with Pro/TOOLKIT.................. D-3
    Techniques of Conversion and Mixing ............................................ D-4
      Terminology ................................................................................... D-4
      General Functionality ................................................................. D-4
      Registry Files ............................................................................... D-7
      Menu and Message Files ............................................................. D-7
      Unlocking Your Application ....................................................... D-7
      Application Program Structure ............................................... D-7
      Handles and Data Types ............................................................. D-7
    Equivalent Pro/DEVELOP Functions ......................................... D-16
    Data Structures for Pro/DEVELOP-Style Functions...................... D-30
      Prodtl_item Structure Definition ............................................. D-30
      Data Structure Constants ......................................................... D-32
Contents - xxviii

Pro/TOOLKIT User's Guide

Ptc_surf Structure Definition.......................................................... D-33
Data Structure Constants ............................................................... D-38
Select3d Structure Definition......................................................... D-38
Data Structure Constants ............................................................... D-39

Appendix E: Pro/TOOLKIT Objects E-1
List of Pro/TOOLKIT Objects .......................................................... E-2

Appendix F: Geometry Traversal F-1
Overview ......................................................................................... F-2
Geometry Terms ............................................................................. F-2
Example 1 ....................................................................................... F-3
Example 2 ....................................................................................... F-4
Example 3 ....................................................................................... F-4
Example 4 ....................................................................................... F-5
Example 5 ....................................................................................... F-6

Appendix G: Geometry Representations G-1
Domain of Evaluation......................................................................... G-2
Surface Data Structures ..................................................................... G-3
Plane ................................................................................................. G-4
Cylinder ......................................................................................... G-5
Cone ................................................................................................. G-6
Torus ................................................................................................. G-7
General Surface of Revolution ......................................................... G-8
Ruled Surface ................................................................................ G-9
Tabulated Cylinder ......................................................................... G-10
Coons Patch .................................................................................. G-11
Fillet Surface ................................................................................ G-12
Spline Surface ............................................................................... G-13
NURBS Surface ............................................................................ G-14
Cylindrical Spline Surface ............................................................. G-16
Edge and Curve Data Structures .................................................. G-18
Line ................................................................................................. G-19
Arc ................................................................................................. G-20
Spline ........................................................................................... G-21
NURBS .......................................................................................... G-22
About This Guide

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

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>ii</td>
</tr>
<tr>
<td>Audience and Prerequisite Experience</td>
<td>ii</td>
</tr>
<tr>
<td>Contents</td>
<td>ii</td>
</tr>
<tr>
<td>Documentation</td>
<td>vii</td>
</tr>
</tbody>
</table>
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 following chapters:

Chapter 1, Fundamentals—Describes the fundamental concepts and functions used by Pro/TOOLKIT.

Chapter 2, The Selection Object—Describes 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.
Chapter 3, Messages—Describes the functions used to communicate with the user through the text message area.

Chapter 4, Menus—Describes the Pro/TOOLKIT functions that enable you to create and manipulate menus.

Chapter 5, User Interface Components—Describes the new User Interface (UI) functionality.

Chapter 6, Modes, Models, and Model Items—Describes the Pro/TOOLKIT functions used to access and manipulate modes, models, and model items.

Chapter 7, Solids and Parts—Describes how to access solids and parts, and their contents.

Chapter 8, Graphics and Object Display—Describes Pro/TOOLKIT functions that create and manipulate graphics and object displays.

Chapter 9, Geometry—Describes the objects and actions used to extract the geometry of a Pro/ENGINEER solid.

Chapter 10, Coordinate Systems and Transformations describes the various coordinate systems used by Pro/ENGINEER and Pro/TOOLKIT, and how to transform coordinates from one to another.

Chapter 11, Dimensions and Relations—Describes the Pro/TOOLKIT functions that deal with dimensions of parts and assemblies.

Chapter 12, Basic Access to Features—Describes the functions that deal with features as a whole, and the way they relate to each other.

Chapter 13, Assemblies—Describes the Pro/TOOLKIT functions that access the contents of a Pro/ENGINEER assembly.

Chapter 14, Parameters—Describes the functions that give access to Pro/ENGINEER user and manufacturing parameters.

Chapter 15, Geometric Tolerancing—Describes functions that read, modify, and create Geometric Tolerances (gtols) in a solid or drawing.

Chapter 16, Principles of Feature Creation—Describes basic principles of programmatic feature creation, applicable to all feature types that can be created under the current version.

Chapter 17, Creating Datum Features—Describes the element tree structure required to create datum planes, points, axes, and coordinate system features.
Chapter 18, Creating Datum Curves—Describes how to create, redefine, and access data for datum curve features using Pro/TOOLKIT.

Chapter 19, Creating Chamfer Features—Describes the element tree structure required to create a chamfer feature.

Chapter 20, Creating Hole Features—Describes how to use the include files ProHole.h so that you can create the hole features programmatically.

Chapter 21, Creating Patterns—Describes the element tree structure required to create a pattern feature.

Chapter 22, Assembling Components—Describes how to assemble components by feature creation.

Chapter 23, Sections—Describes programmatic creation sections using Pro/TOOLKIT.

Chapter 24, Creating Sketched Features—Describes the functions that enable you to create and manipulate sketched features.

Chapter 25, Creating Extruded and Revolved Features—describes how to use the include files ProExtrude.h, and ProRevolve.h so that you can create extruded and revolved features programmatically.

Chapter 26, Creating Swept Features—Describes the Pro/TOOLKIT functions that enable you to create swept features.

Chapter 27, Foreign Datum Surfaces and Curves—Describes the functions that enable you to create foreign datum curves and surfaces.

Chapter 28, Tweak Surface Replacement—Describes the Pro/TOOLKIT functions that enable you to create a tweak surface replacement feature.

Chapter 29, External Analyses and Analysis Features—Describes the functions that enable you to create analysis and analysis feature objects in a Pro/ENGINEER solid.

Chapter 30, Manufacturing—Describes how to use Pro/TOOLKIT functions for manufacturing operations.

Chapter 31, External Tool Database Access—Describes using Pro/TOOLKIT to access and query external, third-party tool databases.
Chapter 32, Auxiliary and Custom NC Sequences, and CL Commands—Describes the Pro/TOOLKIT functions that enable you to create auxiliary and custom NC sequences and CL commands.

Chapter 33, Process Planning—Describes the functions that enable you to use Pro/TOOLKIT for process planning.

Chapter 34, Family Tables—Describes the functions that enable you to manipulate the family table for an object.

Chapter 35, External Data—Describes how to store and retrieve external data.

Chapter 36, Notify—Describes how your Pro/TOOLKIT application can trap specified Pro/ENGINEER event types and arrange for your own function to be called before or after each such an event.

Chapter 37, Welds—Describes modeling welds in assemblies, generating report tables about weld parameters, and showing welding symbols in assembly drawings.

Chapter 38, Simplified Representations—Describes the Pro/TOOLKIT functions that enable you to access all the simplified representation functionality of Pro/ENGINEER.

Chapter 39, Drawings—Describes the functions that deal with drawings.

Chapter 40, Cross Sections—Describes the Pro/TOOLKIT functions that enable you to access, modify, and delete cross sections, and to create planar cross sections.

Chapter 41, Customized Plot Driver—Describes the customized plot driver functions supported by Pro/TOOLKIT.

Chapter 42, Diagrams and Cabling—Describes the Pro/TOOLKIT diagram and cabling functions.

Chapter 43, Finite Element Modeling (FEM)—Describes the functions that give you access to data generated by the Pro/MESHTM module of Pro/ENGINEER.

Chapter 44, External Objects—Describes the functions that enable you to create and manipulate external objects.

Chapter 45, Design Manager—Describes programmatic use of Design Manager functionality.

Chapter 46, Animation—Describes the Pro/TOOLKIT functions that enable you to create animation frames and movies.
Chapter 47, Access to Pro/INTRALINK—Describes the functions that perform design management.

Chapter 48, Piping—Describes Pro/TOOLKIT support for Pro/Piping features.

Chapter 49, Utilities—Describes the Pro/TOOLKIT utility functions.

Chapter 50, Interface—Describes various methods of importing and exporting files in Pro/TOOLKIT.

Chapter 51, Importing Features—Describes how to create import features in Pro/ENGINEER using Pro/TOOLKIT.

Chapter 52, Asynchronous Mode—Describes how to use Pro/TOOLKIT in Asynchronous Mode.

Chapter 53, Task Based Application Libraries—Describes how applications created using the different Pro/ENGINEER API products are interoperable.

Chapter 54, Help—Describes use of Pro/TOOLKIT Help files.

This manual also contains the following appendixes:

Appendix A, Summary of Technical Changes—Provides a summary of technical changes for this release of Pro/TOOLKIT.

Appendix B, Pro/TOOLKIT Registry File—Describes the Registry file used by Pro/Toolkit.

Appendix C, Debugging Pro/TOOLKIT Applications—Describes how to debug a Pro/TOOLKIT application.

Appendix D, Pro/DEVELOP to Pro/TOOLKIT Function Mapping—Describes which Pro/TOOLKIT functions supersede which Pro/Develop functions.

Appendix E, Pro/TOOLKIT Objects—Describes the Pro/TOOLKIT objects.

Appendix F, Geometry Traversal—Illustrates the relationships between faces, contours, and edges.

Appendix G, Geometry Representations—Describes the geometry representations of the data structures defined in ProGeomitem.h.

Appendix H, ptk_revtool Utility—Describes the purpose and use of the ptk_revtool utility.

Glossary—Provides a list of words that have meanings specific to Pro/TOOLKIT.

Index—Index of the Pro/TOOLKIT Users’ Guide
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.

<table>
<thead>
<tr>
<th>Convention</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Pro/ENGINEER-type menu name (for example, PART).</td>
</tr>
<tr>
<td>Boldface</td>
<td>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.</td>
</tr>
<tr>
<td>Monospace (Courier)</td>
<td>Code samples appear in courier font.</td>
</tr>
<tr>
<td>SMALLCAPS</td>
<td>Key names appear in smallcaps (for example, ENTER).</td>
</tr>
<tr>
<td>Emphasis</td>
<td>Important information appears in italics. Italic font also indicates file names and function arguments.</td>
</tr>
<tr>
<td>Choose</td>
<td>Highlight a menu option by placing the arrow cursor on the option and pressing the left mouse button.</td>
</tr>
<tr>
<td>Select</td>
<td>A synonym for “choose” as above, Select also describes the actions of selecting elements on a model and checking boxes.</td>
</tr>
<tr>
<td>Element</td>
<td>An element describes redefinable characteristics of a feature in a model.</td>
</tr>
<tr>
<td>Mode</td>
<td>An environment in Pro/ENGINEER in which you can perform a group of closely related functions (Drawing, for example).</td>
</tr>
<tr>
<td>Model</td>
<td>An assembly, part, drawing, format, layout, case study, sketch, and so on.</td>
</tr>
<tr>
<td>Option</td>
<td>An item in a menu or an entry in a configuration file or a setup file.</td>
</tr>
<tr>
<td>Solid</td>
<td>A part or an assembly.</td>
</tr>
</tbody>
</table>
• Important information that should not be overlooked appears in notes like this.
• All references to mouse clicks assume use of a right-handed mouse.
This chapter describes fundamental Pro/TOOLKIT concepts and functions.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction to Pro/TOOLKIT</td>
<td>1 - 2</td>
</tr>
<tr>
<td>Online Documentation — Pro/TOOLKIT APIWizard</td>
<td>1 - 2</td>
</tr>
<tr>
<td>The APIWizard Interface</td>
<td>1 - 7</td>
</tr>
<tr>
<td>APIWizard Search Feature (Find)</td>
<td>1 - 11</td>
</tr>
<tr>
<td>Pro/TOOLKIT Style</td>
<td>1 - 17</td>
</tr>
<tr>
<td>Installing Pro/TOOLKIT</td>
<td>1 - 20</td>
</tr>
<tr>
<td>Developing a Pro/TOOLKIT Application</td>
<td>1 - 28</td>
</tr>
<tr>
<td>User-Supplied Main</td>
<td>1 - 45</td>
</tr>
<tr>
<td>Asynchronous Mode</td>
<td>1 - 46</td>
</tr>
<tr>
<td>Pro/TOOLKIT Techniques</td>
<td>1 - 46</td>
</tr>
<tr>
<td>Visit Functions</td>
<td>1 - 56</td>
</tr>
<tr>
<td>Wide Strings</td>
<td>1 - 60</td>
</tr>
<tr>
<td>Wide String Functions</td>
<td>1 - 62</td>
</tr>
</tbody>
</table>
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:

\(<\text{proe_loadpoint}\>/\text{protoolkit/protkdoc}\)

To load the APIWizard manually, copy all files from

\(<\text{proe_loadpoint}\>/\text{protoolkit/protkdoc}\)
to your target directory.

To Run the APIWizard

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

\(<\text{proe_loadpoint}\>/\text{protoolkit/protkdoc/index.html}\)

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.
- SGI hardware platform users must install the Swing class. For more information, refer to the section on SGI Hardware Platforms. 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:
   
   \[\text{system\_disk}/Program Files/Netscape/Communicator/Program/java/classes\]

   **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.

SGI Hardware Platforms

SGI hardware platform users must download the Java Foundation Class (Swing) archive and install it in their CLASSPATH as described in Loading the Swing Class Library. On SGI platforms, if Netscape temporarily downloads the Swing archive and then starts the APIWizard, the following exception will be thrown, even though the class `javax/swing/text/MutableAttributeSet` exists in the downloaded archive.

```
java.lang.ClassNotFoundException: javax/swing/text/MutableAttributeSet
```

This exception is not thrown when the Swing archive is properly installed on the user's machine.

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 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 either the Pro/TOOLKIT Objects or Pro/TOOLKIT Users' Guide menu items.

In Objects mode, this frame displays an alphabetical list of the Pro/TOOLKIT objects. It can also display 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 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 next figure shows the APIWizard interface layout.
Navigating the Topic/Object Selection Tree

Use the Topic/Object Selection frame to access all Pro/TOOLKIT APIWizard online documentation data for objects, functions, or the Pro/TOOLKIT User’s 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 Objects or Pro/TOOLKIT User’s Guide at the top of the Topic/Object Selection frame. The APIWizard displays the tree structure for the objects or User’s Guide in collapsed form. The Switch icon to the far left of a node (that is, an object or chapter name) signifies that this node contains subnodes. (If a node has no Switch icon, it has no subnodes.) Selecting the Switch icon (or double-clicking on the node text) toggles 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 with the Pro/TOOLKIT APIWizard

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

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

Figure 1-2: 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 double-clicking the object name. The APIWizard then displays functions for that object as shown the preceding figure Pro/TOOLKIT object and function display.

**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 Selection frame. In this mode, the APIWizard displays the User’s Guide section headings.

Click Switch icon next to the section you wish to view by clicking the Switch icon next to the desired section name or double-clicking the section name text. The APIWizard then displays a tree of subsections under the selected section. The text for the selected section (or subsection) appears in the Display frame. Click 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.
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 Selection frame to display the APIWizard **Search** dialog box (see the following figure on the APIWizard Search dialog box).

**Figure 1-4: APIWizard Search Dialog Box**

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

- **Enter Search String or Strings**
  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 **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.
ProFeatureDelete

Description
Deletes the specified features from the solid.

Synopsis
#include <ProFeature.h>

ProError ProFeatureDelete
ProSolid solid
   "(In)
The part or assembly to which the features belong
/
int *feat_ids
   "(In)
An array of feature identifiers to be deleted
/
int feat_count
   "(In)
The size of the feat_ids array
/
ProFeatureDeleteOptions *delete_opts
   "(In)
An array of options for deleting features
/
int num_opts
   "(In)
The size of the delete_opts array
/
}

Returns
PRO_TK_NO_ERROR       The function successfully deleted the features
PRO_TK_BAD_INPUTS    One or more of the arguments are invalid
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 renames 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.

<table>
<thead>
<tr>
<th>Action Verb</th>
<th>Type of Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get</td>
<td>Read information directly from the Pro/ENGINEER database.</td>
</tr>
<tr>
<td>Eval</td>
<td>Provide the result of a simple calculation.</td>
</tr>
<tr>
<td>Compute</td>
<td>Provide the result of a computation that typically involves numerical analysis of the model geometry.</td>
</tr>
</tbody>
</table>

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 Pro/SETUP, one of the optional components is “API Toolkits”. This includes Pro/TOOLKIT, Pro/Web.Link, and J-Link.

If you select Pro/TOOLKIT, it is installed automatically under the load point 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 since Revision 18.
- 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 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 Pro/SETUP. For a description of using Pro/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).

To help you make this test, the Pro/TOOLKIT loadpoint includes the source of a simple application designed specifically for this purpose. 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, sgi_elf2 or 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:

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

with:

```c
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:

```bash
make -f make_install
```

This creates the Pro/TOOLKIT executable file called `pt_install_test`.

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

```bash
nmake -f make_install
```

This creates a file called `pt_install_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 Sample Registry Files for syntax requirements for this file. The `protk.dat` file should contain the following lines:

```plaintext
name install_test
exec_file pt_install_test
text_dir <TK_LOADPOINT>/protk_appls/pt_install_test
revision Wildfire
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_install_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_install_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 enter the following line instead of the line “make -f make_install”:

```
make -f make_install dll
```

This creates a file called `pt_install_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_install_test.dll
text_dir <TK_LOADPOINT>/protk_appls/pt_install_test
revision Wildfire
end
```

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

On Windows systems, use the Windows directory syntax.

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 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.
Compiling and Linking a Pro/TOOLKIT Application

Function Introduced

- user_initialize()

This section describes compiling and linking Pro/TOOLKIT applications. The topics are as follows:

- Makefiles
- Registering a Pro/TOOLKIT Application

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

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.

When compiling a Pro/TOOLKIT application on SUN systems using SUN C++ Version 5, you must use the compiler flag -compat=4.

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 in the following locations, in this order:

1. A file called `protk.dat` or `prodev.dat` in the current directory

2. A file named in a "PROTKDAT", "PRODEVDAT", or "TOOLKIT_REGISTRY_FILE" statement in the Pro/ENGINEER configuration file
3. A file called `protk.dat` or `prodev.dat` in the directory <Pro/ENGINEER>/<MACHINE>/text/<LANGUAGE>

4. A file called `protk.dat` or `prodev.dat` in the directory <Pro/ENGINEER>/text

In the last two options, the variables are as follows:

- `<Pro/ENGINEER>`—The Pro/ENGINEER loadpoint (not the Pro/TOOLKIT loadpoint)
- `<MACHINE>`—The machine-specific subdirectory (such as sgi_elf2 or i486_nt)
- `<LANGUAGE>`—The name of the Pro/ENGINEER language the Pro/TOOLKIT application will be used with (such as “usascii” (English), “german”, or “japanese”)

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 that option 1 is used normally during development, because the Pro/TOOLKIT application is seen only if you start Pro/ENGINEER from the specific directory that contains `protk.dat`.

Option 2 or 4 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.

Option 3 enables you to have a different registry file for each platform, and for each Pro/ENGINEER language. However, you can also support different platforms and languages with a single registry file by referencing environment variables in its contents.

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
revision Wildfire
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 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 directory that contains the menu and message files used by the Pro/TOOLKIT application.

- **revision**—Specifies the version of Pro/TOOLKIT against which you built the application.

- **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

### 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.

- **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 onwards 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 sun4_solaris running with a sun4_solaris_64 installation of Pro/ENGINEER. This is supported beginning in Pro/ENGINEER Wildfire 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 hpux_pa64 machine running with an hpux11_pa32 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:**

```plaintext
startup spawn
```

**DLL mode:**

```plaintext
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:

```plaintext
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 that 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

If you have a bug in your Pro/TOOLKIT application, the first test should be to see if it shows up in both multiprocess mode and DLL mode.
It is quicker to use a debugger on your application in multiprocess mode, because you need to load only the relatively small Pro/TOOLKIT application executable. In DLL mode, you must load the entire Pro/ENGINEER executable into the debugger. In both modes, however, it is necessary to take special steps to ensure that the process you are debugging is started correctly.

In multiprocess mode, the Pro/TOOLKIT application is designed to be spawned by Pro/ENGINEER with arguments that tell it how to connect to Pro/ENGINEER. Therefore, if you start the Pro/TOOLKIT executable from within the debugger, it will not behave correctly. The simplest solution is to start Pro/ENGINEER in the usual way, use operating system commands to find the process identifier of the Pro/TOOLKIT application, and then tell the debugger to connect to that process.

If you have a bug in your Pro/TOOLKIT application that appears when using DLL mode only, you need to use the debugger in DLL mode using the same technique. This is necessary because the Pro/ENGINEER executable is normally started by a script—the Pro/ENGINEER key whose name you type to start Pro/ENGINEER. One of the jobs of that script is to access the licensing information for a particular Pro/ENGINEER serial number. If you attempt to run the Pro/ENGINEER executable directly, it will fail and report that it cannot run without an authorized key. In effect, you have tried to bypass the Pro/ENGINEER license. The simplest solution is to run Pro/ENGINEER in the usual way, then find the process identifier of the Pro/ENGINEER executable. For this, you need to know the name of the Pro/ENGINEER executable, which is <Pro/ENGINEER>/<MACHINE>/obj/pro. In this directory specification, <Pro/ENGINEER> is the name of the Pro/ENGINEER loadpoint, and <MACHINE> is the name of the platform type (such as sgi_elf2 or i486_nt).

The Pro/ENGINEER executable does not contain debugging information for the contents of the Pro/TOOLKIT functions. However, if you recompile your own Pro/TOOLKIT application code with the appropriate debug option (usually “-g”), you can debug your own code at the source level.

For a full description of how to do this on UNIX, Windows NT, and Windows 95 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 application without having Pro/TOOLKIT as an option.

To unlock your application, enter the following command:

```
<Pro/ENGINEER>/bin/protk_unlock filename
```

In this example, `filename` is the full path to the Pro/TOOLKIT application.

**Notes:**
- Once you have unlocked the executable, you can distribute your application program to Pro/ENGINEER users in accordance with the license agreement.
- For NT systems, make sure to include the `.bat` extension with the file name.

Unlock Messages

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

<table>
<thead>
<tr>
<th>Message</th>
<th>Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usage: protk_unlock &lt;executable_file&gt; ex: protk_unlock /disk1/app1/&lt;machine&gt;/obj/pt_test</td>
<td>Wrong number of arguments supplied.</td>
</tr>
<tr>
<td>Error: Licenses don't contain a Pro/TOOLKIT option code.</td>
<td>A requirement for unlocking a Pro/TOOLKIT application.</td>
</tr>
<tr>
<td>ERROR: No READ access to pt_install_test. ERROR: No WRITE access to pt_install_test.</td>
<td>You do not have READ/WRITE access for the executable.</td>
</tr>
<tr>
<td>The time...about 15 minutes. Please wait a moment...</td>
<td>The application is unlocked successfully.</td>
</tr>
<tr>
<td>Successfully unlocked the application./sgi_elf2/obj/pt_install_test</td>
<td></td>
</tr>
</tbody>
</table>

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 ‘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.

The Core of a Pro/TOOLKIT Application

Functions introduced:

- user_initialize()
- 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. All input and output arguments to this function are optional and do not need to be in the function signature. Refer to section the section user_initialize() Arguments for details on using this function.

Note: Applications must contain at least one Pro/TOOLKIT function call in order to be registered with Pro/ENGINEER through this function.

The function user_initialize() is called after the Pro/ENGINEER application has been initialized and the Graphics Window has been created. user_initialize() should contain any initializations that your Pro/TOOLKIT application needs, including any modification of Pro/ENGINEER menus (such as adding new buttons).

The user_initialize() function is called with a number of optional arguments that can add to your function definition. 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.

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.

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.

```c
#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

Function Description Continued

- user_initialize()

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.
- char* build The build number of the Pro/ENGINEER being used.
The output argument is:

```c
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:

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

Command-line arguments such as

```c
-g:no_graphics
```

are interpreted by Pro/ENGINEER but not passed on to
Pro/TOOLKIT.

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

Function introduced:

- `ProEngineerEnd()`
- `user_initialize()`

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:

```bash
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.

```c
/*================================================================*
FUNCTION: UserAddHoles
PURPOSE: Find the circular datum curves and replace them with holes.
/*================================================================*/
UserAddHoles (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.

/*---------------------------------------------*/
  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");
}

Pro/TOOLKIT User's Guide
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()*.

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.

```c
#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 '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:

```c
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:

```c
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:

```c
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 J-Link.
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 **ProNoteCreate()**, 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 **ProNoteCreate()**.
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 function called **UsrFeaturesCollect()** is such a utility. It allocates an empty, expandable array of feature handles, and then passes this array to the application data to **ProSolidFeatVisit()**. The visit function adds the handle to the visited feature to the array using **ProArrayObjectAdd()**.

The function **UsrFeatList()** is a simple test harness that shows how the utility would be used.
FUNCTION : UsrFeatList()
PURPOSE : Command to list features

```c
int UsrFeatList()
{
ProMdl model;
ProFeature *features;
int f, n_features, num;
ProFeattype ftype;
ProName name;
FILE *fp;

Get the current part
ProMdlCurrentGet(&model);

Use local utility to get an expandable array of active features
if(!UsrFeaturesCollect(model, &features))
    return(0);

fp = fopen("featlist.txt","w");

Get the number of features
ProArraySizeGet(features, &n_features);

Write the features and their types to a file, and display the file
for(f=0;f<n_features;f++)
{
    ProFeatureTypeGet(&features[f], &ftype);
    fprintf(fp, "Feature number %2d, id %2d, type %d\n", f+1,
        features[f].id, ftype);
}
fclose(fp);

ProStringToWstring(name, "featlist.txt");
ProInfoWindowDisplay(name, NULL, NULL);

return(1);
}
```
PURPOSE : Action function called when visiting features
\*====================================================================*/
ProError UsrFeatAction(
    ProFeature *feature,
    ProError filter_status,
    ProAppData data)
{
    int n_feats, max;
    ProBoolean visible;
    ProFeatStatus status;

    /*-----------------------------------------------*/
    /* If the feature is internal, skip it */
    ProFeatureVisibilityGet(feature, &visible);
    if(!visible)
        return(PRO_TK_NO_ERROR);

    /*-----------------------------------------------*/
    /* If the feature is not currently active, skip it. */
    ProFeatureStatusGet(feature, &status);
    if(status != PRO_FEAT_ACTIVE)
        return(PRO_TK_NO_ERROR);

    /*-----------------------------------------------*/
    /* Check whether the number is too large for an expandable array */
    ProArraySizeGet(*(ProArray*)data, &n_feats);
    ProArrayMaxCountGet(sizeof(ProFeature), &max);
    if(n_feats == max)
        return(PRO_TK_OUT_OF_MEMORY);

    /*-----------------------------------------------*/
    /* Add the feature to the array */
    ProArrayObjectAdd((ProArray*)data, -1, 1, feature);

    return(PRO_TK_NO_ERROR);
}

/*====================================================================*
FUNCTION : UsrFeaturesCollect()
PURPOSE : Utility for creating an expandable array of active features
\*====================================================================*/
int UsrFeaturesCollect(
    ProSolid solid,
    ProFeature **features)
{
    /*-----------------------------------------------*/
    Allocate the empty array

Visit all the features in the given solid

```c
if(ProSolidFeatVisit(solid, UsrFeatAction, NULL, features) != PRO_TK_NO_ERROR)
{
    ProArrayFree((ProArray*)features);
    return(0);
}
return(1);
```

/* END OF EXAMPLE */

**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:

```c
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:

```c
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:

```c
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.
- Any other value (including **PRO_TK_CONTINUE**)—Terminate the visits.

### 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.

```c
typedef struct surface_visit_data
{
    FILE     *fp;
    ProSolid part;
} AxisVisitData_t;

/*===============================================================*
FUNCTION:  UserDemoAxisAct()
PURPOSE :  Axis-visit action function that writes 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 identifier.

status = ProAxisIdGet (axis, &id);
if (status != PRO_TK_NO_ERROR)
    return (status);

/*---------------------------------------------------------------*/

Make a ProModelItem handle for the axis.

status = ProModelItemInit (part, id, PRO_AXIS, &modelitem);
if (status != PRO_TK_NO_ERROR)
    return (status);

/*---------------------------------------------------------------*/

Get the feature to which the axis belongs.

status = ProGeomItemFeatureGet (&modelitem, &feature);
if (status != PRO_TK_NO_ERROR)
    return (status);

/*---------------------------------------------------------------*/

Get the feature type.

status = ProFeatureTypeGet (&feature, &ftype);
if (status != PRO_TK_NO_ERROR)
    return (status);

/*---------------------------------------------------------------*/

If the type was not HOLE, skip it.

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 identifier.

fprintf (fp, "Axis %s belongs to hole feature %d\n", name, feature.id);

/*---------------------------------------------------------------*/

Highlight the owning hole.

ProSelectionAlloc (NULL, &feature, &selection);
ProSelectionHighlight (selection, PRO_COLOR_HIGHLITE);
ProSelectionFree (&selection);

return (PRO_TK_NO_ERROR);

/*===============================================================*/
FUNCTION: UserDemoHoleList()
PURPOSE: List the axes that belong to the 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 = fopen ("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);
    /*---------------------------------------------------------------*/
    Close the file
    /*---------------------------------------------------------------*/
    fclose (data.fp);
    return (PRO_TK_NO_ERROR);
}
```

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:

  ```c
  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);
  }
  ```

Wide String Functions

Functions introduced:

- **ProStringToWstring()**
- **ProWstringToString()**

  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()**. These functions are fully documented in the online browser.
If you want to provide your own utilities for manipulating wide strings, observe the following recommendations to ensure complete portability:

- Do not use names like `wstrcmp()`, because functions with these names are provided by some platforms.
- Do not make any assumptions about the definition of `wchar_t` and how a wide string is encoded. If you are in fact using ASCII encoding, the best policy is to convert all input wide strings to character strings using `ProWstringToString()`; perform the necessary actions, using standard C functions where appropriate; and convert the results back to wide strings using `ProStringToWstring()`.

If you are using KANJI, for example, you can assume that a wide string is terminated by a zero-valued `wchar_t`. You can then write utilities by directly accessing the contents of the `wchar_t` array (for example, using code copied from a good C programming manual).

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

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

```c
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);
}```
sprintf (filename, "%s.%s;%d", obj_name,
            obj_type, data.version);

return (PRO_TK_NO_ERROR);
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.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Selection Object</td>
<td>2 - 2</td>
</tr>
<tr>
<td>Interactive Selection</td>
<td>2 - 5</td>
</tr>
<tr>
<td>Highlighting</td>
<td>2 - 9</td>
</tr>
</tbody>
</table>
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()
• ProSelectionAsmcomppathGet()
• ProSelectionModelitemGet()
• ProSelectionUvParamGet()
• ProSelectionViewGet()
• ProSelectionPoint3dGet()
• ProSelectionDepthGet()
• ProSelectionVerify()
• ProSelectionWindowIdGet()
• ProSelectionUvParamSet()
• ProSelectionViewSet()
• ProSelectionPoint3dSet()
• ProSelectionDrawingGet()
• ProSelectionDwgtblcellIGet()
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.

<table>
<thead>
<tr>
<th>Pro/TOOLKIT Function</th>
<th>Pro/TOOLKIT Object</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProSelectionAsmcomppathGet()</td>
<td>ProAsmcomppath</td>
<td>Assembly component path</td>
</tr>
<tr>
<td>ProSelectionModelitemGet()</td>
<td>ProModelitem</td>
<td>Model item</td>
</tr>
<tr>
<td>ProSelectionPoint3dGet()</td>
<td>ProPoint3d</td>
<td>3-D point on the model item</td>
</tr>
<tr>
<td>ProSelectionUvParamSet()</td>
<td>ProUvParam</td>
<td>u and v, or t, of that point</td>
</tr>
<tr>
<td>ProSelectionDepthGet()</td>
<td>double</td>
<td>Selection depth</td>
</tr>
<tr>
<td>ProSelectionViewGet()</td>
<td>ProView</td>
<td>Drawing view in which the selection was made</td>
</tr>
<tr>
<td>ProSelectionWindowIdGet()</td>
<td>int</td>
<td>Window where a selection is done</td>
</tr>
<tr>
<td>ProSelectionDrawingGet()</td>
<td>ProDrawing</td>
<td>Drawing in which the selection was made</td>
</tr>
<tr>
<td>ProSelectionDwgtblcellGet()</td>
<td>several integers</td>
<td>Table segment, row, and column of a selected</td>
</tr>
<tr>
<td></td>
<td></td>
<td>drawing table cell</td>
</tr>
</tbody>
</table>

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 Assemblies 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.

  In Pro/ENGINEER Wildfire, the selection user interface has migrated to a new mechanism and the **Get Select()** menu is no longer displayed. 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.

  The output of this function is an array of **ProSelection** structures that describe the selected items. The previous section explains how to analyze the contents of a **ProSelection** object.

  **Note:** When using this function in a UI command, set the command priority for the UI command which invokes this function. Improper priority settings can cause unpredictable results. See also Normal priority actions in the 'Menus' chapter.
The synopsis of **ProSelect()** is as follows:

```c
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. */
                  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.
<table>
<thead>
<tr>
<th>Pro/ENGINEER Database Item</th>
<th>ProSelect() Option</th>
<th>ProType</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datum point</td>
<td>point</td>
<td>PRO_POINT</td>
</tr>
<tr>
<td>Datum axis</td>
<td>axis</td>
<td>PRO_AXIS</td>
</tr>
<tr>
<td>Datum plane</td>
<td>datum</td>
<td>PRO_SURFACE</td>
</tr>
<tr>
<td>Coordinate system datum</td>
<td>csys</td>
<td>PRO_CSYS</td>
</tr>
<tr>
<td>Feature</td>
<td>feature</td>
<td>PRO_FEATURE</td>
</tr>
<tr>
<td>Edge (solid or datum surface)</td>
<td>edge</td>
<td>PRO_EDGE</td>
</tr>
<tr>
<td>Vertex</td>
<td>edge_end</td>
<td>PRO_EDGE_START, or PRO_EDGE_END</td>
</tr>
<tr>
<td>Datum curve</td>
<td>curve</td>
<td>PRO_CURVE</td>
</tr>
<tr>
<td>Datum curve end</td>
<td>curve_end</td>
<td>PRO_CRV_START, or PRO_CRV_END</td>
</tr>
<tr>
<td>Edge (solid only)</td>
<td>sldedge</td>
<td>PRO_EDGE</td>
</tr>
<tr>
<td>Edge (datum surface only)</td>
<td>qltedge</td>
<td>PRO_EDGE</td>
</tr>
<tr>
<td>Surface (solid or quilt)</td>
<td>surface</td>
<td>PRO_SURFACE</td>
</tr>
<tr>
<td>Surface (solid)</td>
<td>sldface</td>
<td>PRO_SURFACE</td>
</tr>
<tr>
<td>Surface (datum surface)</td>
<td>qltface</td>
<td>PRO_SURFACE</td>
</tr>
<tr>
<td>Quilt</td>
<td>dtmqlt</td>
<td>PRO_QUILT</td>
</tr>
<tr>
<td>Dimension</td>
<td>dimension</td>
<td>PRO_DIMENSION</td>
</tr>
<tr>
<td>Geometric tolerance</td>
<td>gtol</td>
<td>PRO_GTOL</td>
</tr>
<tr>
<td>Detail symbol</td>
<td>dtl_symbol</td>
<td>PRO_SYMBOL_INSTANCE</td>
</tr>
<tr>
<td>Drawing table</td>
<td>dwg_table</td>
<td>PRO_DRAW_TABLE</td>
</tr>
<tr>
<td>Note</td>
<td>any_note</td>
<td>PRO_NOTE</td>
</tr>
<tr>
<td>Note</td>
<td>note_3d</td>
<td>PRO_NOTE</td>
</tr>
<tr>
<td>Drawing view</td>
<td>dwg_view</td>
<td>PRO_VIEW</td>
</tr>
<tr>
<td>Diagram fixed connector, fixed component, or parametric connector</td>
<td>dgm_obj</td>
<td>PRO_DIAGRAM_OBJECT</td>
</tr>
<tr>
<td>Diagram wire (not a cable)</td>
<td>dgm_non_cable_wire</td>
<td>PRO_DIAGRAM_OBJECT</td>
</tr>
</tbody>
</table>
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. 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: an expandable array of `ProSelection` structures, and the number of items in the array.

<table>
<thead>
<tr>
<th>Pro/ENGINEER Database Item</th>
<th>ProSelect() Option</th>
<th>ProType</th>
</tr>
</thead>
<tbody>
<tr>
<td>Draft entity</td>
<td>draft_ent</td>
<td>PRO_DRAFT_ENTITY</td>
</tr>
<tr>
<td>Part</td>
<td>part</td>
<td>PRO_PART</td>
</tr>
<tr>
<td>Part or subassembly</td>
<td>prt_or_asm</td>
<td>PRO_PART, or PRO_ASSEMBLY</td>
</tr>
<tr>
<td>External object</td>
<td>ext_obj</td>
<td>PRO_EXTOBJ</td>
</tr>
<tr>
<td>Reference dimension</td>
<td>ref_dim</td>
<td>PRO_REF_DIMENSION</td>
</tr>
<tr>
<td>Table cell</td>
<td>table_cell</td>
<td>Not Applicable</td>
</tr>
<tr>
<td>Composite Curve</td>
<td>comp_crv</td>
<td>PRO_CURVE</td>
</tr>
<tr>
<td>Assembly component model</td>
<td>component</td>
<td>PRO_PART, PRO_ASSEMBLY</td>
</tr>
<tr>
<td>Component feature</td>
<td>membfeat</td>
<td>PRO_FEATURE</td>
</tr>
</tbody>
</table>

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. 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: an expandable array of `ProSelection` structures, and the number of items in the array.
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.
This chapter describes the functions used to communicate with the user through the text message area, including keyboard entry.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writing a Message using a Popup Dialog</td>
<td>3 - 2</td>
</tr>
<tr>
<td>Writing a Message to the Message Window</td>
<td>3 - 4</td>
</tr>
<tr>
<td>Message Classification</td>
<td>3 - 7</td>
</tr>
<tr>
<td>Writing a Message to an Internal Buffer</td>
<td>3 - 9</td>
</tr>
<tr>
<td>Getting Keyboard Input</td>
<td>3 - 9</td>
</tr>
<tr>
<td>Using Default Values</td>
<td>3 - 9</td>
</tr>
</tbody>
</table>
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

```c
#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, 2, PRO_UI_MESSAGE_NO,
                              NULL, (void*)NULL);
```

Example 1: Displaying the UI Message Dialog

```c
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, 2, PRO_UI_MESSAGE_NO,
                              NULL, (void*)NULL);
```
PROCARRAYFREE((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_u gui.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;
}

/*====================================================================*\nFUNCTION: 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 \textit{w} is available for wide strings.
• You do not need the character constant (\textbackslash 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.

<table>
<thead>
<tr>
<th>Conversion Character</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{f}</td>
<td>Float (or double)</td>
</tr>
<tr>
<td>\textit{d}</td>
<td>Decimal integer</td>
</tr>
<tr>
<td>\textit{s}</td>
<td>Ordinary string (or type \textit{char[f]}))</td>
</tr>
<tr>
<td>\textit{w}</td>
<td>Wide character strings</td>
</tr>
<tr>
<td>\textit{e}</td>
<td>Exponential</td>
</tr>
<tr>
<td>\textit{g}</td>
<td>Either float or exponential, as appropriate</td>
</tr>
</tbody>
</table>

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.

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`.

```c
#include <ProToolkit.h>
#include <ProMessage.h>
#include <TestError.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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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 = fopen ("msg_test.dat", "w");
if (msg_test != NULL)
{
   err = ProMessageToBuffer (msg_line, msgfil,
      "USER Values entered", wstr, &d, &i2, &i1);
   ProWstringToString (line, msg_line);
   fprintf (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
#
#
This chapter describes all the functions provided by Pro/TOOLKIT to create and manipulate menus and menu buttons.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>4 - 2</td>
</tr>
<tr>
<td>Menu Bar Buttons and Menus</td>
<td>4 - 2</td>
</tr>
<tr>
<td>Designating Commands</td>
<td>4 - 20</td>
</tr>
<tr>
<td>Mode-Specific Buttons and Menus</td>
<td>4 - 25</td>
</tr>
<tr>
<td>Entering Pro/ENGINEER Commands</td>
<td>4 - 44</td>
</tr>
</tbody>
</table>
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.
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.

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 action id.

A way to determine a command id 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' '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:

  ```c
  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`:

    ```c
    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:

```c
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.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uiCmdPrioDefault</td>
<td>Normal priority actions</td>
</tr>
<tr>
<td></td>
<td>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 <code>ProSelect()</code> 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.</td>
</tr>
<tr>
<td>uiProeImmediate,</td>
<td>Levels of immediate priority. Actions of each level of priority dismiss actions with lower level priorities.</td>
</tr>
<tr>
<td>uiProe2ndImmediate,</td>
<td></td>
</tr>
<tr>
<td>and uiProe3rdImmediate</td>
<td></td>
</tr>
<tr>
<td>uiProeAsynch</td>
<td>Asynchronous priority. Actions with asynchronous priority are independent of all other actions.</td>
</tr>
</tbody>
</table>

- `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`:

```c
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:

```c
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. 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
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 \texttt{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 \texttt{ProMenubarmenuPushbuttonAdd()} does not need to be separate from the message file used by the function \texttt{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:

• \texttt{ProCmdOptionAdd()}

• \texttt{ProMenubarmenuChkbuttonValueGet()}

• \texttt{ProMenubarmenuChkbuttonValueSet()}

The function \texttt{ProCmdOptionAdd()} adds a command to Pro/ENGINEER.

The syntax of this function is as follows:

\begin{verbatim}
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 );
\end{verbatim}
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`:

  ```c
  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:

```c
ProError ProMenubarmenuChkbuttonAdd (  
    ProMenuItemName parent_menu,  
    ProMenuItemName check_button_name,  
    ProMenuItemLabel check_button_label,  
    ProMenuLineHelp 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 **ProMenubarmenuMenuAdd()**, 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:

```c
ProError ProMenubarmenuRadiogrpAdd
    (ProMenuItemName parent_menu,
     ProMenuItemName radio_group_name,
     int number_radio_group_items,
     ProMenuItemName *radio_group_items,
     ProMenuItemLabel *radio_group_labels,
     ProMenuLineHelp *one_line_helps,
     ProMenuItemName neighbor,
```

```c
`ProMenubarmenuRadiogrpAdd` (parent_menu, radio_group_name, number_radio_group_items, radio_group_items, radio_group_labels, one_line_helps, neighbor,)
```
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.

• **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().

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

In the call to ProMenubarMenuAdd(), if you set the Boolean flag to PRO_B_TRUE, the added menu will be displayed to the right of the neighbor. Note that you cannot set the neighbor argument for this function to NULL.
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:

```plaintext
[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]
```

```c
/*================================================================*
FUNCTION: TestAccessDefault()
PURPOSE:  Define the accessibility of menu buttons.
="/*******陪伴您的人生，每一天*\
static uiCmdAccessState TestAccessDefault (uiCmdAccessMode access_mode)
{
    return (ACCESS_AVAILABLE);
}
```
FUNCTION: MiscAction()

Generic action function

```c
int MiscAction()
{
    ProMessageDisplay (UserMsg, "USER %s", "Action function called.");
    return (0);
}
```

FUNCTION: user_initialize()

Pro/TOOLKIT standard initialize

```c
int user_initialize (...)
{
    .
    .
    .

    Message file.

    ProStringToWstring (UserMsg, "msg_ugfund.txt");
    ProMessageDisplay (UserMsg, "USER %s",
        "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);
```
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:

```c
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`:

  ```c
  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 as the other input. The valid formats for the icon file are the Pro/ENGINEER .BIF file or a .GIF file. The Pro/ENGINEER toolbar button is replaced with the icon image.

Toolbar commands which are not assigned an icon will display the button label.

You may also use this function to assign a small icon to a menubar menu button. The icon will appear to the left of the button label.

Designating the Command

Function introduced:

• ProCmdDesignate()

This function allows you to designate the command as available in the Screen Customization dialog of Pro/ENGINEER. After a Pro/TOOLKIT application has used the function ProCmdDesignate() on a command, an interactive user is allowed to drag the toolbar button that you associate with the command, on to the Pro/ENGINEER toolbar in Pro/ENGINEER. If this function is not called, the toolbar button will not be visible in the Screen Customization dialog of Pro/ENGINEER. Its syntax 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”. 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 data.
Figure 4-1: The Customize Screen With The Icons To be Designated

Figure 4-2: The Pro/ENGINEER Toolbar With The Designated Icons
Example 2: Designating a Command

This example code illustrates how to designate a command to be available for placement as a toolbar button.

```c
/*---------------------------------------------------------------------*
 Set up the entry point to the demos
 *---------------------------------------------------------------------*/
ProStringToWstring (wmsgfil, "utilities.txt");

err = ProCmdActionAdd("-UG README!",
 (uiCmdCmdActFn)UserREADMESetup,
 uiProe2ndImmediate, NULL,
 PRO_B_TRUE, PRO_B_TRUE, &cmd_id);
ERROR_CHECK("user_initialize","ProCmdActionAdd()",&err);

err = ProMenubarmenuPushButtonAdd(
   "Utilities", "-UG README!", "-UG README!",
   "Enter the README menu of the User Guide",
   "Utilities.psh_util_aux", PRO_B_TRUE, 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,
   
   "Enter the README menu of the User Guide",
   "UG README description",
   wmsgfil);
ERROR_CHECK("user_initialize","ProCmdDesignate()",&err);

-UsrGuide Main
-UsrGuide Main
#
# Enter the Main menu of the User Guide.
Enter the Main menu of the User Guide.
#
# UG Main description
This command displays the non-mode specific userguide examples
#
#
```
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 throughout 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 3: 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.

```plaintext
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 4: Adding Alternate Names and Help Text to a Button

This example code creates an alternate button name and Help text for the previous example.

```plaintext
Menu file "part.aux":
[Start of file on next line]
PART
<blank line>
<blank 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 5: Adding a Menu Button to a Pro/ENGINEER Menu; Example 6: Defining a New Menu that Closes Itself; and Example 7: Defining a New Menu the User Must Close.
Example 5: 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 Menus

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 `userInitialize()` 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 6: Defining a New Menu that Closes Itself**

This example code defines a new menu, **MYMENU**, that closes itself
using the function ProMenuDelete().

```
MYMENU
<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:

```c
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);
```
Example 7: Defining a New Menu the User Must Close

In the following example code, the user has to close MYMENU.

```c
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 8: Using a New Menu**

The following example code shows how to use the functions **ProMenuProcess**() and **ProMenuCreate**(). The example builds on the previous examples.

```c
int action, menuId;

ProMenuCreate (PROMENUTYPE_MAIN, "mymenu", &menuId);
ProMenuProcess ("", &action);
```

**Creating a Menu for Selecting a Single Value**

Function introduced:

- **ProMenuDeleteWithStatus**()

Functions described further:

- **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 9: 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 9: 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()`.

```c
/*----------------------------------------------------------------*
 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;
  /*-----------------------------------------------*/
  ProMenuFileRegister ("value", "value.mnu", &menuId);
  ProMenubuttonActionSet ("value", "Value 1", ProUserValueGet, NULL, EX3_VALUE1);
  ProMenubuttonActionSet ("value", "Value 2", ProUserValueGet, NULL, EX3_VALUE2);

  /---------------------------------------------------------------*/
  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);
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 10: 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.

```c
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 (ProMessageStringRead (w_answer, 4))
        return (0);
    /*----------------------------------------------------------------*/
    ProWstringToString (answer, w_answer);
    return (tolower (answer[0]) == 'n');
}
```

Pro/TOOLKIT User's Guide
Submenus

Function described further:

- **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().

**Entering Pro/ENGINEER Commands**

Functions introduced:

- ProMacroLoad()
- ProMacroExecute()
- ProMenuCommandPush()

The function ProMacroLoad() loads a macro string onto a stack of macros that are executed once control returns to Pro/ENGINEER. A Pro/TOOLKIT macro string is equivalent to a Pro/ENGINEER map key without the key sequence and the map key name. To generate a macro string, create a map key in Pro/ENGINEER. Refer to the Pro/ENGINEER online help for more information about creating a map key.

Copy the value of the generated map key option from the Tools > Options dialog box. An example value is as follows:

```plaintext
$F2 @MAPKEY_LABELtest;
~ Activate `main_dlg_cur` `ProCmdModelNew.file`;
~ Activate `new` `OK`;
```

The key sequence is $F2. The map key name is @MAPKEY_LABELtest. The remainder of the string following the first semicolon is the macro string that should be passed to ProMacroLoad().
In this case, it is as follows:

- Activate `main_dlg_cur` `ProCmdModelNew.file``;
- Activate `new````````OK``;

**Note:** Creating or editing the macro string manually is not supported, as map keys are not a supported scripting language. The syntax is not defined for users and it may not remain constant across different datecodes of Pro/ENGINEER.

**Execution Rules**

Consider the following rules pertaining to the execution of macros:

- In asynchronous mode, macros are executed as soon as they are loaded with `ProMacroLoad()`. Macros in asynchronous mode are run in the same order that they are saved.
- In synchronous mode, 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. `ProMacroExecute()` forces Pro/ENGINEER to clear the stack of macros previously assigned to it using the function `ProMacroLoad()`, before execution. 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.
A **ProStringToWstring**() call for defining the macro string must be followed by the following calls:

- **ProMacroLoad(macro wstring)** to load to macro
- **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 toggle the wireframe display of model, use the macro:
  
  ```
  ProStringToWstring ( macro_wstring, "~ Select `main_dlg_cur` `ProCmdEnvMdlDisp.mdisp` 1 `Wireframe``");
  ```

- To toggle the shaded display of model, use the macro:

  ```
  ProStringToWstring ( macro_wstring, "~ Select `main_dlg_cur` `ProCmdEnvMdlDisp.mdisp` 1 `Shading``");
  ```

- You can toggle 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 model tree:

```
ProStringToWstring ( macro_wstring, "~ Select
 `main_dlg_cur` `PHTLeft.ProExplorerTab` 1
 `PHTLeft.MdlTreeLay`" );
```
Note: It is possible that a certain macro cannot be executed because a command specified in the macro is currently inaccessible (gray) 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 hit ENTER. However, if you must select something with a mouse (such as selecting a sketching plane), the macro is interrupted and ignores all the remaining commands in the string.

This way, the user can create object-independent macros for long sequences of repeating choices (as long as the user does not have to select any geometry). For example, this macro creates an A-sized drawing:

```plaintext
ProMacroLoad (ProStringToWstring
(wstr,"#drawing;#create;#A;#set up;
#modify val;#view;#rpaint;#views;#done");
```

**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 executing 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 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.
User Interface Components

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>5 - 2</td>
</tr>
<tr>
<td>Dialog Components</td>
<td>5 - 2</td>
</tr>
<tr>
<td>Resource Files</td>
<td>5 - 11</td>
</tr>
<tr>
<td>The Dialog Viewer</td>
<td>5 - 20</td>
</tr>
<tr>
<td>Dialog Programming: Common Functions</td>
<td>5 - 21</td>
</tr>
<tr>
<td>Programming Dialog Components</td>
<td>5 - 26</td>
</tr>
</tbody>
</table>

5 - 1
Introduction

This chapter describes the User Interface (UI) Dialogs available in Pro/TOOLKIT for Pro/ENGINEER Wildfire. The UI dialogs allow Pro/TOOLKIT users to create dialogs with the same look and feel as those in Pro/ENGINEER. The functions do not allow Pro/TOOLKIT users to modify Pro/ENGINEER dialogs.

The UI dialogs provide UI components and attributes to help users limit the cost of implementing, testing, and documenting applications that use these dialogs. The UI components support most of the commonly Pro/ENGINEER UI choices, and the attributes supported allow users to control dialog look and feel within Pro/ENGINEER style guidelines.

The components and attributes supported by Pro/TOOLKIT are described fully in the sections that follow. Note that in this release they form a subset of the components and attributes used by Pro/ENGINEER.

The dialog component library is integral to Pro/ENGINEER; thus, the components can only be used while a Pro/ENGINEER session is active. The library can not be used to create user interfaces for applications that do not run with, or connect to, Pro/ENGINEER.

Dialog Components

The dialog components that Pro/TOOLKIT can access are all shown in the following figure. See for a example resource file.
Figure 5-1: All Components Dialog

[Image of the All Components dialog box with various UI components such as check boxes, input panel, label, list, option menu, progress bar, radio group, separator, slider, spin box, table, text area, and thumbwheel.]
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 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.

- **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.

Example 1: 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)
    (CheckButton CheckButton1)
    (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)
    (CheckButton CheckButton3)
    (Label Label7)
    (Label Label11)
    (ThumbWheel ThumbWheel1)
)
(ProgressBar ProgressBar1)
(SpinBox SpinBox1)
(Label Label14)
(Separator Separator1)
(Label Label13)
(Table Table1)
(Label Label17)

(Resources
(InputPanel1.InputType 2)
(Label1.Label "Input Panel")
(Label1.AttachLeft True)
(CheckButton1.Label "Check2")
(CheckButton1.Set True)
(CheckButton1.TopOffset 4)
(CheckButton1.BottomOffset 4)
(CheckButton1.LeftOffset 4)
(CheckButton1.RightOffset 4)
(Label12.Label "Check Buttons")
(Label12.AttachLeft True)
(Label12.TopOffset 4)
(Label12.BottomOffset 4)
(Label12.LeftOffset 4)
(Label12.RightOffset 4)
(Label13.Label "Label")
(Label13.AttachLeft True)
(Label14.Label "Label Text")
(List1.Names "n1"
"n2"
"n3"
"n4"
"n5")
(List1.Labels "Value 1"
"Value 2"
"Value 3"
"Value 4"
"Value 5")
(Label5.Label "List")
(Label5.AttachLeft True)
(OptionMenu1.Names "n1"
"n2"
"n3"
"n4"
"n5")
(OptionMenu1.Labels "Option 1"
"Option 2"
"Option 3"
"Option 4"
"Option 5")
(Label6.Label "Option Menu")
(Label6.AttachLeft True)
Dialogs with Menubars

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 5-2: Menubar Dialog

See Common Attributes 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.

Resource Files

The overall structure of a dialog is described in a text file called a resource file. When the Pro/TOOLKIT application wants to show a dialog to the Pro/ENGINEER user, it simply asks Pro/ENGINEER to load the dialog from the file. The first task for the Pro/TOOLKIT user who wants to display his own dialog is to write the resource file.

The resource file describes:

• Overall attributes of the dialog
• 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.

Many of the dialog and component attributes can also be read and modified programmatically with Pro/TOOLKIT functions.

The resource files used by Pro/ENGINEER dialogs can be found in the directory

{Pro/E loadpoint}/text/resource

Syntax of Resource Files

A resource file must be called dialog_name.res where dialog_name is the name of the dialog. 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. If the application uses language specific directories, such as "usascii" for English, the resource file must be located in text/<language>/resource. There must be one resource file for each language supported by the application. If the application supports only one language, it may be located in text/resource.

Note: Resource files are case insensitive.
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 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.

The two statements that follow the name of the dialog in the Dialog statement 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, or CascadeButton, have their own top-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 Common Attributes.
Example 2: 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.

Figure: Simple Dialog shows the resulting dialog.

(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
      )
    )
  )
)
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.

Example 3: 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)
 )
)```
Example 4: 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)
    (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
    )
  )
)

Figure 5-5: Layout Dialog

Common Attributes

The table below shows attributes which are valid for all dialog components. Where the final column shows a Pro/TOOLKIT function, this function allows the attribute to be accessed programmatically; if the column is blank, the attribute must be set in the resource file and cannot be read or modified by the application.

<table>
<thead>
<tr>
<th>Attribute Names</th>
<th>Values</th>
<th>Pro/TOOLKIT Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>.AttachLeft</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>.AttachRight</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>.AttachTop</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>.AttachBottom</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
</tbody>
</table>
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.

Note: The default value for these attributes changes with the components.

Generally, keep components completely unattached so that they do not resize; the size of input panels, text areas, and so on, can be fully specified using attributes specific to those components.

Note: The only common exception is that if you have a layout containing a column of components to the right of a column of labels labelling the components—a common form of dialog—the labels should all have .AttachLeft TRUE to ensure that the labels align with each other.

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.
**Note:** The Attach and Offset attributes do not apply to components which belong to a MenuPane, as a MenuPane may not be resized.

The .Mapped attribute identifies whether an object takes up space when invisible. Set this to TRUE to have the item take up the same space it normally would if it were visible. Set it to FALSE to not occupy any space in the dialog while invisible.

The .Resizeable attribute identifies whether an object is allowed to resize based on changing 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.

**Note:** The dialog to which the component belongs must also be resizeable to allow the individual components to resize.

The .HelpText attribute is the text that appears in a yellow box if you leave the cursor over that component for a few moments. The value must be enclosed in double quotes. If the help text contains more than one line, indicate each new line by a new line in the resource file.

The .Sensitive attribute describes whether the component accepts user input, or is grayed out. A disabled component occupies the same location as it does when it is enabled.

**Note:** An insensitive component does not trigger any callback actions, while a readonly component (inputpanel, textarea) still is affected by focus and other action callbacks.

The .Visible attribute describes whether the component appears in the dialog. A hidden component still exists in the dialog but cannot be seen. The space occupied is determined by .Mapped attribute.

### Example 5: Resource File with Offsets, Attachments, and Help Text

This example shows a section of resource file which specifies offsets, attachments, and help text is shown below. This is an extended version of the previous example.

```plaintext
(Resources
(FeatNameLabel.Label            "Feature name")
(FeatNameLabel.AttachLeft       True)
(FeatNameLabel.TopOffset        4)
(FeatNameLabel.BottomOffset     4)
(FeatNameLabel.LeftOffset       4)
(FeatNameLabel.RightOffset      4)
(FeatNamePanel.HelpText         "Enter the Feature name here.")
(FeatNamePanel.TopOffset        4)
```
The .Bitmap attribute is the name of a file which contains a bitmap description of an image which should be applied to the component. The file can be one of two types:

- A PTC proprietary bitmap format file used for Pro/ENGINEER's dialog components. The files have the extension ".bif".
- A GIF format file.

If you supply only the root of the filename, the .bif format will be assumed.

Use the bif format when you want to make use of a bitmap file used in a Pro/ENGINEER dialog. Pro/ENGINEER's bif files can be found in the same directory as its resource files

<Pro/ENGINEER loadpoint>/text/usascii/resource

When you see a bitmap image on a Pro/ENGINEER dialog, and you want to use the same image in your own dialog, first find out the name of Pro/ENGINEER's resource file for that dialog, by searching the contents of the resource directory. Then you should be able to identify the bitmap file itself by reading the resource file.

You do not have to copy the bitmap file to a new location, or provide it to customers along with your Pro/TOOLKIT application, as the resource directory is already in Pro/ENGINEER's search path.

The Dialog Viewer

To enable you to test your resource files without having to create a Pro/TOOLKIT application and run Pro/ENGINEER, Pro/TOOLKIT provides a utility which just reads in a resource file and displays the dialog it describes. It takes the resource file name as a single argument: prodialog_view

Note: The executable, prodialog_view, is installed in protoolkit/<machine_type>/obj. Make sure you set the environment variable PRO_DIRECTORY to point to the Pro/ENGINEER loadpoint before running this utility.
Dialog Programming: Common Functions

Functions introduced

- ProUIDialogCreate()
- ProUIDialogActivate()
- ProUIDialogExit()
- ProUIDialogDestroy()
- ProUIDialogCloseActionSet()
- ProUIDialogActivateActionSet()
- ProUIDialogDefaultbuttonSet()
- ProUIDialogTitleSet()

Dialog Attributes

This section describes common attributes and functions used when creating any dialog.

Dialog Attributes are the attributes that apply to the dialog itself. They are specified in the same way as component attributes, but without a component name.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Values</th>
<th>Pro/TOOLKIT Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>.DefaultButton</td>
<td>string</td>
<td>ProUIDialogDefaultbuttonSet()</td>
</tr>
<tr>
<td>.DialogStyle</td>
<td>2 (default), 6</td>
<td>—</td>
</tr>
<tr>
<td>.Label</td>
<td>string</td>
<td>ProUIDialogTitleSet()</td>
</tr>
<tr>
<td>.RememberPosition</td>
<td>TRUE(default), FALSE</td>
<td>—</td>
</tr>
<tr>
<td>.RememberSize</td>
<td>TRUE(default), FALSE</td>
<td>—</td>
</tr>
<tr>
<td>.Resizeable</td>
<td>TRUE, FALSE (default)</td>
<td>—</td>
</tr>
<tr>
<td>.StartLocation</td>
<td>integer 1-9</td>
<td>—</td>
</tr>
</tbody>
</table>

The Default Button is the name of the push button selected if the user hits the Enter key when the mouse is over the dialog.

Set the DialogStyle attribute to 2, except when displaying a progress bar in a dialog. When displaying a progress bar set to 6.
The Label is the text that appears in the header of the window in which the dialog is displayed.

The RememberSize and RememberPosition attributes control whether Pro/ENGINEER should store information about the dialog when it is destroyed, and apply the size and position to the dialog again if it is shown again.

To prevent the user from resizing the dialog, set the Resizeable attribute to FALSE.

The StartLocation defines the position on the computer screen that a dialog will take up when it is first displayed. The meaning of the nine possible integer values are shown in the following table:

<table>
<thead>
<tr>
<th>Value</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Top Left</td>
</tr>
<tr>
<td>2</td>
<td>Top Center</td>
</tr>
<tr>
<td>3</td>
<td>Top Right</td>
</tr>
<tr>
<td>4</td>
<td>Middle Left</td>
</tr>
<tr>
<td>5</td>
<td>Middle Center</td>
</tr>
<tr>
<td>6</td>
<td>Middle Right</td>
</tr>
<tr>
<td>7</td>
<td>Bottom Left</td>
</tr>
<tr>
<td>8</td>
<td>Bottom Center</td>
</tr>
<tr>
<td>9</td>
<td>Bottom Right</td>
</tr>
</tbody>
</table>

It is usual to make dialogs appear in the middle center location unless it is necessary for the user to view the graphics window while interacting with the dialog.

The function *ProUIDialogCreate()* loads a dialog from a resource file into memory, but does not display it. *ProUIDialogCreate()* takes two names, the first argument is the name that is given to the instance of the dialog. The second name is the name of the resource file that contains the definition of the dialog. This allows you to have one dialog definition, but multiple instances of the dialog on the screen at the same time. There is no requirement for the two names to be the same.

The function *ProUIDialogActivate()* displays the dialog, and controls the user interaction with it. It returns when a call to *ProUIDialogExit()* is made from a one of the callbacks assigned to the dialog components.

Do these two things after the call to *ProUIDialogCreate()* and before the call to *ProUIDialogActivate()*:

- Assign the action callbacks to be called when the user selects the dialog components. This is done using functions such as *ProUICheckbuttonActivateActionSet()* which are described fully in the following sections.
• Modify any dialog or component attributes which are not set in the resource file, or need to be modified. This is done with functions such as ProUILabelTextSet() (see section Programming with Labels).

Following the return from ProUIDialogActivate() you should call ProUIDialogDestroy(), which removes the dialog from the screen, and clears it from memory.

The function ProUIDialogActivateActionSet() sets the function to be called when the dialog becomes the current foreground window. The input arguments are the name of the dialog, the pointer to the callback function, and a pointer to user data that is passed to the callback.

The function ProUIDialogCloseActionSet() can also be used to set a callback to be called when the dialog is closed. This callback will be called after you call ProUIDialogExit(), but it will also get called when the user attempts to close the operating system window which contains the dialog. If you do not set such a callback, attempts to close the dialog in this way will fail. If you want the user to be able to close the dialog by closing its window, set this callback and call ProUIDialogExit() from within it.

The functions ProUIDialogDefaultbuttonSet() and ProUIDialogTitleSet() define the default button, that is, the button highlighted when the dialog is displayed. Press Return to “push” the default button. These functions provide the ability to set programmatically the dialog attributes defined in the resource file as .DefaultButton and .Label respectively. If your resource file already contains the values for these that you want to use, you don’t need to call the functions. You can also define the default button in the resource file.

**Example 6: 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.

```plaintext
(Dialog confirm
 (Components
   (Label Question)
   (PushButton OK)
   (PushButton Cancel)
 )
 (Resources
   (Question.Label "Dummy label")
   (Question.TopOffset 4)
   (Question.BottomOffset 4)
)```
Example 7: Source for Dialog with Text Question, OK and Cancel Buttons

This example shows the source code that uses this dialog.

```c
#define OK     1
#define CANCEL 0
/*-----------------------------------------------------------------*
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

```c
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.**

```c
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);
}
```
Programming Dialog Components

The following sections introduce attributes and functions that are specific to the various types of dialog components. The functions fall into these categories:

- **Action Set functions**, which define callback functions to be called when the user interacts with the dialog component.
- **Attribute functions**, which can be used to set the values of any component attributes. By modifying the attributes of a dialog and its components programmatically you can sometimes re-use a dialog in different situations instead of defining a new one each time.
- **Functions to read and modify the state of dialog components**. These are used to read the information the user has provided using the dialog, to set default values, and to arrange linkages between components - to set the value of one component automatically in response to the user changing another component.
- **Functions to control the accessibility of a component**, by enabling or disabling it, or by showing or hiding it in the dialog. It is friendly to the user to disable a component, if you choose another component that makes the chosen component unnecessary.

Nearly all your dialogues should have at least an OK and a Cancel button, whose callbacks will close the dialog by calling `ProUIDialogExit()`. It is not essential to use callbacks for the other components on your dialog, as you can read all the information entered by the user from within the OK callback. Often however, there will need to be linkages between the various components, such that one component changes its value, or becomes enabled/disabled, in response to the user interacting with another component; in these cases you will need to use callback functions for all the components that have an immediate effect on other components.
Programming with CascadeButtons

This section describes how to program dialogues that use CascadeButton.

CascadeButton Attributes

This section describes attributes of CascadeButtons.

<table>
<thead>
<tr>
<th>Attribute Names</th>
<th>Values</th>
<th>Pro/TOOLKIT Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CascadeDirection</td>
<td>int</td>
<td>-</td>
</tr>
<tr>
<td>Label</td>
<td>string</td>
<td>ProUICascadebuttonTextSet()</td>
</tr>
</tbody>
</table>

The attribute CascadeDirection determines the position of the cascading menu relative to the cascade button. It takes an integer value representing “Top Left” (1), “Top Right (3)”, Button left” (7), or “Bottom right” (9). The default is 9.

CascadeButton Functions

Functions introduced:

- ProUICascadebuttonTextSet()
- ProUICascadebuttonHelptextSet()
- ProUICascadebuttonEnable()
- ProUICascadebuttonDisable()
- ProUICascadebuttonIsEnabled()
- ProUICascadebuttonShow()
- ProUICascadebuttonHide()
- ProUICascadebuttonIsVisible()

The functions ProUICascadebuttonHelptextSet() and ProUICascadebuttonTextSet() provide the ability to set programmatically the CascadeButton attributes defined in the resource file as .Helptext and .Label respectively. If your resource file already contains the values for these that you want to use, you don’t need to call the functions.

The functions ProUICascadebuttonEnable() and ProUICascadebuttonDisable() control sensitivity of the cascadebutton. The function ProUICascadebuttonIsEnabled() determines whether the cascadebutton is sensitive to user input.
The functions **ProUICascadebuttonShow()** and **ProUICascadebuttonHide()** control the visibility of the cascade button. The function **ProUICascadebuttonIsVisible()** determines whether the cascade button is visible in the dialog.

### Programming with Check Buttons

The next sections describe programming with Check Buttons.

### CheckButton Attributes

The attribute label defines the text which appears immediately to the right of the check button.

The attribute Set defines the initial status of the check button.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Values</th>
<th>Pro/TOOLKIT Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>.Label</td>
<td>string</td>
<td>ProUICheckbuttonTextSet()</td>
</tr>
<tr>
<td>.Set</td>
<td>TRUE, FALSE</td>
<td>ProUICheckbuttonSet(), ProUICheckbuttonUnset(), ProUICheckbuttonGetState()</td>
</tr>
</tbody>
</table>

### Check Button Functions

Functions introduced

- **ProUICheckbuttonActivateActionSet()**
- **ProUICheckbuttonEnable()**
- **ProUICheckbuttonDisable()**
- **ProUICheckbuttonSet()**
- **ProUICheckbuttonUnset()**
- **ProUICheckbuttonGetState()**
- **ProUICheckbuttonHelptextSet()**
- **ProUICheckbuttonTextSet()**
- **ProUICheckbuttonEnable()**
- **ProUICheckbuttonDISABLE()**
- **ProUICheckbuttonIsEnabled()**
- **ProUICheckbuttonShow()**
- **ProUICheckbuttonHide()**
- **ProUICheckbuttonIsVisible()**
The function \texttt{ProUICheckbuttonActivateActionSet()} sets the callback to be called when the user selects the checkbutton. The arguments are the names of the dialog and the checkbutton component, the pointer to the callback function, and a pointer to user data that will be passed to the callback.

The callback is defined as type \texttt{ProUIAction} which has this declaration:

\begin{verbatim}
typedef void (*ProUIAction)(
    char* dialog,
    char *component,
    ProAppData appdata);
\end{verbatim}

The functions \texttt{ProUICheckbuttonEnable()} and \texttt{ProUICheckbuttonDisable()} enable and disable the checkbutton. \texttt{ProUICheckbuttonSet()} and \texttt{ProUICheckbuttonUnset()} set its current state to TRUE and FALSE respectively. \texttt{ProUICheckbuttonGetState()} gets the current state of the checkbutton.

The functions \texttt{ProUICheckbuttonHelptextSet()} and \texttt{ProUICheckbuttonTextSet()} provide the ability to set programmatically the checkbutton attributes defined the resource file as .Helptext and .Label respectively. If your resource file already contains the values for these that you want to use, you don’t need to call the functions.

The functions \texttt{ProUICheckbuttonEnable()} and \texttt{ProUICheckbuttonDisable()} control the sensitivity of the checkbutton. The function \texttt{ProUICheckbuttonIsEnabled()} determines whether the checkbutton is sensitive to user input.

The functions \texttt{ProUICheckbuttonShow()} and \texttt{ProUICheckbuttonHide()} control the visibility of the checkbutton. The function \texttt{ProUICheckbuttonIsVisible()} determines whether the checkbutton is visible in the dialog.
Programming with InputPanels

The next sections describe programming with InputPanels.

InputPanel Attributes

This table shows values and functions for InputPanel attributes.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Values</th>
<th>Pro/TOOLKIT Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>.Columns</td>
<td>integer (default 16)</td>
<td>ProUIInputpanelColumnsSet()</td>
</tr>
<tr>
<td>.Digits</td>
<td>integer (default -1)</td>
<td>—</td>
</tr>
<tr>
<td>.DoubleFormat</td>
<td>string</td>
<td>—</td>
</tr>
<tr>
<td>.Editable</td>
<td>TRUE, FALSE</td>
<td>ProUIInputpanelEditable(), ProUIInputpanelReadOnly()</td>
</tr>
<tr>
<td>.InputType</td>
<td>Integer: One of the values of the ProUIInputtype enum in ProUI.h</td>
<td>ProUIInputpanelInputtypeSet()</td>
</tr>
<tr>
<td>.MaxDouble</td>
<td>double</td>
<td>—</td>
</tr>
<tr>
<td>.MaxInteger</td>
<td>integer</td>
<td>—</td>
</tr>
<tr>
<td>.MaxLen</td>
<td>integer (default 32)</td>
<td>—</td>
</tr>
<tr>
<td>.MinColumns</td>
<td>integer (default 4)</td>
<td>—</td>
</tr>
<tr>
<td>.MinDouble</td>
<td>double</td>
<td>—</td>
</tr>
<tr>
<td>.MinInteger</td>
<td>integer</td>
<td>—</td>
</tr>
<tr>
<td>.Password</td>
<td>TRUE (default), FALSE</td>
<td>—</td>
</tr>
<tr>
<td>.Value</td>
<td>wide string</td>
<td>ProUIInputpanelValueSet(), ProUIInputpanelValueGet()</td>
</tr>
</tbody>
</table>

The Columns attribute defines the initial width in characters of the input panel. If the input panel is attached both left and right, the width may be greater than this initial width depending upon the width of other components in the same grid column, and upon whether the user has resized the dialog.

The MinColumns attribute specifies the smallest number of columns the input panel can have if the grid column gets smaller when the user resizes the dialog.

The Editable attribute specifies whether the value can be edited by the user. If this is FALSE, the input panel value can be set only from the application.
The InputType attribute specifies what type of value is expected. The dialog accepts only input from the user that corresponds to the specified type. Some of the other Input Panel attributes apply only to certain InputType values. The values, and their dependent attributes are specified below:

- **String**—If the Password attribute is TRUE, the value entered by the user will be displayed using asterisks only. The MaxLen attribute specifies the maximum string length.

- **Widestring**—The MaxLen attribute specifies the maximum length of the string.

- **Integer**—The MaxInteger and MinInteger attributes specify the maximum and minimum values.

- **Double**—The Digits attribute specifies the number of digits following the decimal place. The default is -1 which results in six digits.

  The DoubleFormat attribute specifies the format in which the double must be displayed. The syntax is similar to the syntax for printing a double using the C library function printf(), except that an asterisk “*” is put in place of the number of decimals, as that is specified by a separate attribute. The default double format is “%.1f”, which specifies a “long float” (that is, a double), with a decimal point but no specified field width. Another example would be “%0.2e” which would result in exponent notation.

  The MaxDouble and MinDouble attributes specify the maximum and minimum values.

- **Fraction**—Specifies that input must in fraction format, for example 3/4, 7/8.

  The Value attribute is used to set the initial value of the input panel, irrespective of its format.
InputPanel Functions

Functions introduced:

- ProUIInputpanelActivateActionSet()
- ProUIInputpanelInputActionSet()
- ProUIInputpanelFocusinActionSet()
- ProUIInputpanelFocusoutActionSet()
- ProUIInputpanelHelpsetText()
- ProUIInputpanelColumnsSet()
- ProUIInputpanelInputtypeSet()
- ProUIInputpanelDoubleGet()
- ProUIInputpanelDoubleSet()
- ProUIInputpanelIntegerGet()
- ProUIInputpanelIntegerSet()
- ProUIInputpanelStringGet()
- ProUIInputpanelStringSet()
- ProUIInputpanelWidestringGet()
- ProUIInputpanelWidestringSet()
- ProUIInputpanelOrdinalGet()
- ProUIInputpanelOrdinalSet()
- ProUIInputpanelNumeratorGet()
- ProUIInputpanelNumeratorSet()
- ProUIInputpanelDenominatorGet()
- ProUIInputpanelDenominatorSet()
- ProUIInputpanelValueGet()
- ProUIInputpanelValueSet()
- ProUIInputpanelMinintegerSet()
- ProUIInputpanelMaxintegerSet()
- ProUIInputpanelMindoubleSet()
- ProUIInputpanelMADXdoubleSet()
- ProUIInputpanelDigitsSet()
- ProUIInputpanelEditable()
The function `ProUIInputpanelActivateActionSet()` sets the callback function to be called when the user presses Enter after typing a new value in the input panel.

The function `ProUIInputpanelInputActionSet()` sets the callback function to be called when the user changes the contents of the input panel. This function is called for every keypress.

The functions `ProUIInputpanelFocusinActionSet()` and `ProUIInputpanelFocusoutActionSet()` set the action functions to be called when user clicks on or off of the input panel component in the dialog.

The functions `ProUIInputpanelHelptextSet()`, `ProUIInputpanelColumnsSet()` and `ProUIInputpanelInputtypeSet()` provide the ability to set programmatically the input panel attributes defined in the resource file as .Helptext, .Columns and .Inputtype respectively.

The functions `ProUIInputpanelDoubleGet()`, `ProUIInputpanelDoubleSet()`, `ProUIInputpanelIntegerGet()`, `ProUIInputpanelIntegerSet()`, `ProUIInputpanelStringGet()`, `ProUIInputpanelStringSet()`, `ProUIInputpanelWidestringGet()`, `ProUIInputpanelWidestringSet()`, `ProUIInputpanelOrdinalGet()`, `ProUIInputpanelOrdinalSet()`, `ProUIInputpanelNumeratorGet()`, `ProUIInputpanelINumeratorSet()`, `ProUIInputpanelDenominatorGet()`, and `ProUIInputpanelDenominatorSet()` allow you to access the value contained in the input panel, depending on the type of value, as determined by the InputType attribute.

The `ProUIInputpanelValueGet()` and `ProUIInputpanelValueSet()` get and set the value of the input panel, in terms of a wide string.
The functions `ProUIInputpanelMinintegerSet()`, `ProUIInputpanelMaxintegerSet()`, `ProUIInputpanelMindoubleSet()`, `ProUIInputpanelMaxdoubleSet()`, and `ProUIInputpanelDigitsSet()` control the limitations and formatting assigned for the input panel value.

The functions `ProUIInputpanelEditable()` and `ProUIInputpanelReadOnly()` control whether the input panel can be modified by the user. If the input panel is read only, it’s background color is grey instead of white.

The functions `ProUIInputpanelEnable()` and `ProUIInputpanelDisable()` control whether the input panel is sensitive to user input. The function `ProUIInputpanelIsEnabled()` returns whether the input panel is sensitive or not.

The functions `ProUIInputpanelShow()` and `ProUIInputpanelHide()` control whether the input panel is visible in the dialog. The function `ProUIInputpanelIsVisible()` returns whether or not the input panel is visible.

### Programming with Labels

The next sections describe programming with Labels.

#### Label Attributes

The Columns attribute specifies how many columns the Label occupies. This only make sense if the value is greater than the number of characters in the Label attribute, in which case it controls the minimum width of the grid column containing that label.

The Label attribute is the text of the Label.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Values</th>
<th>Pro/TOOLKIT Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>.Columns</td>
<td>integer</td>
<td>—</td>
</tr>
<tr>
<td>.Label</td>
<td>wide string</td>
<td><code>ProUILabelTextSet()</code></td>
</tr>
</tbody>
</table>
Label Functions

Function introduced:

- ProUILabelHelptextSet()
- ProUILabelTextSet()
- ProUILabelEnable()
- ProUILabelDisable()
- ProUILabelIsEnabled()

The functions **ProUILabelHelptextSet()** and **ProUILabelTextSet()** provide the ability to set programmatically the label attributes defined in the resource file as .Helptext, .Label. Example 15: Source of Dialog with Slider and Linked InputPanel made use of **ProUILabelTextSet()** to modify the text on a dialog.

The functions **ProUILabelEnable()** and **ProUILabelDisable()** control whether the label is sensitive to user input. The function **ProUILabelIsEnabled()** returns whether or not the label is sensitive.

Programming with Layouts

The next sections describe programming with Layouts.

Layout Attributes

A layout has two main functions:

- A sublayout of a tab component, to contain a group of components that must be shown at one time
- To better control the relative position of components in a dialog by creating a local region with its own grid.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Values</th>
<th>Pro/TOOLKIT Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>.Decorated</td>
<td>TRUE, FALSE (default)</td>
<td>—</td>
</tr>
<tr>
<td>.Label</td>
<td>wide string</td>
<td>ProUILayoutTextSet()</td>
</tr>
</tbody>
</table>

In both these cases it would be usual to use an undecorated layout.

A third use of a layout is to visually separate groups of components to make the logical grouping clear to the user. In this case the layout should be Decorated.
The following figure shows a simple dialog with layout that contains the radio buttons, the input panel, and the labels on those components. This means that the two columns they occupy do not need to be of the same width as the two at the top level which contain the OK and Cancel buttons.

Figure 5-7: Undecorated Layout Dialog

![Undecorated Layout Dialog](image)

The following figure shows a dialog with the same structure, but with the layout set as Decorated.

Figure 5-8: Decorated Layout Dialog

![Decorated Layout Dialog](image)

The Label attribute has two purposes:

- Decorated layout—specifies the text to appear at the top of the layout.
- Layout belonging to decorated tab component—specifies the text in the tab handle. (See Programming with Tabs for a description of Tabs.)
Layout Functions

Function introduced:

- ProUILayoutHelptextSet()
- ProUILayoutTextSet()
- ProUILayoutEnable()
- ProUILayoutDisable()
- ProUILayoutIsEnabled()
- ProUILayoutShow()
- ProUILayoutHide()
- ProUILayoutIsVisible()

The function **ProUILayoutHelptextSet()** sets the help text associated with the specified layout. The function **ProUILayoutTextSet()** sets the label that appears on the border of a decorated layout.

The functions **ProUILayoutEnable()** and **ProUILayoutDisable()** control whether the layout is sensitive to user input. The function **ProUILayoutIsEnabled()** returns whether layout is sensitive or not.

The functions **ProUILayoutShow()** and **ProUILayoutHide()** control whether the layout is visible in the dialog. The function **ProUILayoutIsVisible()** returns whether or not the layout is visible.
Programming with Lists

The next sections describe programming with Lists.

List Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Values</th>
<th>Pro/TOOLKIT Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>.ColumnLabel</td>
<td>wide string</td>
<td>ProUIListColumnlabelSet()</td>
</tr>
<tr>
<td>.Columns</td>
<td>integer</td>
<td>ProUIListColumnset()</td>
</tr>
<tr>
<td>.ItemHelpText</td>
<td>wide string</td>
<td>ProUIListItemhelptextSet()</td>
</tr>
<tr>
<td>.Labels</td>
<td>wide string</td>
<td>ProUIListLabelsSet()</td>
</tr>
<tr>
<td>.ListType</td>
<td>tabulated, standard, check</td>
<td>ProUIListListtypeSet()</td>
</tr>
<tr>
<td>.MinColumns</td>
<td>integer</td>
<td>—</td>
</tr>
<tr>
<td>.MinRows</td>
<td>integer</td>
<td>ProUIListMinrowsSet()</td>
</tr>
<tr>
<td>.Names</td>
<td>string</td>
<td>ProUIListNamesSet()</td>
</tr>
<tr>
<td>.SelectionPolicy</td>
<td>single, multiple</td>
<td>ProUIListSelectionpolicySet()</td>
</tr>
<tr>
<td>.TabStops</td>
<td>integers</td>
<td>—</td>
</tr>
<tr>
<td>.VisibleRows</td>
<td>integer</td>
<td>ProUIListVisiblerowsSet()</td>
</tr>
</tbody>
</table>

The ColumnLabel attribute specifies the column header if the ListType attribute is set to tabulated. The headers can be divided into columns by inserting tab characters ('t') into the text string.

The Columns attribute specifies the initial width of the list in terms of the number of characters to display. To allow the user to change this width by resizing the dialog, set the AttachLeft and AttachRight attributes to TRUE.

The ItemHelpText attribute specifies the help text to appear when the mouse is left over one item in the list. The value should be a list of strings, one for each item, each enclosed in double quotes. There is no separator between items in the list, although the resource file will allow white space and new lines between strings. If the help text on a single item should contain more than one line, include a newline inside the string itself. If the attribute is specified, the number of help items must be the same as the number of names specified by the Names attribute.
The Labels attribute specifies the set of values displayed in the list component. Each should be enclosed in double quotes. If the list is tabulated, each string can contain tab characters. The number of items in the list must be the same as the number in the Names attribute.

The ListType attribute controls the appearance of the list. The three options are:

- **PROUILISTTYPE_STANDARD**—the items appear as text, and appear in reverse colors when selected.
- **PROUILISTTYPE_TABULATED**—the List component includes a header whose text is defined by the ColumnLabel attribute.
- **PROUILISTTYPE_CHECK**—each item in the list is displayed with a checkbutton that shows whether it is currently selected.

The MinColumns attribute specifies the minimum number of columns that must be displayed if the user changes the width of the list by resizing the dialog.

The Names attribute specifies the names by which the displayed list items are identified from the application. The Pro/TOLKIT functions `ProUIListSelectednamesGet()` and `ProUIListSelectednamesSet()`, described fully later in this chapter, refer to the list items by these names.

The SelectionPolicy attribute takes the integer value of the following types:

- **PROUISELPOLICY_SINGLE**—only one item (or none) can be selected at a time. Selecting one item causes the others to be unselected.
- **PROUISELPOLICY_BROWSE**—exactly one item must be selected at a time. Selecting one item causes the others to be unselected.
- **PROUISELPOLICY_MULTIPLE**—any number of items can be selected. Picking an item causes it to toggle between selected and unselected. Picking with the shift key depressed selects all the items between the current item and the previously selected item.
- **PROUISELPOLICY_EXTENDED**—any number of items can be selected, but the user must explicitly add new items to the selection using the <Control> key.
The TabStops attribute specifies the way in which tab characters in the Labels and ColumnLabel attributes will be interpreted. The value is a list of integers, separated by white space, which specify the column positions to which tabs align.

The VisibleRows attribute specifies the initial number of rows to be visible. This includes the header if the ListType attribute is set to tabulated. If you want the user to be able to change the number of rows by resizing the dialog, set the AttachTop and AttachBottom attributes to TRUE.

The Names, Labels, and ItemHelpText attributes would usually be set programmatically. Refer to the section on Dialog Programming: Common Functions for examples of this.
List Functions

Functions introduced:

- ProUIListActivateActionSet()
- ProUIListSelectActionSet()
- ProUIListTriggerhighlightActionSet()
- ProUIListLastentereditemGet()
- ProUIListFocusinActionSet()
- ProUIListFocusoutActionSet()
- ProUIListHelpertextSet()
- ProUIListItemhelpertextSet()
- ProUIListColumnsSet()
- ProUIListListtypeSet()
- ProUIListColumnlabelSet()
- ProUIListMinrowsSet()
- ProUIListSelectionpolicySet()
- ProUIListVisiblerowsSet()
- ProUIListLabelsGet()
- ProUIListLabelsSet()
- ProUIListNamesGet()
- ProUIListNamesSet()
- ProUIListEnable()
- ProUIListDisable()
- ProUIListIsEnabled()
- ProUIListShow()
- ProUIListHide()
- ProUIListIsVisible()
- ProUIListSelectednamesGet()
- ProUIListSelectednamesSet()

The function ProUIListActivateActionSet() sets a callback which is called when the user hits Return with the mouse over the list.
The function *ProUIListSelectActionSet()* sets a callback which is called whenever the user selects or deselects any item(s) in the list.

The function *ProUIListTriggerhighlightActionSet()* sets a callback which is called whenever the user moves the mouse over an item on the list. Use the function *ProUILastentereditemGet()* to return the name of the item pointed to by the cursor.

The functions *ProUIListFocusinActionSet()* and *ProUIListFocusoutActionSet()* set callback functions that are called whenever the user moves the dialog focus onto or off of the list component using the <Tab> key or the mouse.

The functions *ProUIListHelptextSet()* sets the general help text assigned to the list, while the function *ProUIListItemhelptextSet()* assigns the help text for each individual member of the list.

The function *ProUIListColumnsSet()* sets the width of the list.

The function *ProUIListListTypeSet()* sets the type of list displayed (standard, tabulated or checked).

The function *ProUIListColumnlabelSet()* sets the number and values of the column labels for a tabulated list.

The functions *ProUIListMinrowsSet()* and *ProUIListVisiblerowsSet()* control the current and minimum list height.

The function *ProUIListSelectionpolicySet()* determines the manner in which the user may select items, described in the previous section.

The functions *ProUIListNamesGet()* and *ProUIListLabelsGet()* return the current members in the list. By using *ProUIListNamesSet()* and *ProUIListLabelsSet()* to set the contents of the list programmatically, you can create a dialog which replaces the functionality of *ProMenuStringsSelect()*.

The function *ProUIListDisable()* makes the list unselectable by the user, and *ProUIListEnable()* makes it selectable again. The function *ProUIListIsEnabled()* returns the status of the list.

The functions *ProUIListShow()* and *ProUIListHide()* change the visibility of the list component. The function *ProUIListIsVisible()* returns the list visibility.
**ProUIListSelectednamesGet()** is used to find out which names are currently selected, and **ProUIListSelectednamesSet()** will select names programmatically.

Figure 5-9: Populate dialog with Listed Values

![UI List dialog](image)

**Example 8: UI List resource file**

```pro
(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_textarea)
    (Label  ug_uilist_label_2)
  )

  (Resources
```

User Interface Components 5 - 43
Example 9: 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);
}

User Interface Components
/*====================================================================*
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("ProDrawingSolidsCollect()
   ", "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

```c
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

```c
status = ProMdlNameGet ((ProMdl)solids[i],  model_names[i]);
TEST_CALL_REPORT ("ProMdlNameGet()", "UserUIListImplement()",
    status, status!=PRO_TK_NO_ERROR);
```

Convert widestrings to strings

```c
ProWstringToString(strings[i], model_names[i]);
```

Create and set the pushbutton activate actions

```c
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 ("ProUIPushbuttonActivateActionSet()", "UserUIListImplement()",
    status, status!=PRO_TK_NO_ERROR );

status = ProUIPushbuttonActivateActionSet (list_dialog, list_pbcancel,
    list_CancelAction, NULL);
TEST_CALL_REPORT ("ProUIPushbuttonActivateActionSet()", "UserUIListImplement()",
    status, status!=PRO_TK_NO_ERROR );
```

Populate the list with the model names after activating the dialog and destroy when done

```c
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);
### Programming with MenuBars

The next sections describe programming with MenuBars.

#### MenuBar Attributes

This section describes attributes of MenuBars.

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Values</th>
<th>Pro/TOOLKIT Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Names</td>
<td>string array</td>
<td>ProUIMenubarNamesSet()</td>
</tr>
<tr>
<td>Itemhelptext</td>
<td>wide string array</td>
<td>ProUIMenubarItemhelptextSet()</td>
</tr>
<tr>
<td>Selectablenames</td>
<td>string array</td>
<td>ProUIMenubarSelectablenamesSet()</td>
</tr>
<tr>
<td>Visiblenames</td>
<td>string array</td>
<td>ProUIMenubarVisiblenamesSet()</td>
</tr>
</tbody>
</table>

The attribute Names assigns a name to each of the MenuPanels that belong to the MenuBar. These names do not affect the appearance or behavior of the MenuPanels, but are used only as a reference for the attributes Selectablenames and Visiblenames.

The ItemHelpText attribute specifies the help text to appear when the mouse is left over a button representing a MenuPane in the MenuBar. The value must be a list of strings, one for each item, each enclosed in double quotes. There is no separator between items in the list, although the resource file allows white space and new lines between strings. If the help text on a single item contains more than one line, include a newline inside the string itself. If the attribute is specified, the number of help items must be the same as the number of names specified by the Names attribute.

The Selectablenames attribute identifies which MenuPanels are currently selectable. It refers to the MenuPanels by the names specified in the Names attribute for the MenuBar. If the attribute is not specified, all MenuPanels are selectable. Unselectable items are displayed with their text shown as indented, with no foreground color. It is more usual to specify this attribute dynamically using the function ProUIMenubarSelectablenamesSet().
The Visiblenames attribute identifies which MenuPanes are currently visible. It refers to the MenuPanes by the names specified in the Names attribute for the MenuBar. If the attribute is not specified, all MenuPanes are visible. It is more usual to specify this attribute dynamically using the function `ProUIMenubarVisibleNamesSet()`.

**MenuBar Functions**

Functions introduced:

- `ProUIMenubarNamesSet()`
- `ProUIMenubarItemHelpTextSet()`
- `ProUIMenubarSelectableNamesSet()`
- `ProUIMenubarVisibleNamesSet()`
- `ProUIMenubarEnable()`
- `ProUIMenubarDisable()`
- `ProUIMenubarIsEnabled()`
- `ProUIMenubarShow()`
- `ProUIMenubarHide()`
- `ProUIMenubarIsVisible()`

These functions set the attributes described in section MenuBar Attributes. Use functions `ProUIMenubarSelectableNamesSet()` and `ProUIMenubarVisibleNamesSet()` to set the availability of menus on your menubar according to the context in which the dialog is being used.

The functions `ProUIMenubarEnable()` and `ProUIMenubarDisable()` control the sensitivity of the menubar component. The function `ProUIMenubarIsEnabled()` returns the current sensitivity setting.

The functions `ProUIMenubarShow()` and `ProUIMenubarHide()` control the visibility of the menubar component. The function `ProUIMenubarIsVisible()` returns the current visibility setting.

**Example 10: Resource File for Dialog with Menubar**

The example below shows the resource file used to create the dialog with MenuBars as described in section Dialogs with Menubars.

```plaintext
(Dialog menus
  (Components

5 - 50)
(MenuBar
  MenuBar1
  MenuPane1
  MenuPane2)

(TextArea
  TextArea1)

(Resources
  (TextArea1.Rows 10)
  (TextArea1.Columns 25)
  (.Label "Menubar dialog")
  (.Layout
    (Grid (Rows 1) (Cols 1)
      TextArea1
    )
  )
)

(MenuPane MenuPanel
  (Components
    (PushButton PushButton1)
    (PushButton PushButton2)
  )

  (Resources
    (.Label "MenuPanel")
    (PushButton1.Label "Button1")
    (PushButton2.Label "Button2")
  )
)

(MenuPane MenuPane2
  (Components
    (PushButton PushButton3)
    (Separator Separator1)
    (RadioGroup RadioGroup1)
    (RadioGroup RadioGroup2)
    (Separator Separator2)
    (CascadeButton CascadeButton2 MenuPane4)
    (CheckButton CheckButton1)
  )

  (Resources
    (.Label "MenuPane2")
    (PushButton3.Label "PushButton")
    (RadioGroup1.Names "RG1" "RG2")
    (RadioGroup1.Labels "Radio group 1" "Radio group 2")
    (CascadeButton2.Label "Cascade button")
  )
Programming with MenuPanes

The next sections describe programming with MenuPanes.

MenuPane Attributes

This section describes attributes of MenuPanes.

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Values</th>
<th>Pro/TOOLKIT Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label</td>
<td>string</td>
<td>ProUIMenupaneTextSet()</td>
</tr>
</tbody>
</table>

The attribute Label determines the text that will appear on the MenuBar button which activates the MenuPane. If the attribute is not specified, the name of the MenuPane is used.

MenuPane Functions

Functions introduced:

- ProUIMenupaneTextSet()
- ProUIMenupaneShow()
- ProUIMenupaneHide()
- ProUIMenupanelsVisible()

The function ProUIMenupanetextSet() sets the Label attribute described above.

The functions ProUIMenupaneShow() and ProUIMenupaneHide() control the visibility of the menupane component. The function ProUIMenupanelsVisible() returns current visibility setting.
### Programming with OptionMenus

The next sections describe programming with OptionMenus.

#### OptionMenu Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Values</th>
<th>Pro/TOOLKIT Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>.Columns</td>
<td>integer</td>
<td>ProUIOptionmenuColumnsSet()</td>
</tr>
<tr>
<td>.Editable</td>
<td>TRUE, FALSE (default)</td>
<td>ProUIOptionmenuEditable()</td>
</tr>
<tr>
<td>.ItemHelpText</td>
<td>wide string array</td>
<td>ProUIOptionmenuItemhelptextSet()</td>
</tr>
<tr>
<td>.Labels</td>
<td>wide string array</td>
<td>ProUIOptionmenuLabelsSet()</td>
</tr>
<tr>
<td>.MinColumns</td>
<td>integer (default 4)</td>
<td></td>
</tr>
<tr>
<td>.Names</td>
<td>wide string array</td>
<td>ProUIOptionmenuNamesSet()</td>
</tr>
<tr>
<td>.VisibleRows</td>
<td>integer</td>
<td>ProUIOptionmenuVisiblerowsSet()</td>
</tr>
</tbody>
</table>

The Columns attribute specifies the initial width of the OptionMenu in terms of the number of characters to display. To allow the user to change this width by resizing the dialog, set the AttachLeft and AttachRight attributes to TRUE.

The Editable attribute specifies whether the user may enter a custom string in the option menu component in addition to selecting an option in the pull-down menu.

The ItemHelpText attribute specifies the help text to appear when the mouse is left over one item in the option menu. The value should be a list of strings, one for each item, each enclosed in double quotes. There is no separator between items in the list, although the resource file will allow white space and new lines between strings. If the help text on a single item should contain more than one line, include a newline inside the string itself. If the attribute is specified, the number of help items must be the same as the number of names specified by the Names attribute.

The Labels attribute specifies the set of values that are displayed in the option menu. Each should be enclosed in double quotes. The number of items in the list must be the same as the number in the Names attribute.

The MinColumns attribute specifies the minimum number of columns that must be displayed if the user changes the width of the option menu by resizing the dialog.
The Names attribute specifies the names by which the displayed option menu items are identified from the application. The Pro/TOOLKIT functions `ProUIOptionmenuValueSet()` and `ProUIOptionmenuValueGet()`, described fully later in this chapter, refer to the option menu items by these names.

The VisibleRows attribute specifies the number of rows visible in the menu of options that appears when the user selects the option menu push button. If the visible number is less than the total number, the menu has a vertical scroll bar.

**OptionsMenu Functions**

Functions introduced:

- `ProUIOptionmenuActivateActionSet()`
- `ProUIOptionmenuSelectActionSet()`
- `ProUIOptionmenuInputActionSet()`
- `ProUIOptionmenuTriggerhighlightActionSet()`
- `ProUIOptionmenuLastentereditemGet()`
- `ProUIOptionmenuFocusinActionSet()`
- `ProUIOptionmenuFocusoutActionSet()`
- `ProUIOptionmenuHelpsetText()`
- `ProUIOptionmenuColumnsSet()`
- `ProUIOptionmenuItemhelptextSet()`
- `ProUIOptionmenuLabelsSet()`
- `ProUIOptionmenuNamesSet()`
- `ProUIOptionmenuVisibleRowsSet()`
- `ProUIOptionmenuEditable()`
- `ProUIOptionmenuValueSet()`
- `ProUIOptionmenuValueGet()`
- `ProUIOptionmenuSelectednamesSet()`
- `ProUIOptionmenuSelectednamesGet()`
- `ProUIOptionmenuEnable()`
- `ProUIOptionmenuDisable()`
- `ProUIOptionmenuShow()`
- `ProUIOptionmenuHide()`
The function `ProUIOptionmenuActivateActionSet()` sets a callback which is called when the user presses Return with the mouse over the list.

The function `ProUIOptionmenuSelectActionSet()` sets a callback which is called whenever the user changes the current selected option.

The function `ProUIOptionmenuInputActionSet()` sets a callback which is called when the user enters text in the option menu. This is valid only for Editable option menus.

The function `ProUIOptionmenuTriggerhighlightActionSet()` sets a callback that is called when the user moves the mouse over an item in the option menu. Use the function `ProUIOptionmenuLastentereditemGet()` to return the name of the item pointed to by the cursor.

The functions `ProUIOptionmenuHelptextSet()` and `ProUIOptionmenuItemhelptextSet()` set the helptext for the option menu and for each individual item in the option menu.

The function `ProUIOptionmenuColumnsSet()` sets the width of the option menu.

The functions `ProUIoptionmenuNamesSet()` and `ProUIOptionmenuLabelsSet()` set the internal and user-visible names of the items in the option menu.

The function `ProUIOptionmenuVisiblerowsSet()` sets the number of rows visible when the user uses the pull down option menu.

The function `ProUIOptionmenuEditable()` makes the option menu editable, which allows the user to input a string in the menu.

The functions `ProUIOptionmenuValueGet()` and `ProUIOptionmenuValueSet()` get and set the current selected option. Note that `ProUIOptionmenuValueGet()` returns the label of the option item.

The functions `ProUIOptionmenuSelectednamesGet()` and `ProUIOptionmenuSelectednamesSet()` get and set the current selected names in the menu.

The function `ProUIOptionmenuDisable()` disables the option menu for the user, and the `ProUIOptionmenuEnable()` makes it selectable again.
The functions *ProUIOptionmenuShow()* and *ProUIOptionmenuHide()* control the visibility of the option menu. The function *ProUIOptionmenusVisible()* returns the option menu visibility status.

**Programming with ProgressBars**

The next sections describe programming with ProgressBars.

**ProgressBar Attributes**

This section describes attributes for ProgressBars.

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Values</th>
<th>Pro/TOOLKIT Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>integer</td>
<td>ProUIProgressbarIntegerSet()</td>
</tr>
<tr>
<td>Length</td>
<td>integer</td>
<td>ProUIProgressbarLengthSet()</td>
</tr>
<tr>
<td>Maxinteger</td>
<td>integer</td>
<td>ProUIProgressbarMaxintegerSet()</td>
</tr>
<tr>
<td>Mininteger</td>
<td>integer</td>
<td>ProUIProgressbarMinintegerSet()</td>
</tr>
<tr>
<td>Orientation</td>
<td>boolean</td>
<td>-</td>
</tr>
<tr>
<td>ProgressStyle</td>
<td>integer: one of the values of the ProUIProgressstyle enum in ProUI.h</td>
<td>ProUIProgressbarProgressstyleSet()</td>
</tr>
</tbody>
</table>

The Integer attribute determines the current value of the progress bar. By setting this in the resource file, you set the initial value of the progress.

The Length attribute determines the length of the ProgressBar. The units are approximate character widths.

Maxinteger and Mininteger set the limiting values of the number that records the current progress status. For example, if Mininteger is -100 and Maxinteger is 100, the progress will show 50% when the progressbar integer is set to 0.

The Orientation attribute should be False (the default) for a horizontal progressbar, and True for a vertical one.

The ProgressStyle attribute determines the appearances of the progress bar, and takes the following values:

- **PROUIPROGRESS_NOTEXT**—The progress bar progressively changes color but shows no value
• `PROUIPROGRESS_VALUE`—The integer value of the progress is shown as text.

• `PROUIPROGRESS_PERCENT` (Percentage Value)—The value of the progress is shown as a percentage of the difference between Mininteger and Maxinteger.

• `PROUIPROGRESS_INTERVALS` (Intervals)—The progress is shown in separate blocks of color with no text.

**Note:** The default progress style is Percentage Value.

The styles are illustrated in the following dialog, and explained in the example code.

Figure 5-10: Progress Bar

![Progress Bar](image)

Example 11: Progress Bar Resource File

```plaintext
(Dialog progressbars
  (Components
    (ProgressBar                    ProgressBar2)
    (ProgressBar                    ProgressBar3)
    (ProgressBar                    ProgressBar4)
    (ProgressBar                    ProgressBar5)
    (Label                          Label2)
    (Label                          Label3)
    (Label                          Label4)
    (Label                          Label5)
    (Separator                      Separator1)
    (SpinBox                        SpinBox1)
    (SpinBox                        SpinBox2)
    (ThumbWheel                     ThumbWheel1)
  )

  (Resources
    (ProgressBar2.ProgressStyle     0)
    (ProgressBar2.TopOffset         4)
    (ProgressBar2.BottomOffset      4)
    (ProgressBar2.LeftOffset        4)
  )
)
```

User Interface Components
ProgressBar Functions

Functions introduced:

- ProUIProgressbarHelpTextSet()
- ProUIProgressbarLengthSet()
- ProUIProgressbarMinintegerSet()
- ProUIProgressbarMaxintegerSet()
- ProUIProgressbarProgressstyleSet()
- ProUIProgressbarIntegerSet()
- ProUIProgressbarShow()
- ProUIProgressbarHide()
- ProUIProgressbarIsVisible()

The functions ProUIProgressbarHelpTextSet(), ProUIProgressbarLengthSet(), ProUIProgressbarMinintegerSet(), ProUIProgressbarMaxintegerSet(), and ProUIProgressbarProgressstyleSet() set the attributes described in section ProgressBar Attributes.

The functions ProUIProgressbarShow() and ProUIProgressbarHide() control the visibility of the progressbar. The function ProUIProgressbarIsVisible() return visibility of the progressbar.

How to Program a Dialog Containing a Progress Bar

Unlike the other dialog components supported by Pro/TOOLKIT, a progress bar needs to have its display updated in the absence of user interaction. This requires you to program the dialog in a different way. The differences are:

- The .DialogStyle attribute for the owning dialog should be 6 (which means "Working"). This means that the dialog is non-model and will therefore redisplay when you update the progress bar, even if it's not expecting user input.
• When you call `ProUIDialogActivate()` for a “Working” dialog, the function displays the dialog and returns immediately, instead of the usual behavior which is to block other input and return only when you call `ProUIDialogExit()`.

• You don’t need to call `ProUIDialogExit()`.

• Call `ProUIDialogDestroy()` to remove the dialog, but only when the progress is complete or the user has asked to quit. Don’t call it from within a callback to a dialog component.

• You should call `ProUIDialogActivate()` to update the display each time you change the value of the progress bar (using `ProUIProgressbarIntegerSet()`).

Do not set the `.DialogStyle` to any other value for any other dialog which does not contain a progress bar. Non-modal dialogs are difficult to use correctly, and not yet fully supported by Pro/TOOLKIT.

• If you have other buttons on a “Working” dialog which also contains a progress bar, remember that they could be selected at any time before you call `ProUIDialogDestroy()`, not only while `ProUIDialogActivate()` is executing.

Use few other components on a dialog that contains a progress bar — preferably a Stop button at most.

### Programming with Pushbuttons

The next sections describe programming with Push Buttons.

### Pushbutton Attributes

The Label attribute specifies the text of the label, which is to appear on the push button.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Values</th>
<th>Pro/TOOLKIT Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>.Label</td>
<td>wide string</td>
<td>—</td>
</tr>
</tbody>
</table>


Pushbutton Functions

Functions Introduced:

- `ProUIPushbuttonActivateActionSet()`
- `ProUIPushbuttonHelptextSet()`
- `ProUIPushbuttonTextSet()`
- `ProUIPushbuttonEnable()`
- `ProUIPushbuttonDisable()`
- `ProUIPushbuttonIsEnabled()`
- `ProUIPushbuttonShow()`
- `ProUIPushbuttonHide()`
- `ProUIPushbuttonIsVisible()`

The function `ProUIPushbuttonActivateActionSet()` sets the action performed when the pushbutton is activated (pushed). The function `ProUIPushbuttonHelptextSet()` sets the help text associated with this pushbutton.

The function `ProUIPushbuttonTextSet()` sets the label for the pushbutton.

The functions `ProUIPushbuttonEnable()` and `ProUIPushbuttonDisable()` control whether the button is sensitive to user input. The function `ProUIPushbuttonIsEnabled()` returns whether the pushbutton is sensitive or not.

Figure 5-11: Component Visibility

Example 12: Component Visibility resource file

```plaintext
(Dialog uguivisibility
  (Components
    (SubLayout       PushbuttonVisibility)
    (PushButton      CloseButton)
  )
  (Resources
```

User Interface Components 5 - 61
(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 PushbuttonVisibility
(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)
      TargetBtn
      VisibleCheck
  )

5 - 62 Pro/TOOLKIT User's Guide
Example 13: Controlling Component Visibility or Sensitivity at Runtime

The source code for the Component Visibility resource file is given below.

```c
#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" checkbutton
/*====================================================================*/
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" checkbutton
/*====================================================================*/
static void UserSensitivityToggle (char* dialog, char* button, ProAppData data)
{
    // Code for UserSensitivityToggle
}
```

User Interface Components
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);
The next sections describe programming with RadioGroups.

RadioGroup Attributes

The ItemHelpText attribute specifies the help text that will appear on each button in the radio group.

The Labels attribute specifies the text that will be displayed against each button in the group.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Values</th>
<th>Pro/TOOLKIT Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>.ItemHelpText</td>
<td>wide string</td>
<td>ProUIRadiogroupItemhelpsetTextSet()</td>
</tr>
<tr>
<td>.Labels</td>
<td>wide string</td>
<td><strong>ProUIRadiogroupLabelsSet()</strong></td>
</tr>
<tr>
<td>.Names</td>
<td>wide string</td>
<td>ProUIRadiogroupNamesSet()</td>
</tr>
<tr>
<td>.Orientation</td>
<td>FALSE (default)</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>TRUE</td>
<td>—</td>
</tr>
</tbody>
</table>

The Names attribute specifies the names by which the radio buttons will be identified from the application. The Pro/TOOLKIT functions **ProUIRadiogroupSelectednamesGet()** and **ProUIRadiogroupSelectednamesSet()** refer to the button by these names.

The Orientation attribute specifies whether the radio buttons is in a horizontal row (FALSE - the default), or a vertical row (TRUE). Pro/ENGINEER convention is to make all radio buttons vertical.
RadioGroup Functions

Functions introduced:

- `ProUIRadiogroupSelectActionSet()`
- `ProUIRadiogroupHelpsetTextSet()`
- `ProUIRadiogroupItemHelpsetTextSet()`
- `ProUIRadiogroupLabelsSet()`
- `ProUIRadiogroupLabelsGet()`
- `ProUIRadiogroupNamesSet()`
- `ProUIRadiogroupNamesGet()`
- `ProUIRadiogroupSelectednamesGet()`
- `ProUIRadiogroupSelectednamesSet()`
- `ProUIRadiogroupEnable()`
- `ProUIRadiogroupDisable()`
- `ProUIRadiogroupIsEnabled()`
- `ProUIRadiogroupShow()`
- `ProUIRadiogroupHide()`
- `ProUIRadiogroupIsVisible()`

The function `ProUIRadiogroupSelectActionSet()` sets a callback which is called each time the user selects one of the buttons in the radio group.

The functions `ProUIRadiogroupHelpsetTextSet()` though `ProUIRadiogroupNamesSet()` set radio group attributes that can also be set in the resource file, and are documented in the section Radio Group Attributes. The function `ProUIRadiogroupNamesSet()` defines the names by which the buttons in the group are identified from the application code, and therefore also their number. The function `ProUIRadiogroupLabelsSet()` defines the strings displayed alongside the buttons. The functions `ProUIRadiogroupNamesGet()` and `ProUIRadiogroupLabelsGet()` return the current internal and external labels for the radio group.

`ProUIRadiogroupSelectednamesGet()` and `ProUIRadiogroupSelectednamesSet()` get and set the name of the button which is currently selected.
The functions `ProURIRadiogroupEnable()` and `ProURIRadiogroupDisable()` control whether the button is sensitive to user input. The function `ProURIRadiogroupIsEnabled()` returns whether or not the radiogroup is sensitive.

The functions `ProURIRadiogroupShow()` and `ProURIRadiogroupHide()` control whether the button is visible. The function `ProURIRadiogroupIsVisible()` returns whether radiogroup is visible or not.

### Programming with Separators

The next sections describe programming with Separators.

#### Separator Attributes

This section describes attributes for Separators.

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Values</th>
<th>Pro/TOOLKIT Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>.Length</td>
<td>integer</td>
<td>—</td>
</tr>
<tr>
<td>.Orientation</td>
<td>boolean</td>
<td>—</td>
</tr>
</tbody>
</table>

The Length attribute determines the length of the separator, in approximate character sizes. If this is set to 0, the length of the separator will match the size of the neighboring components.

The Orientation attribute should be False (the default) for a horizontal separator, or True for a vertical one. Note that a separator on a menu pane will always be horizontal.

#### Separator Functions

There are no functions to modify attributes of a separator.

### Programming with Sliders

The next sections describe programming with Sliders.

#### Slider Attributes

This section describes attributes for Sliders.

The Integer attribute specifies the current value of the slider.
The Length attribute specifies the displayed length of the slider. The default is 8.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Values</th>
<th>Pro/TOOLKIT Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>.Integer</td>
<td>integer</td>
<td>ProUISliderIntegerGet()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ProUISliderIntegerSet()</td>
</tr>
<tr>
<td>.Length</td>
<td>integer</td>
<td>ProUISliderLengthSet()</td>
</tr>
<tr>
<td>.MaxInteger</td>
<td>integer</td>
<td>ProUISliderMaxintegerSet()</td>
</tr>
<tr>
<td>.MinInteger</td>
<td>integer</td>
<td>ProUISliderMinintegerSet()</td>
</tr>
<tr>
<td>.Orientation</td>
<td>TRUE, FALSE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(default)</td>
<td>—</td>
</tr>
<tr>
<td>.Tracking</td>
<td>TRUE, FALSE</td>
<td>ProUISliderDisableTracking()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ProUISliderEnableTracking()</td>
</tr>
</tbody>
</table>

The MaxInteger and MinInteger attributes specify the range of values which the slider can take.

The Orientation attribute should be FALSE (the default) if the slider should be horizontal, and TRUE for vertical.

The Tracking attribute specifies whether the action callback should be called after every move of the slider (TRUE) or only when the user releases the mouse button (FALSE).

Generally set Tracking to TRUE, and combine the slider with an input panel which shows the current value of the slider.
Slider Functions

Functions introduced:

- `ProUISliderUpdateActionSet()`
- `ProUISliderHelpTextSet()`
- `ProUISliderLengthSet()`
- `ProUISliderMinIntegerSet()`
- `ProUISliderMaxIntegerSet()`
- `ProUISliderDisableTracking()`
- `ProUISliderEnableTracking()`
- `ProUISliderIntegerGet()`
- `ProUISliderIntegerSet()`
- `ProUISliderEnable()`
- `ProUISliderDisable()`
- `ProUISliderIsEnabled()`
- `ProUISliderShow()`
- `ProUISliderHide()`
- `ProUISliderIsVisible()`

The function `ProUISliderUpdateActionSet()` sets a callback which is called when the slider is moved, or when the slider is released by the user, depending upon the setting of the Tracking attribute.

The remaining functions listed above set and get attributes defined in the section Slider Attributes.

The functions `ProUISliderEnable()` and `ProUISliderDisable()` control whether the button is sensitive to user input. The function `ProUISliderIsEnabled()` returns whether the slider is sensitive or not.

The functions `ProUISliderShow()` and `ProUISliderHide()` control whether the button is visible. The function `ProUISliderIsVisible()` returns whether or not the slider is visible.

It is common to associate a slider with an input panel to show the current value, and allow the value to be entered through the input panel for more delicate control.
Example 14: Resource File for Dialog with Slider and Linked InputPanel

This example shows the resource file for a dialog with slider and linked input panel.

(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
      )
   )
)
Slider
InputPanel

(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
)
)
)

The source code of the application is below.

Example 15: 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 "ProUIDialog.h"
#include "ProUIInputpanel.h"
#include "ProUILabel.h"
#include "ProUISlider.h"
static ProName msgfil;
int user_initialize()
{
  int menu_id, UsrModAngle();
  ProStringToWstring(msgfil,"umsg.txt");
```
ProMenuFileRegister("part","part.mnu",&menu_id);
ProMenuAuxfileRegister("part","part.aux",&menu_id);
ProMenuButtonActionSet("part","Slider",UsrModAngle,NULL,0);
return(0); }

global user_terminate() {
}

/*====================================================================*
FUNCTION : UsrModAngle() PURPOSE : Command to modify an angular
dimension using a slider
/*====================================================================*/
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(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);
sprintf(prompt,"Enter the new value of %s",name);

Input the new angle using a dialog 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);

#define OK      1
#define CANCEL  0

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 : UsrSliderAction() PURPOSE : Callback for movement of the slider
void UsrSliderAction(
    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
    /*-------------------------------------------------------------*/
    sprintf(str, "%d", *angle);
    ProStringToWstring(wstr, str);
    ProUIInputpanelValueSet(dialog, "InputPanel", wstr);
}

فجر Function: UsrInputpanelAction()  Purpose: Callback for input to the
input panel
فجر 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
 ProStringToString(wline,text);
 ProUILabelTextSet("angle","Prompt",wline);

 Set the OK and Cancel callbacks
 ProUIPushbuttonActivateActionSet("angle","OK",UsrOKAction,NULL);
 ProUIPushbuttonActivateActionSet("angle","Cancel",UsrCancelAction,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
 sprintf(str, "%d", *angle);
 ProStringToString(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);
Programming with Spinboxes

The next sections describe programming with Spinboxes.

Spinbox Attributes

This section describes attributes for Spinboxes.

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Values</th>
<th>Pro/TOOLKIT Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Columns</td>
<td>integer</td>
<td>—</td>
</tr>
<tr>
<td>Delay</td>
<td>integer</td>
<td>—</td>
</tr>
<tr>
<td>Digits</td>
<td>integer</td>
<td>ProUISpinboxDigitsSet()</td>
</tr>
<tr>
<td>Double</td>
<td>double</td>
<td>ProUISpinboxDoubleSet()</td>
</tr>
<tr>
<td>DoubleFormat</td>
<td>string</td>
<td>ProUISpinboxDoubleFormatSet()</td>
</tr>
<tr>
<td>DoubleIncrement</td>
<td>double</td>
<td>ProUISpinboxDoubleIncrementSet()</td>
</tr>
<tr>
<td>FastDoubleIncrement</td>
<td>double</td>
<td>ProUISpinboxFastDoubleIncrementSet()</td>
</tr>
<tr>
<td>FastIncrement</td>
<td>integer</td>
<td>ProUISpinboxFastIncrementSet()</td>
</tr>
<tr>
<td>Increment</td>
<td>integer</td>
<td>ProUISpinboxIncrementSet()</td>
</tr>
<tr>
<td>InputType</td>
<td>integer: one of the values of the enum ProUIInputType in ProUI.h</td>
<td>—</td>
</tr>
<tr>
<td>Integer</td>
<td>integer</td>
<td>ProUISpinboxIntegerSet()</td>
</tr>
<tr>
<td>MaxDouble</td>
<td>double</td>
<td>ProUISpinboxMaxDoubleSet()</td>
</tr>
<tr>
<td>MinDouble</td>
<td>double</td>
<td>ProUISpinboxMinDoubleSet()</td>
</tr>
<tr>
<td>MaxInteger</td>
<td>integer</td>
<td>ProUISpinboxMaxIntegerSet()</td>
</tr>
<tr>
<td>Mininteger</td>
<td>integer</td>
<td>ProUISpinboxMinIntegerSet()</td>
</tr>
<tr>
<td>Rate</td>
<td>double</td>
<td>—</td>
</tr>
</tbody>
</table>

The Columns attribute determines the width of the spin box in approximate character widths.

When you click once on one of the arrow buttons on a spinbox, the values changes by an increment which is determined by an attribute. If you hold the button down, the value changes by repeated increments for a short time, and then by larger increments.
The Delay attribute determines the time in milliseconds after the button is pressed before the larger increment is applied. The default is 2000 (2 seconds).

A Spinbox can take either an integer or a double value. This is determined by the value of the InputType attribute, which is 2 for Integer (the default) and 3 for Double.

If the InputType is Integer, these attributes apply:

- Digits— the number of digits to display. The default is -1 which means no leading zeros.
- Integer—the current value
- Maxinteger—the maximum value
- Mininteger—the minimum value
- Increment—the initial increment
- FastIncrement—the increment applied after the delay

If the InputType is Double, these attributes apply:

- Digits—the number of digits to display. The default is -1 which means no leading zeros before the point, or trailing after.
- Double—the current value
- DoubleFormat—the format of the double
- DoubleIncrement—the initial increment
- FastDoubleIncrement—the increment applied after the delay
- MaxDouble—the maximum value
- MinDouble—the minimum value

The DoubleFormat is similar to the format string for printf(), except that the character '*' should be used in place of the number of digits - the number of digits is set by the Digits attribute. The default format is %.lf.

Other examples are

\[
%.1e
\]

\[
%12.1f
\]

The Rate attribute determines the number of times per second that the fast increment will be applied after the button has been held for longer than the specific delay. The default is 1.
Spinbox Functions

Functions introduced:

- ProUISpinboxHelptextSet()
- ProUISpinboxEditable()
- ProUISpinboxReadOnly()
- ProUISpinboxInputtypeSet()
- ProUISpinboxIntegerSet()
- ProUISpinboxIntegerGet()
- ProUISpinboxMinintegerSet()
- ProUISpinboxMaxintegerSet()
- ProUISpinboxIncrementSet()
- ProUISpinboxFastincrementSet()
- ProUISpinboxDoubleSet()
- ProUISpinboxDoubleGet()
- ProUISpinboxMindoubleSet()
- ProUISpinboxMaxdoubleSet()
- ProUISpinboxDigitsSet()
- ProUISpinboxDoubleformatSet()
- ProUISpinboxDoubleincrementSet()
- ProUISpinboxFastdoubleincrementSet()
- ProUISpinboxUpdateActionSet()
- ProUISpinboxActivateActionSet()
- ProUISpinboxEnable()
- ProUISpinboxDisable()
- ProUISpinboxIsEnabled()
- ProUISpinboxShow()
- ProUISpinboxHide()
- ProUISpinboxIsVisible()

The functions named after attributes described in section Spinbox Attributes can be used to set those attributes dynamically.
The function `ProUISpinboxReadonly()` sets the spinbox so only able to be modified, and not edit the value directly. The function `ProUISpinboxEditable()` reverses that setting.

The function `ProUISpinboxUpdateActionSet()` sets a callback function to be called whenever the spin box value changes. The function `ProUISpinboxActivateActionSet()` sets a callback to be called whenever the user clicks the mouse in the spinbox.

The functions `ProUISpinboxEnable()` and `ProUISpinboxDisable()` control whether the button is sensitive to user input. The function `ProUISpinboxIsEnabled()` returns whether or not the spinbox is sensitive.

The functions `ProUISpinboxShow()` and `ProUISpinboxHide()` control whether the button is visible. The function `ProUISpinboxIsVisible()` returns whether or not spinbox is visible.

### Programming with Tabs

The next sections describe programming with Tabs.

### Tab Attributes

The Decorated attribute specifies whether the user can switch the layouts in the tab by selecting layout handles. Always set this to TRUE in this release.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Values</th>
<th>Pro/TOOLKIT Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>.Decorated</td>
<td>TRUE</td>
<td>—</td>
</tr>
<tr>
<td>.ItemHelpText</td>
<td>wide string</td>
<td><code>ProUITabItemhelpsetText()</code></td>
</tr>
<tr>
<td>.Labels</td>
<td>wide string array</td>
<td><code>ProUITabLabelsSet()</code>, <code>ProUITabLabelsGet()</code></td>
</tr>
</tbody>
</table>

The ItemHelpText attribute specifies the help text to appear when the mouse is positioned over the handles for the sublayouts. The number of text items should match the number of sublayouts in the tab component.

The Labels attribute specified the names of the tab entries. If this is not used the tabs inherit the text assigned to the individual layouts.
Tab Functions

Functions introduced:

- `ProUITabSelectActionSet()`
- `ProUITabHelpsetText()`
- `ProUITabItemhelpsetText()`
- `ProUITabSelectednamesGet()`
- `ProUITabSelectednamesSet()`
- `ProUITabLabelsSet()`
- `ProUITabLabelsGet()`
- `ProUITabEnable()`
- `ProUITabDisable()`
- `ProUITabIsEnabled()`
- `ProUITabShow()`
- `ProUITabHide()`
- `ProUITabisVisible()`

The function `ProUITabSelectActionSet()` sets the callback to be called when the user selects one of the tab layout handles, to make that layout visible.

The functions `ProUITabHelpsetText()` and `ProUITabItemhelpsetText()` set the tab attributes HelpText and ItemHelpText described in the earlier sections Common Attributes and Tab Attributes.

The function `ProUITabSelectednamesGet()` returns the name of the currently selected (visible) tab panel. The function `ProUITabSelectednamesSet()` sets the currently visible tab panel.

The functions `ProUITabLabelsSet()` and `ProUITabLabelsGet()` affect the user-visible text for the tab components.

The functions `ProUITabEnable()` and `ProUITabDisable()` control whether the button is sensitive to user input. The function `ProUITabIsEnabled()` returns whether or not the tab is sensitive.

The functions `ProUITabShow()` and `ProUITabHide()` control whether the button is visible. The function `ProUITabisVisible()` returns whether or not the tab is visible.
Programming with Tables

The next sections describe programming with Tables.

Table Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Value</th>
<th>Pro/TOOLKIT Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>.ActivateOnReturn</td>
<td>TRUE, FALSE</td>
<td></td>
</tr>
<tr>
<td>.Alignment</td>
<td>integer - use values defined in ProUI.h for ProUIAlignment</td>
<td></td>
</tr>
<tr>
<td>.AutoHighlight</td>
<td>TRUE, FALSE</td>
<td>ProUITableAutohighlightEnable()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ProUITableAutohighlightDisable()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ProUITableIsAutohighlightEnabled()</td>
</tr>
<tr>
<td>.ColumnLabels</td>
<td>wide-string array</td>
<td>ProUITableColumnlabelsGet()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ProUITableColumnlabelsSet()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ProUIColumnLabelGet()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ProUIColumnLabelSet()</td>
</tr>
<tr>
<td>.ColumnNames</td>
<td>string array</td>
<td>ProUITableColumnNamesGet()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ProUITableColumnNamesSet()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ProUITableColumnRename()</td>
</tr>
<tr>
<td>.ColumnResizings</td>
<td>integer array(0= do not resize, else use integer value as resizing factor relative to other columns)</td>
<td>ProUITableColumnresizingsSet()</td>
</tr>
<tr>
<td>.ColumnSelectionPolicy</td>
<td>integer - use values defined in ProUI.h for ProUISelectionPolicy</td>
<td>ProUITableColumnselectionpolicySet()</td>
</tr>
<tr>
<td>.ColumnWidths</td>
<td>integer array (number of characters in each column)</td>
<td>ProUITableColumnwidthsSet()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ProUITableColumnWidthSet()</td>
</tr>
<tr>
<td>.Columns</td>
<td>integer(number of characters)</td>
<td>ProUITableColumnsSet()</td>
</tr>
<tr>
<td>.DefaultColumnWidth</td>
<td>integer(number of characters)</td>
<td></td>
</tr>
<tr>
<td>.DefaultHelpText</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>Attribute Name</td>
<td>Value</td>
<td>Pro/TOOLKIT Functions</td>
</tr>
<tr>
<td>----------------------</td>
<td>----------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>.LockedColumns</td>
<td>Integer</td>
<td>ProUITableLockedcolumnsSet()</td>
</tr>
<tr>
<td>.LockedRows</td>
<td>Integer</td>
<td>ProUITableLockedrowsSet()</td>
</tr>
<tr>
<td>.MinColumns</td>
<td>Integer</td>
<td>ProUITableMincolumnsSet()</td>
</tr>
<tr>
<td>.MinRows</td>
<td>Integer</td>
<td>ProUITableMinrowsSet()</td>
</tr>
<tr>
<td>.RowLabels</td>
<td>wide-string array</td>
<td>ProUITableRowlabelsGet()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ProUITableRowlabelsSet()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ProUITableRowLabelGet()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ProUITableRowLabelSet()</td>
</tr>
<tr>
<td>.RowNames</td>
<td>string array</td>
<td>ProUITableRownamesGet()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ProUITableRownamesSet()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ProUITableRowRename()</td>
</tr>
<tr>
<td>.ScrollBarsWhenNeeded</td>
<td>TRUE, FALSE</td>
<td></td>
</tr>
<tr>
<td>.SelectionPolicy</td>
<td>integer - use values defined in ProUI.h for ProUISelectionPolicy</td>
<td>ProUITableSelectionPolicySet()</td>
</tr>
<tr>
<td>.Sensitive</td>
<td>TRUE, FALSE</td>
<td>ProUITableEnable()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ProUITableDisable()</td>
</tr>
<tr>
<td>.ShowGrid</td>
<td>TRUE, FALSE</td>
<td></td>
</tr>
<tr>
<td>.TruncateLabel</td>
<td>TRUE, FALSE</td>
<td>ProUITableShow()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ProUITableHide()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ProUITableIsVisible()</td>
</tr>
<tr>
<td>.Visible</td>
<td>TRUE, FALSE</td>
<td></td>
</tr>
<tr>
<td>.VisibleRows</td>
<td>Integer</td>
<td>ProUITableVisiblerowsSet()</td>
</tr>
</tbody>
</table>

The ActivateOnReturn attribute specifies whether RETURN key press should generate a UI_ACTIVATE_ACTION callback or whether it should cause the default button in the dialog to be pressed.

The Alignment attribute specifies the alignment of text in the column and row header labels of the table.

The AutoHighlight attribute indicates whether to highlight an entire row when a selection is made in a cell in the row.

The ColumnLabels attribute specifies the labels of the columns of the table. The table displays column headers only if labels are specified.
The ColumnNames attribute specifies the names of columns of the table.

The ColumnResizings attribute specifies the resizing factors of the columns of the table. The default value of factor is 0.

The ColumnSelectionPolicy attribute specifies the selection policy of the Table.

The ColumnWidths attribute specifies the widths of the columns of the Table, in character widths. The default is specified by the value of UI_default_column_width_Attr.

The Columns attribute specifies the width of the Table, in character widths.

The DefaultColumnWidth attribute specifies the default column width if no widths are specified or if value of any width is specified lesser than 0.

The DefaultHelpText attribute specifies the help-text to be displayed when the pointer is over the component but is not over any of the cells of the Table.

The LockedColumns and LockedRows attribute specifies the number of locked columns and rows of the Table respectively.

The MinColumns attribute specifies the minimum width of the Table, in character widths.

The MinRows attribute specifies the minimum number of visible rows of the Table.

The RowLabels attribute specifies the labels of the rows of the Table. The Table displays row headers if labels are specified.

The RowNames attribute specifies the names of rows of the Table.

The ScrollBarsWhenNeeded attribute sets a flag as TRUE, indicating that the scrollbars should be displayed only when they are required or as FALSE indicating that scrollbars should always be visible.
The SelectionPolicy attribute specifies the selection policy of the attribute as shown in the following table.

<table>
<thead>
<tr>
<th>Selection Policy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UI_NO_SELECT</td>
<td>No selected items</td>
</tr>
<tr>
<td>UI_SINGLE_SELECT</td>
<td>1 or 0 selected items</td>
</tr>
<tr>
<td>UI_BROWSE_SELECT</td>
<td>Always 1 selected item</td>
</tr>
<tr>
<td>UI_MULTIPLE_SELECT</td>
<td>Any number of selected items</td>
</tr>
<tr>
<td>UI_EXTENDED_SELECT</td>
<td>Any number of selected items, including support for range selection</td>
</tr>
</tbody>
</table>

The Sensitive attribute specifies whether a flag is set FALSE indicating whether the component is disabled, or TRUE indicating that the component is sensitive to user input.

The ShowGrid attribute indicates whether to display grid lines of the Table.

The TruncateLabel attribute indicates whether to truncate labels of newly created cells in the Table to the size of their cell.

**Note:** Modifying the TruncateLabel attribute after adding rows and columns does not effect any existing cell’s attributes. When a new cell is drawn, it is the cell’s own attribute that is used.

The Visible attribute sets a flag as TRUE indicating whether the component is shown, or as FALSE indicating whether the component is hidden.

The VisibleRows attribute specifies the number of visible rows of the Table.

**Example 16: 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.
Figure 5-12: UG Tables

(Dialog uguitableexample
  (Components
    (SubLayout TableLayout)
    (SubLayout ButtonLayout)
    (OptionMenu 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"
  "Epsilon")
(LargeTable.ColumnLabels  "One"
  "Two"
  "Three"
  "Four")
(LargeTable.ShowGrid True)
(.AttachLeft True)
(.AttachRight True)
(.AttachTop True)
(.AttachBottom True)
(.Layout
  (Grid (Rows 1) (Cols 1)
   LargeTable)
 }
)
)

(TableLayout LargeTable
 (Components
   (PushButton BaseButton)
   (CheckButton BaseCheckButton)
   (InputPanel BaseInputPanel)
 )
)

(Resources
   (BaseButton.Label "Table Button")
   (BaseCheckButton.Label "Table Check Button")
   (BaseInputPanel.Value "Table Input Panel")
 )
)

(Layout ButtonLayout
 (Components
5 - 86 ProTOOLKIT User's Guide
User Interface Components

Table Inquiry Functions

Functions introduced

- `ProUITableIsEnabled()`
- `ProUITableIsVisible()`
- `ProUITableIsAutohighlightEnabled()`

The function `ProUITableIsEnabled()` determines whether or not table is sensitive to user input.

The function `ProUITableIsVisible()` determines whether or not table is visible.

The function `ProUITableIsAutohighlightEnabled()` determines whether or not autohighlighting is enabled in the table.
Table Modification Functions

Functions introduced

- `ProUITableColumnsSet()`
- `ProUITableVisiblerowsSet()`
- `ProUITableLockedcolumnsSet()`
- `ProUITableLockedrowsSet()`
- `ProUITableMincolumnsSet()`
- `ProUITableMinrowsSet()`
- `ProUITableHelptextSet()`
- `ProUITableTruncatetextSet()`
- `ProUITableEnable()`
- `ProUITableDisable()`
- `ProUITableShow()`
- `ProUITableHide()`
- `ProUITableAutohighlightEnable()`
- `ProUITableAutohighlightDisable()`
- `ProUITableComponentCopy()`
- `ProUITableComponentDelete()`

The function `ProUITableColumnsSet()` sets the width of the table.

The function `ProUITableVisiblerowsSet()` sets the number of rows visible in the table.

The functions `ProUITableLockedcolumnsSet()` and `ProUITableLockedrowsSet()` sets the number of columns or rows to repeat at the top of the table while scrolling.

The function `ProUITableMincolumnsSet()` sets the minimum width of the table.

The function `ProUITableMinrowsSet()` sets the minimum number of rows to be displayed in the table.

The function `ProUITableHelptextSet()` sets the help text for the table. This will be used by all elements of the table except for cells assigned in a help text value using `ProUITableCellHelpTextSet()`.
The function **ProUITableTruncateLabelSet()** identifies whether or not to truncate the label in a newly created cell based on cell width.

The functions **ProUITableEnable()** and **ProUITableDisable()** sets the table to be sensitive or insensitive to user input.

The functions **ProUITableShow()** and **ProUITableHide()** makes the table visible or invisible.

The functions **ProUITableAutohighlightEnable()** and **ProUITableAutohighlightDisable()** turns on or off the autohighlighting of rows when cells are selected.

The function **ProUITableComponentCopy()** copies a predefined component and places it in the table. The component is not displayed until it is assigned to a table cell using **ProUITableCellComponentNameSet()**. The component properties can be updated as needed and is displayed at a later time.

The function **ProUITableComponentDelete()** deletes a component from the table.

**Example 17: 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);
}
/*====================================================================*

User Interface Components
5 - 89
```
FUNCTION: UserUITableRedesignExample
PURPOSE: Change the table columns and rows before displaying the dialog

```c
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",
```

```c
    "LargeTable",
    new_column_count, new_column_names ,
    new_column_labels);
```
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 Row Functions

Functions introduced

• ProUITableRownamesGet()
• ProUITableRownamesSet()
• ProUITableRowlabelsGet()
• ProUITableRowlabelsSet()
• ProUITableRowRename()
• ProUITableRowIndexGet()
• ProUITableRowLabelGet()
• ProUITableRowLabelSet()
• ProUITableRowsInsert()
• ProUITableRowsDelete()
• ProUITableRowlabelsSet()

The functions ProUITableRownamesGet() and ProUITableRownamesSet() gets or sets the name of the rows in the table.

The function ProUITableRowlabelsGet() gets the name of the user-visible row labels in the table and the function
ProUITableRowlabelsSet() sets the name of user-visible row labels in the table.

The function ProUITableRowRename() renames the table row.
The function `ProUITableRowIndexGet()` gets the index of the table row with the given name.

The functions `ProUITableRowLabelGet()` and `ProUITableRowLabelSet()` gets and sets the user-visible label for table row.

The functions `ProUITableRowsInsert()` and `ProUITableRowsDelete()` inserts and deletes one or more rows from the table.

The function `ProUITableRowlabelsSet()` sets the names of the user-visible row labels in the table.

### Table Column Functions

Functions introduced

- `ProUITableColumnnamesGet()`
- `ProUITableColumnnamesSet()`
- `ProUITableColumnlabelsGet()`
- `ProUITableColumnlabelsSet()`
- `ProUITableColumnresizingsSet()`
- `ProUITableColumnwidthsSet()`
- `ProUITableColumnRename()`
- `ProUITableColumnIndexGet()`
- `ProUITableColumnLabelGet()`
- `ProUITableColumnLabelSet()`
- `ProUITableColumnWidthGet()`
- `ProUITableColumnWidthSet()`
- `ProUITableColumnResizingFactorGet()`
- `ProUITableColumnResizingFactorSet()`
- `ProUITableColumnsInsert()`
- `ProUITableColumnsDelete()`
- `ProUITableResetColumnWidth()`

The functions `ProUITableColumnnamesGet()` and `ProUITableColumnnamesSet()` gets and sets the names of the columns in the table.
The functions ProUITableColumnlabelsGet() and ProUITableColumnlabelsSet() gets and sets the user-visible labels of the columns in the table.

The function ProUITableColumnresizingsSet() sets the resizing policies of the columns in the table.

The function ProUITableColumnwidthsSet() sets the widths of the column in the table in characters.

The function ProUITableColumnRename() renames a table column.

The function ProUITableColumnIndexGet() gets the column index for a given column.

The functions ProUITableColumnLabelGet() and ProUITableColumnLabelSet() gets and sets the user-visible label for the column.

The functions ProUITableColumnWidthGet() and ProUITableColumnWidthSet() gets and sets the width of the table in the column.

The functions ProUITableColumnResizingFactorGet() and ProUITableColumnResizingFactorSet() sets and gets the resizing factor of the specified column of the table.

The functions ProUITableColumnsInsert() and ProUITableColumnsDelete() inserts or deletes one or more columns from the table.

The function ProUITableResetColumnWidth() sets the column width to default.
Table Cell Functions

Functions introduced

- ProUITableCellLabelGet()
- ProUITableCellLabelSet()
- ProUITableIsCellSensitive()
- ProUITableCellEnable()
- ProUITableCellDisable()
- ProUITableCellComponentCopy()
- ProUITableCellComponentNameGet()
- ProUITableCellComponentNameSet()
- ProUITableCellComponentDelete()

The functions ProUITableCellLabelGet() and ProUITableCellLabelSet() gets or sets the text contained in the table cell.

The function ProUITableIsCellSensitive() determines whether the cell in the table is sensitive to user input.

The functions ProUITableCellEnable() and ProUITableCellDisable() sets the table cells to be sensitive or insensitive to user input.

The function ProUITableCellComponentCopy() copies a predefined component and places it in the table in the designated cell. The component is displayed in this cell.

The functions ProUITableCellComponentNameGet() and ProUITableCellComponentNameSet() gets and sets the component name contained in the table cell.

The function ProUITableCellComponentDelete() removes the component contained in the table cell.

Example 18: 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 16: 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.

(Dialog uguitablecomponents

  (Components

    (PushButton                     ExternalButtonToCopy)

  )

  (Resources

    (ExternalButtonToCopy.Label     "From other dialog..."
     (.Layout

      (Grid (Rows 1) (Cols 1)

        ExternalButtonToCopy

      )

     )

    )

  )

)
/*-----------------------------------*/
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", "CopyOfOptionMenu");

ProUITableCellComponentNameSet (UGUITABLEDLG, "LargeTable", 
    "B", "2", 
    "CopyOfOptionMenu");

ProUIOptionmenuShow (UGUITABLEDLG, "CopyOfOptionMenu");

/*-----------------------------------*/
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);
}

Table Selection Functions

Functions introduced

- ProUITableSelectionpolicySet()
- ProUITableColumnselectionpolicySet()
- ProUITableSelectednamesGet()
- ProUITableSelectednamesSet()
- ProUITableSelectedcolumnnamesGet()
- ProUITableSelectedcolumnnamesSet()
- ProUITableAnchorCellGet()
- ProUITableAnchorCellSet()
- ProUITableFocusCellGet()
- ProUITableFocusCellSet()

The function ProUITableSelectionpolicySet() sets the table selection policy.

The function ProUITableColumnselectionpolicySet() sets the table column selection set.

The functions ProUITableSelectednamesGet() and ProUITableSelectednamesSet() gets or sets the selected cells in the table. The values to this function is an array of selected names. This array should consist of twice the number of selections, organized in pairs. The array is in the following format:

element [ 0, 2, 4...] = row name of the selected cell

element [ 1, 3, 5...] = column name of the selected cell

The functions ProUITableSelectedcolumnnamesGet() and ProUITableSelectedcolumnnamesSet() gets and sets the selected column from the table.
The function `ProUITableAnchorCellGet()` and `ProUITableAnchorCellSet()` gets or sets the coordinates of the table selection anchor cell.

`ProUITableFocusCellGet()` and `ProUITableFocusCellSet()` gets and sets the coordinates of the table selection focus cell.

Example 16: UG Tables Resource file is the resource file for the following example.

**Example 20: To Access Selected Names Array from Tables**

```c
#include <ProUI.h>
#include <ProUIDialog.h>
#include <ProUIPushbutton.h>
#include <ProUITable.h>

#define UGUITABLEDLG "uguitableexample"

/*====================================================================*\nFUNCTION: 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);
}

/*====================================================================*\nFUNCTION: 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",

User Interface Components

```c
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);
}

Table Action Functions

Functions introduced

- ProUITableActivateActionSet()
- ProUITableColumnselectActionSet()
- ProUITableArmActionSet()
- ProUITableDisarmActionSet()
- ProUITableSelectActionSet()
- ProUITableFocusinActionSet()
- ProUITableFocusoutActionSet()

The function **ProUITableActivateActionSet()** sets 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.

The function **ProUITableColumnselectActionSet()** sets the column selection action for the table. This function is called when the user changes the currently selected table.

The function **ProUITableArmActionSet()** sets the arm action for a table. This function is called when the user changes the selection anchor and focus cell.

The function **ProUITableDisarmActionSet()** sets the disarm action for a table. This function is called when the user changes the selection focus cell in the table.
The function `ProUITableSelectActionSet()` sets the select action for a table. This function is called when the user changes the selected cells in the table.

The functions `ProUITableFocusinActionSet()` and `ProUITableFocusoutActionSet()` sets the focus in or focus out action for a table. These functions are called when the user moves the cursor onto or off the table using the mouse or the <TAB> key.

**Programming with TextAreas**

The next sections describe programming with TextAreas

**Text Area Attributes**

The Columns attribute specifies the starting width of the text area in terms of the number of characters.

The Editable attribute specified whether the text area is editable by the user.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Values</th>
<th>Pro/TOOLKIT Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>.Columns</td>
<td>integer</td>
<td>ProUITextareaColumnsSet()</td>
</tr>
<tr>
<td>.Editable</td>
<td>TRUE(default),FALSE</td>
<td>ProUITextareaEditable() ProUITextareaReadOnly()</td>
</tr>
<tr>
<td>.Maxlen</td>
<td>integer</td>
<td>ProUITextareaMaxlenSet()</td>
</tr>
<tr>
<td>.MinColumns</td>
<td>integer</td>
<td>—</td>
</tr>
<tr>
<td>.MinRows</td>
<td>integer</td>
<td>ProUITextareaMinrowsSet()</td>
</tr>
<tr>
<td>.Rows</td>
<td>integer</td>
<td>ProUITextareaRowsSet()</td>
</tr>
<tr>
<td>.Value</td>
<td>wide string</td>
<td>ProUITextareaValueSet() ProUITextareaValueGet()</td>
</tr>
<tr>
<td>.Visible</td>
<td>TRUE,FALSE</td>
<td>ProUITextareaShow() ProUITextareaHide() ProUITextareaIsVisible()</td>
</tr>
</tbody>
</table>

The MaxLen attribute specified the maximum length of a line of text that can be entered into the text area.

The MinColumns and MinRows attributes specify the minimum size the text area can be if the user resizes the dialog.

The Value attribute specifies the text contents of the text area. The string should be enclosed in double quotes. If the text contains more than one line, insert a newline inside the text string.
The Visible attribute specifies whether or not the component is visible at the present time.

**TextArea Functions**

Functions introduced:

- `ProUITextareaInputActionSet()`
- `ProUITextareaActivateActionSet()`
- `ProUITextareaFocusinActionSet()`
- `ProUITextareaFocusoutActionSet()`
- `ProUITextareaEditable()`  
- `ProUITextareaReadOnly()`  
- `ProUITextareaHelpsetText()`  
- `ProUITextareaColumnsSet()`  
- `ProUITextareaMaxlenSet()`  
- `ProUITextareaMinrowsSet()`  
- `ProUITextareaRowsSet()`  
- `ProUITextareaValueSet()`  
- `ProUITextareaValueGet()`  
- `ProUITextareaEnable()`  
- `ProUITextareaDisable()`  
- `ProUITextareaShow()`  
- `ProUITextareaHide()`  
- `ProUITextareaisVisible()`

The function `ProUITextareaInputActionSet()` sets the callback to be called each time the user edits text in the text area.

The function `ProUITextareaActivateActionSet()` sets the callback to be called when the user double-clicks in the text area.

The function `ProUITextareaFocusinActionSet()` and `ProUITextareaFocusoutActionSet()` sets the callback to be called when the user moves the focus onto or off of the text area using the mouse or <Tab> key.
The functions `ProUITextareaEditable()` and `ProUITextareaReadOnly()` determine whether the text in the text area can be edited by the user. If not, the background of the text area is shown in grey.

The functions `ProUITextareaHelpTextSet()` through `ProUITextareaRowsSet()` set attributes that can also be set in the resource file, as described in the section TextArea attributes.

The functions `ProUITextareaValueSet()` and `ProUITextareaValueGet()` set and get the text in the text area. This is a single wide string, in which the lines of text are separated by a wide return character (wchar_t)\n.

The functions `ProUITextareaEnable()` and `ProUITextareaDisable()` control whether the button is sensitive to user input.

The functions `ProUITextareaShow()` and `ProUITextareaHide()` control whether the button is visible. The function `ProUITextareaisVisible()` returns whether or not the textarea is visible.

### Programming with Thumbwheels

The next sections describe programming with Sliders.

### Thumbwheel Attributes

This section describes attributes for Thumbwheels.

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Values</th>
<th>Pro/TOOLKIT Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>InfiniteRange</td>
<td>TRUE, FALSE</td>
<td>-</td>
</tr>
<tr>
<td>Integer</td>
<td>integer</td>
<td><code>ProUIThumbwheelIntegerGet()</code></td>
</tr>
<tr>
<td>Length</td>
<td>integer</td>
<td>-</td>
</tr>
<tr>
<td>Maxinteger</td>
<td>integer</td>
<td><code>ProUIThumbwheelMaxintegerSet()</code></td>
</tr>
<tr>
<td>Mininteger</td>
<td>integer</td>
<td><code>ProUIThumbwheelMinintegerSet()</code></td>
</tr>
<tr>
<td>Orientation</td>
<td>TRUE, FALSE</td>
<td>-</td>
</tr>
<tr>
<td>Tracking</td>
<td>TRUE, FALSE</td>
<td>-</td>
</tr>
<tr>
<td>UnitsPerRotation</td>
<td>integer</td>
<td><code>ProUIThumbwheelUnitsPerRotationSet()</code></td>
</tr>
</tbody>
</table>
The attribute InfiniteRange should be set if the thumbwheel should not obey any maximum or minimum value.

The attribute Integer sets the initial value for the thumbwheel.

The attribute Length sets the width of the thumbwheel.

The attributes Maxinteger and Mininteger set the range of valid integer values, unless InfiniteRange is TRUE.

The Orientation attribute should be False (the default) for a horizontal thumbwheel, and True for a vertical one.

Tracking is TRUE if the action callback should be called every time the thumbwheel moves, and FALSE if it should be called only when the user releases the mouse button after changing the value.

UnitsPerRotation sets the sensitivity of the thumbwheel, that is, the value by which the integer changes for each complete revolution of the wheel.

**Thumbwheel Functions**

Functions introduced:

- `ProUIThumbwheelHelpsetTextSet()`
- `ProUIThumbwheelIntegerSet()`
- `ProUIThumbwheelIntegerGet()`
- `ProUIThumbwheelMinintegerSet()`
- `ProUIThumbwheelMaxintegerSet()`
- `ProUIThumbwheelUnitsperrotationSet()`
- `ProUIThumbwheelUpdateActionSet()`
- `ProUIThumbwheelEnable()`
- `ProUIThumbwheelDisable()`
- `ProUIThumbwheelIsEnabled()`
- `ProUIThumbwheelShow()`
- `ProUIThumbwheelHide()`
- `ProUIThumbwheelIsVisible()`

The functions named after attributes are used to set those attributes dynamically.

The function `ProUIThumbwheelUpdateActionSet()` sets the callback function which is to be called when the thumbwheel moves.
The functions `ProUThumbwheelEnable()` and `ProUThumbwheelDisable()` control whether the button is sensitive to user input. The function `ProUThumbwheelIsEnabled()` returns whether or not thumbwheel is sensitive.

The functions `ProUThumbwheelShow()` and `ProUThumbwheelHide()` control whether the button is visible. The function `ProUThumbwheelIsVisible()` returns whether or not thumbwheel is visible.
This chapter describes Pro/TOOLKIT modes, models, and model items.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modes</td>
<td>6 - 2</td>
</tr>
<tr>
<td>Models</td>
<td>6 - 2</td>
</tr>
<tr>
<td>Model Items</td>
<td>6 - 7</td>
</tr>
<tr>
<td>Version Stamps</td>
<td>6 - 10</td>
</tr>
<tr>
<td>Layers</td>
<td>6 - 12</td>
</tr>
<tr>
<td>Layouts</td>
<td>6 - 17</td>
</tr>
<tr>
<td>Visiting Displayed Entities</td>
<td>6 - 18</td>
</tr>
</tbody>
</table>
Modes

Functions introduced:

- `ProModeCurrentGet()`
- `ProModeCurrentGet()`

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).

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:

```c
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()
- ProDrawingFromTmpltCreate()
- 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 ProDrawingFromTmpltCreate().

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 Using Notify for more data on using callback functions.

Identifying Models

Functions introduced:

- `ProMdlNameGet()`
- `ProMdlTypeGet()`
- `ProMdlInit()`
- `ProMdlIdGet()`
- `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 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 `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.

```c
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()**

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.

**File Management Operations**

Functions introduced:

- ProMdlRetrieve()
- ProMdlMultipleRetrieve()
- ProSolidRetrievalErrorsGet()
- ProMdlSave()
- ProMdlErase()
- ProMdlEraseAll
- ProMdlCopy()
- ProMdlFileCopy()
- 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.

• **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.

---

### Model Items

Functions introduced:

• **ProModelitemByNameInit()**
• **ProModelitemInit()**
• **ProModelitemMdlGet()**
• **ProModelitemDefaultnameGet()**
• **ProModelitemNameGet()**
• **ProModelitemNameSet()**
• **ProModelitemNameCanChange()**
• **ProModelitemUsernameDelete()**
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 objects, such as “version stamp,” retain their own object types.)

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

The typedef for the \emph{ProModelitem} data handle is as follows:

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

The function \emph{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.

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.

Example 2: Renaming a Selected Surface

This example shows how to use the functions ProModelItemNameGet() and ProModelItemNameSet(). See the Geometry chapter for an explanation of ProSurface and its functions.

```c
/*================================================================*
FUNCTION: UserSurfRename()
PURPOSE:  Rename the selected surface.
/*================================================================*/
int UserSurfRename()
{
  int            sel_count;
  ProError       status;
  char           name[PRODEV_NAME_SIZE];
  ProModelitem   p_mdl_item;
  ProFileName    msgfile;
  ProName        w_name;
  ProSelection   *psels = NULL;
  /*======================================================*/
  Prompt the user to select the surface.
```
ProStringToWstring (msgfile, "msg_ug9.txt");
status = ProMessageDisplay (msgfile, "UG9 Select Surface to Rename;");  
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 = ProModelitemNameGet (&p_mdl_item, w_name);

/*
   Display the current name or "NONE" if the surface is not named.
*/
if (status != PRO_TK_NO_ERROR)  
sprintf (name, "NONE");  
else  
{  
   ProWstringToString (name, w_name);
}
status = ProMessageDisplay (msgfile, "UG9 Enter Name [%0s]:", name);
status = ProMessageStringRead (PRODEV_NAME_SIZE, w_name);
if (status == PRO_TK_NO_ERROR)
status = ProModelitemNameSet (&p_mdl_item, w_name);

ProMessageClear();
return ((int)status);
}

Version Stamps

Functions introduced:

• ProMdlVerstampGet()
• ProFeatureVerstampGet()
• ProVerstampAlloc()
• ProVerstampFree()
• ProVerstampStringGet()
• ProVerstampStringFree()
• ProStringVerstampGet()
• ProVerstampEqual()
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.

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

Functions introduced:

- ProMdlLayerGet()
- ProLayerCreate()
- ProLayerDelete()
- ProLayerItemsGet()
- ProLayerItemInit()
- ProDwgLayerItemInit()
- ProLayerItemAdd()
- ProLayerItemRemove()
- ProLayerItemLayersGet()
- ProLayerDisplaystatusGet()
- ProLayerDisplaystatusSet()
- ProDwgLayerDisplaystatusGet()
- ProDwgLayerDisplaystatusSet()
- ProLayerDisplaystatusSave()
- ProLayerDefLayerSet()
- ProLayerDefLayerGet()
- ProLayerViewDependencySet()
- ProLayerViewDependencyGet()
- ProMdlLayerVisit()

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.

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.
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. The function reuses the memory for the array on subsequent calls.

Pro/TOOLKIT also includes functions to modify layers. 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. The enumerated type **ProLayerDisplay** contains the possible display statuses. To set the display status of a layer, call the **ProLayerDisplaystatusSet()** function. Note that the function does not repaint the model after it sets the display status.

Unique functions are required to retrieve and set the status of layers in drawing models. **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**().

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.

**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.
#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 ("ublank6", "ublank6.mnu", &menu_id);

    err = ProMenubuttonActionSet ("ublank6", "Part",
                                   UserLayerItemAction, NULL, PRO_LAYER_PART);

    err = ProMenubuttonActionSet ("ublank6", "Feature",
                                   UserLayerItemAction, NULL, PRO_LAYER_FEAT);

    err = ProMenubuttonActionSet ("ublank6", "Curve",
                                   UserLayerItemAction, NULL, PRO_LAYER_CURVE);

    err = ProMenubuttonActionSet ("ublank6", "Quilt",
                                   UserLayerItemAction, NULL, PRO_LAYER_QUILT);

    err = ProMenubuttonActionSet ("ublank6", "Point",
                                   UserLayerItemAction, NULL, PRO_LAYER_POINT);

    err = ProMenubuttonActionSet ("ublank6", "ublank6",
                                   UserLayerItemAction, NULL, -1);

    ProMenuCreate (PROMENUTYPE_MAIN, "ublank6", &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");
/*-----------------------------------------------------------------*\nGet 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);
}

/*-----------------------------------------------------------------*\nGet the layer name from the user.
\*-----------------------------------------------------------------*/
ProMessageDisplay (msg_file, "USER %0s",
                    "Enter name of new layer: ");
ProMessageStringRead (MAX_BUFFER_LENGTH, layer_name);

/*-----------------------------------------------------------------*\nCreate 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_CUUILT  : option = "dtmqlt"  ; break;
case PRO_LAYERPOINT    : option = "point"   ; break;
}
while (ProSelect (option, 1, NULL, NULL, NULL, &sel, &nsel ) ==
   PRO_TK_NO_ERROR && nsel > 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)
```
```c
{  
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);
}
This chapter describes how to access solids, parts, and their contents.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solid Objects</td>
<td>7 - 2</td>
</tr>
<tr>
<td>Part Objects</td>
<td>7 - 9</td>
</tr>
<tr>
<td>Material Properties</td>
<td>7 - 10</td>
</tr>
<tr>
<td>Notes</td>
<td>7 - 15</td>
</tr>
</tbody>
</table>
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()**
- **ProSolidCsystVisit()**
- **ProSolidFeatstatusGet()**
- **ProSolidFeatstatusSet()**

  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 datum surfaces in a part or an assembly, but not the quilts.
  
  - **ProSolidAxisVisit()**—Visits all the axes in a part or an assembly.
• **ProSolidCssysVisit()**—Visits all the coordinate system datums in a part or an assembly.

For a detailed explanation of these functions, see the Geometry chapter.

The function **ProSolidFeatstatusGet()** enables you to find the identifiers and statuses of features in a solid, and **ProSolidFeatstatusSet()** enables you to set the order and statuses of the features. Both of these functions are described in detail in the Basic Access to Features chapter.

### 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()**.

```c
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, PRO_PART, &part);
    /**----------------------------------------*
    Check that the part was retrieved.
    \*----------------------------------------*/
    if (err != PRO_TK_NO_ERROR)
```

Solids and Parts 7 - 3
Regenerating a Solid

Function introduced:

- **ProSolidRegenerate()**

  The function **ProSolidRegenerate()** regenerates the specified solid. It does not perform recursive regeneration in assemblies. (To do so, use the function **ProAsmcompRegenerate()**, described in the Assemblies chapter.) 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 must be used only in interactive applications. If this option is not included, the user interface does not update if regeneration is successful. Use **ProWindowRepaint()** and **ProTreetoolRefresh()** to perform the update if needed.
  - **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.
Example 2: Combining Regeneration Flags

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

```c
#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;
}
```

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 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 coordinate system
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.

```c
/*================================================================*\
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;
  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, &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()**

These functions enable you to set and retrieve the relative accuracy
of the specified part. The relative part accuracy is the same as using
the Pro/ENGINEER command **Set Up, Accuracy**.

These functions apply to parts only. For assemblies, you must walk
through all the parts.

The relevant geometry epsilon is derived as follows:

\[ \text{geometry_epsilon} = \text{model_size} \times \text{relative_part_accuracy} \]

Solid Units

Functions introduced:

- **prodb_get_model_units()**
- **prodb_set_model_units()**

These functions correspond exactly to the Pro/ENGINEER
command **Set Up, Model Units**. There are two types of unit, identified
by an integer argument that can have one of the following values:

- LENGTH_UNIT
- MASS_UNIT
There is also a subtype that is the actual unit used. If the unit type is LENGTH_UNIT, the subtype can have one of the following values:

- UNIT_CM
- UNIT_FOOT
- UNIT_INCH
- UNIT_M
- UNIT_MM
- UNIT_USER

If the unit type is MASS_UNIT, the subtype can have one of the following values:

- UNIT_OUNCE
- UNIT_POUND
- UNIT_TON
- UNIT_GRAM
- UNIT_KILOGRAM
- UNIT_TONNE
- UNIT_USER

Use the value UNIT_USER to set a unit that has not been previously defined in Pro/ENGINEER. If you set a new unit, you must supply two other arguments: the name of the unit and the conversion factor. The conversion factor for a length unit is the number of inches it represents. The conversion factor for a mass unit is the number of pounds it represents.

**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).
- **coor_sys_inertia[3][3]**—The inertia matrix.
- **coor_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).

### 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.

### 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.
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.

Material Properties

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

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

Material Objects

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
- `ProMaterialData`—A structure that contains the material properties most common in engineering (for example, Young’s modulus, mass density, and thermal conductivity)

Note that in releases prior to Release 19, the `ProMaterialData` object was known as the `ProMaterialProps` object.
Accessing Material Data

Functions introduced:

- **ProMaterialCreate()**
- **ProMaterialDataGet()**
- **ProMaterialDataSet()**
- **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 function **ProMaterialDataGet()**, given the part material, obtains the properties of a specific material. To set material properties within the Pro/ENGINEER database, pass to **ProMaterialDataSet()** a **ProMaterialData** object that contains the new properties.

Remember that material data is, in general, stored in a part's database and not in a separate file on disk. The function **ProPartMaterialsGet()** obtains an array that contains the 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 a part database, call the function **ProMaterialDelete()**.

The current material of a part determines which material properties will be used in some computational analyses of that part (such as mass properties). 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()**.
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. Note that this function does not automatically append the .mat extension to the material file name.

The function ProMaterialfileRead() reads from a material (.mat) file the properties of the material with the specified name. This function constructs the name of the material file from the material name and the .mat extension. 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. In both cases, the current material becomes the material contained in the material file.

Example 4: Reading Material Properties from a File

This example shows how to use Pro/TOOLKIT functions to read a material file and create a new material.

```c
/*--------------------------------------------------------------------*
* Pro/TOOLKIT includes
*/
#include <ProToolkit.h>
#include <ProMdl.h>
#include <ProUtil.h>
#include <TestError.h>
#include <ProMaterial.h>
#include <ProSolid.h>

Application data

#define USER_MAX_BUFFER 20

FUNCTION: UserMaterial()
PURPOSE: Read a material from file, create a new material, and compute mass properties for both materials.
```
ProError UserMaterial (ProMdl part)
{
    FILE     *fp;
    ProMdlType        mdl_type;
    ProCharLine       err_str, msg_str;
    ProName           w_mdl_name, w_matl_name, w_new_matl_name;
    ProCharName       mdl_name, matl_name, str;
    ProCharName       new_matl_name = "new_material";
    ProName           msg_file;
    ProMaterial       matl, new_matl;
    ProMaterialdata   matl_data, new_matl_data;
    ProMassProperty   mass_prop;
    int               status;
    ProFileName       new_matl_file;
}/*====================================================================*/

Set up the message file.

ProStringToWstring (msg_file, "testmsg.txt");
/*--------------------------------------------------------------------*/

Check to make sure you have a part.

status = ProMdlTypeGet (part, &mdl_type);
if (mdl_type != PRO_PART)
{
    ProMessageDisplay (msg_file, "USER %0s",
                       "ERROR: Current model is not a part. ");
    return (-1);
}
/*--------------------------------------------------------------------*/

Get the material name from the user.

status = ProMessageStringRead (USER_MAX_BUFFER, w_matl_name);
if (status == PRO_TK_MSG_USER_QUIT)
{
    ProMessageDisplay (msg_file, "USER %0s",
                       "Need a material name to continue. Aborting.");
    return (status);
}

ProWstringToString (matl_name, w_matl_name);
/*--------------------------------------------------------------------*/

Read the material from the file.

status = ProMaterialfileRead (part, w_matl_name);
if (status != PRO_TK_NO_ERROR)
{
    sprintf (err_str, "Error opening material file %s.",

matl_name, ".mat");
ProMessageDisplay (msg_file, "USER %0s", err_str);
return (status);
}

Get the current material properties. (The following function calls are easier than reading the text of the material file.)

status = ProMaterialCurrentGet (part, &matl);
status = ProMaterialDataGet (&matl, &matl_data);

Compute the mass properties.

status = ProSolidMassPropertyGet (part, NULL, &mass_prop);

sprintf (msg_str, "Part mass with material %s: %6.2e", matl_name, mass_prop.mass);
ProMessageDisplay (msg_file, "USER %0s", msg_str);

Create a new material that is identical to matl_name, but with a slightly higher density. Set to be the current material.

ProStringToWstring (w_new_matl_name, new_matl_name);
status = ProMaterialCreate (part, w_new_matl_name, &new_matl_data, &new_matl);
new_matl_data = matl_data;
new_matl_data.mass_density = matl_data.mass_density * 1.10;
status = ProMaterialDataSet (&new_matl, &new_matl_data);
status = ProMaterialCurrentSet (&new_matl);

sprintf (msg_str, "Current material set to %s.", new_matl_name);
ProMessageDisplay (msg_file, "USER %0s", msg_str);

Write the new material to a file.

strcat (new_matl_name, ".mat");
ProStringToWstring (new_matl_file, new_matl_name);
status = ProMaterialfileWrite (&new_matl, new_matl_file);

sprintf (msg_str, "New material written to file %s", new_matl_name);
ProMessageDisplay (msg_file, "USER %0s", msg_str);

Compute the mass properties with the new material.

status = ProSolidMassPropertyGet (part, NULL, &mass_prop);

sprintf (msg_str, "Part mass with material %s: %6.2e", new_matl_name, mass_prop.mass);
ProMessageDisplay (msg_file, "USER %0s", msg_str);
return (0);
Notes

The functions in this section enable you to access the notes created in Pro/ENGINEER using the command Setup, Notes.

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:

- ProNoteCreate()
- ProNoteDelete()

The function ProNoteCreate() 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. Before the note can be displayed, its location and leaders, if any, must be defined, as explained in the section Note Placement.

The function ProNoteDelete() deletes the note specified by its ProNote object.

Note Text

Functions introduced:

- ProNoteLinelistGet()
- ProNoteLinelistSet()

The function ProNoteLinelistGet() retrieves the text of the specified note. The function ProNoteLinelistSet() modifies the text of an existing note by copying from the specified ProLineList.
Visiting Notes
Function introduced:

• **ProMdlNoteVisit()**

  The function **ProMdlNoteVisit()** enables you to visit all the notes in the specified solid or drawing.

Note Display
Function introduced:

• **ProNoteDisplay()**

  The function **ProNoteDisplay()** enables you to display a note specified by the **ProNote** argument. It takes a second argument of type **ProDrawMode** (defined in **ProGraphics.h**) that provides the option of drawing in complement mode.

Note Owner
Function introduced:

• **ProNoteOwnerGet()**

  The function **ProNoteOwnerGet()** retrieves the owner of the specified note.

Note Placement
Functions introduced:

• **ProNoteAttachAlloc()**
• **ProNoteAttachRelease()**
• **ProNoteAttachFreeGet()**
• **ProNoteAttachFreeSet()**
• **ProNoteAttachAddend()**
• **ProNotePlacementGet()**
• **ProNotePlacementSet()**
• **ProNoteURLGet()**
• **ProNoteURLSet()**
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 **ProNoteAttachAlloc()** allocates a **ProNoteAttach** object for a note attachment, whereas the function **ProNoteAttachRelease()** releases the allocated opaque handle.

The function **ProNoteAttachFreeGet()** retrieves the location of an attachment point. To set the location of an attachment point, call the function **ProNoteAttachFreeSet()**. The attachment point is described in solid coordinates.

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**

You can retrieve the placement information for the specified note in the form of a **ProNoteAttach** object using function **ProNotePlacementGet()**. To place a note at a specific location, call the function **ProNotePlacementSet()**.

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 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()**

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, and mirror.

**Note:** The system uses the value –1.0 for properties that use default values, or that have not been set yet.

**Example 5: Creating a Model Note**

This example shows how to create a note with default text properties.

```c
/*-----------------------------------------------*/
Pro/TOOLKIT includes

#include <ProToolkit.h>
#include <ProSelection.h>
#include <ProObjects.h>
```
#include <ProSolid.h>
#include <ProMdl.h>
#include <ProMessage.h>
#include <ProNote.h>
#include <ProArray.h>
#include <ProGraphic.h>

/*----------------------------------------------------------------*
Application includes
*/
#include "ProUtil.h"
#include "TestError.h"
#include "UtilMath.h"
#include "UtilMatrix.h"

/*----------------------------------------------------------------*
Application data
*/
static ProName msg_file;

/*================================================================*
FUNCTION: UserSolidNoteCreate()
PURPOSE:  Example of note creation for solids
*/
int UserSolidNoteCreate()
{
    #define MAX_NOTE_LEN PRO_LINE_SIZE

    int             err;
    ProSelection   *p_sel;                /* for selection */
    int             n_sel;
    ProModelitem    note_owner;           /* owner of the note */
    ProLinelist     line_arr;             /* list of note text lines */
    ProMdl          model;                /* current model */
    ProLine         w_notetext;           /* line of note text */
    ProNote         note;                 /* note object */
    ProNoteAttach   note_attach;          /* attach information for
                                    note */
    int             i;
    ProMouseButton  button;
    ProPoint3d      note_pos_scrn;        /* note position in screen
                                    coordinates */
    ProPoint3d      outline[2];           /* points delimiting the
                                    model outline */
    ProPoint3d      note_pos;             /* note position, in model
                                    coordinates */
    ProMatrix       vmatrix, inv_vmatrix; /* view matrix and inverse */

    ProStringToWstring (msg_file, "msg_ugfund.txt");

    /* Get the current model. */
    err = ProMdlCurrentGet (&model);

    /* Application includes */
    /* Application data */
if (err != PRO_TK_NO_ERROR)
{
    ProMessageDisplay (msg_file, "USER %s",
        "Error getting current model.");
    return (err);
}

/*---------------------------------------------------------------*\
Get the note text.
\*---------------------------------------------------------------*/
ProMessageDisplay (msg_file, "USER %s", "Enter note text.");
ProMessageStringRead (MAX_NOTE_LEN, w_notetext);
if (err != PRO_TK_NO_ERROR)
    return (err);

/*---------------------------------------------------------------*\
Fill the array of note lines.
\*---------------------------------------------------------------*/
err = ProArrayAlloc (0, sizeof(ProLine), 1, (ProArray*)&line_arr);
err = ProArrayObjectAdd ((ProArray*)&line_arr, PRO_VALUE_UNUSED,
   1, w_notetext);

/*---------------------------------------------------------------*\
Get the modelitem for the model.
\*---------------------------------------------------------------*/
err = ProMdlToModelitem (model, &note_owner);

/*---------------------------------------------------------------*\nCreate the note to be owned by the model.
\*---------------------------------------------------------------*/
err = ProNoteCreate (model, &note_owner, line_arr, &note);

/*---------------------------------------------------------------*\nGet the leaders and add to the attachment.
\*---------------------------------------------------------------*/
ProMessageDisplay (msg_file, "USER %s",
    "Select edges for leaders.");
err = ProSelect ("edge", -1, NULL, NULL, NULL, &p_sel, 
   &n_sel);
if (err != PRO_TK_NO_ERROR)
    return (err);

err = ProNoteAttachAlloc (&note_attach);
for (i = 0; i < n_sel; i++)
{
    err = ProNoteAttachAddend (note_attach, p_sel[i],
        PRO_NOTE_ATT_NONE);
}

/*---------------------------------------------------------------*\nGet the note position, in screen coordinates.
\*---------------------------------------------------------------*/
ProMessageDisplay (msg_file, "USER %s", "Select note location.");
err = ProMousePickGet (PRO_LEFT_BUTTON, &button, note_pos_scrn);
if (err != PRO_TK_NO_ERROR)
    return (err);
Get the view matrix (the transformation from model coordinates to screen coordinates) and invert.

```c
err = ProViewMatrixGet (model, NULL, vmatrix);
err = ProUtilMatrixInvert (vmatrix, inv_vmatrix);
```

Transform the note position from screen coordinates to model coordinates.

```c
ProUtilPointTrans (inv_vmatrix, note_pos_scrn, note_pos);
```

Set the note position in terms of model size parameters (see ProNote.h)

```c
for (i = 0; i < 3; i++)
{
    note_pos[i] = (note_pos[i] - outline[0][i]) / (outline[1][i] - outline[0][i]);
}
err = ProNoteAttachFreeSet (note_attach, note_pos[0], note_pos[1], note_pos[2]);
```

Display the note.

```c
err = ProNoteDisplay (&note, PRO_DRAW_NO_MODE);
```

Free the memory.

```c
err = ProNoteAttachRelease (&note_attach);
err = ProArrayFree ((ProArray*)&line_arr);
return (PRO_TK_NO_ERROR);
```
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).

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manipulating Windows</td>
<td>8 - 2</td>
</tr>
<tr>
<td>Solid Orientation</td>
<td>8 - 6</td>
</tr>
<tr>
<td>Graphics Colors and Line Styles</td>
<td>8 - 10</td>
</tr>
<tr>
<td>Displaying Graphics</td>
<td>8 - 14</td>
</tr>
<tr>
<td>Displaying Text</td>
<td>8 - 17</td>
</tr>
<tr>
<td>Display Lists</td>
<td>8 - 18</td>
</tr>
<tr>
<td>Getting Mouse Input</td>
<td>8 - 20</td>
</tr>
<tr>
<td>Cosmetic Properties</td>
<td>8 - 22</td>
</tr>
</tbody>
</table>
Manipulating Windows

This section describes how to manipulate windows using Pro/TOOLKIT. It is divided into the following subsections:

• Accessing Window Size
• 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()**

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.
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 \textbf{ProViewMatrixGet()} is not the same as the one you previously input to the function \textbf{ProViewMatrixSet()}, although its orientation is the same.
- Each row of the matrix you provide to \textbf{ProViewMatrixSet()} must have a length of 1.0, and the bottom row must be 0, 0, 0, 1.
- The matrix you provide to \textbf{ProViewMatrixSet()} must be normalized—it cannot include scaling or shift. Example 1: Saving Three Views shows how to normalize a matrix.

\section*{Storing Named Views}

Functions introduced:

- \textbf{ProViewStore()}
- \textbf{ProViewRetrieve()}
- \textbf{ProViewNamesGet()}

The \textbf{ProViewStore()} and \textbf{ProViewRetrieve()} functions enable you to save and use a named view of the solid. They are equivalent to the Pro/ENGINEER commands \texttt{View, Saved Views, Save, and View, Saved Views}, then selecting the view you want from the list of view names.

The function \textbf{ProViewNamesGet()} retrieves the names of the views in the specified solid.

\section*{Example 1: Saving Three Views}

This example shows how to set the view matrix and store a named view.

```c
#define UserMatrixNormalize

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).      |                          |

8 - 8 ProTOOLKIT User's Guide
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—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 ProColortype 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;
}

Graphics and Object Display
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.

```c
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);
}

**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' chapter.

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()`
• `ProTextFontNameToId()`

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.

```c
/*================================================================*/
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.)
    }
    /*-----------------------------------------------*/
    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.)
```
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:

- `prodb_get_surface_props()`
- `prodb_set_surface_props()`
- `prodb_unset_surface_props()`
- `prodb_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. Note that this function tessellates a single surface only.

Refer to the Tessellation section of the ‘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:

```c
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.
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 ‘Assemblies’ chapter for information on objects and actions specific to assemblies.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geometry Objects</td>
<td>9 - 2</td>
</tr>
<tr>
<td>Visiting Geometry Objects</td>
<td>9 - 3</td>
</tr>
<tr>
<td>Tessellation</td>
<td>9 - 14</td>
</tr>
<tr>
<td>Evaluating Geometry</td>
<td>9 - 15</td>
</tr>
<tr>
<td>Geometry Equations</td>
<td>9 - 16</td>
</tr>
<tr>
<td>Ray Tracing</td>
<td>9 - 24</td>
</tr>
<tr>
<td>Measurement</td>
<td>9 - 25</td>
</tr>
<tr>
<td>Geometry as NURBS</td>
<td>9 - 27</td>
</tr>
<tr>
<td>Interference</td>
<td>9 - 28</td>
</tr>
<tr>
<td>Sheet Metal</td>
<td>9 - 34</td>
</tr>
</tbody>
</table>
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
- ProCompcrv—Composite datum curve
- 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 9-1: ProGeomItem in Feature In Solid

![Diagram of ProSolid, ProFeature, ProGeomItem hierarchy]

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.
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 'Modes, 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 ‘Basic Access to Features’.
Visiting Solid Geometry

Functions introduced:

- ProSolidSurfaceVisit()
- ProSurfaceContourVisit()
- ProContourEdgeVisit()
- ProContourTraversalGet()
- 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. 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.

```c
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.

```c
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_HIGHLITE);
        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.

in the section Visiting Solid Geometry.

**Visiting Datum Curves**

Functions introduced:

- `ProCurveIdGet()`
- `ProCurveInit()`
- `ProCurveColorGet()`
- `ProCurveColorSet()`
- `ProCurveLinestyleGet()`
- `ProCurveLinestyleSet()`

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()*.

Functions *ProCurveColorGet/Set()* set or return the color of a *ProCurve* object.

Functions *ProCurveLinestyleGet/Set()* set or return the linestyle of a *ProCurve* object.

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. 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 *ProModelIndexInit()*.

To create a *ProPoint* object from the integer identifier, use *ProPointInit()*.
Tessellation

Functions introduced:

- ProEdgeTessellationGet()
- ProCurveTessellationGet()
- prodb_surface_tessellation()
- ProPartTessellate()
- ProPartTessellationFree()

When Pro/ENGINEER draws an edge, it represents it as a sequence of straight lines. If you create a small hole and zoom in on it, these tessellations are easily visible. The function ProEdgeTessellationGet() enables you to invoke the algorithm that generates this sequence of lines. The outputs from the function are as follows:

- 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

Function ProCurveTessellationGet() retrieves the curve tessellation for a datum curve. It returns the number of tessellation points and a list of them.

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. Note that this function tessellates a single surface only.

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 struct data types. Use function ProPartTessellationFree() to release this memory.
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 Faces, 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 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

For these two functions, you provide the memory for each of the outputs. If any of the outputs are not required, you can pass a NULL pointer instead, and the function will not calculate the corresponding values.

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. 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.

**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:

```c
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.

```c
/*================================================================*
FUNCTION: UserArcDiaDisp()
PURPOSE: Display the diameter of the selected arc.
/*================================================================*/

int 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 the user to select an arc.
    /*----------------------------------------------------------------*/
    ProStringToWstring (msgfile, "msg_ug7.txt");
    status = ProMessageDisplay (msgfile, "UG7 Select Arc to Evaluate:");
    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);

    status = ProGeomitemToEdge (&p_mdl_item, &edge);
    status = ProEdgeTypeGet (edge, &edge_type);
    if ((edge_type != PRO_ENT_ARC) && (edge_type != PRO_ENT_CIRCLE))
    {
        status = ProMessageDisplay (msgfile, "UG7 Invalid Arc Selection");
        return ((int)PRO_TK_GENERAL_ERROR);
    }
    status = ProGeomitemdataGet (&p_mdl_item, &geom_data);
    diameter = geom_data->data.p_curve_data->arc.radius*2.0;
    status = ProMessageDisplay (msgfile, "UG7 Arc Diameter = %.0f", &diameter);
    return ((int)status);
```
Geometry of Surfaces

Functions introduced:

- **ProSurfaceTypeGet()**
- **ProSurfaceDataGet()**

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 edge. Function **ProSurfaceDataGet()** returns the data structure associated with the specified edge.

Use function **ProSurfacedataMemoryFree()** to free the top-level memory associated with the edge 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
- 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).

**Example 3: Getting the Angle of a Conical Surface**

This example shows how to read the geometry equation of a surface.

```c
/*===============================================================*
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_uq7.txt");
    status = ProMessageDisplay (msgfile,
        "UG7 Select Cone to Evaluate:");
    if ((ProSelect ("surface", 1, 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.

- **ProQuiltdataGet()**
- **ProQuiltdataSurfArrayGet()**
- **ProQuiltdataSurfArraySet()**

  The function `ProQuiltdataGet()` retrieves information from the quilt data structure. The helper functions
  `ProQuiltdataSurfArrayGet()` and `ProQuiltdataSurfArraySet()` return or set an array of pointers to the quilt surfaces in the quilt data structures.

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()**
- **ProPoint3dOnsurfaceFind()**
- **ProPoint3dIntoleranceFind()**
- **ProSolidProjectPoint()**

The function **ProPointCoordGet()** provides the X, Y, and Z coordinates of the specified *ProPoint* object.

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()**.

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 *progtol_set_gtol()*.

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 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.

**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.

**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.

## Measurement

Functions introduced:

- `ProSurfaceAreaEval()`
- `ProContourAreaEval()`
- `ProSurfaceExtremesEval()`
- `ProSurfacePrincipalCrvtEval()`
- `ProEdgeLengthEval()`
- `ProCurveLengthEval()`
- `ProEdgeLengthT1T2Eval()`
- `ProCurveLengthT1T2Eval()`
- `ProEdgeParamByLengthEval()`
- `ProCurveParamByLengthEval()`
- `ProGeomitemDistanceEval()`
- `ProGeomitemAngleEval()`
- `ProSurfaceDiameterEval()`
- `ProGeomitemDiameterEval()`
- `ProContourBoundbox3dCompute()`
- `ProContourBoundbox2dCompute()`

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 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

Note that 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.

**ProContourBoundbox2d/3dCompute()** 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.

```c
/*================================================================*/
FUNCTION: user_surf_clear
PURPOSE:  Compute the clearance between two surfaces.
/*================================================================*/

int user_surf_clear (p_model, opt)
Prohandle   p_model;
int         opt;
{
    int         interf;
    double      distance, coord[2][3];
    Select3d   *sel;

    while (1)
    {
        promsg_print (MSGFIL, "USER Select a face");
        if (pro_select ("face", 2, &sel, 0, 0) <= 1)
            break;
        /*--------------------------*/
        Computes the clearance between the two selected surfaces.
        /*--------------------------*/
        err = pro_compute_clearance (sel, opt, &distance,
            &interf, coord);
```
promsg_print (MSGFIL, "USER Distance is: %0f", &distance);
if (interf)
    promsg_print (MSGFIL, "USER Interference exists");
}
return (1);

/*================================================================*\
FUNCTION: user_dist_manifolds
PURPOSE:  Compute the critical distance between two objects.
\*================================================================*/

user_dist_manifolds (p_model)
Prohandle p_model;
{
    int          status, old_color;
    Select3d    *sel;
    double       uvpar[2][2], xyz[2][3], distance;

    promsg_print (MSGFIL, "USER %0s", "Select two entities");
    if (pro_select ("point, edge, curve, surface", 2, &sel, 0, 0) < 2)
        return (0);

    /*-----------------------------------------------*/
    Compute the distance between the two manifolds.
    /*-----------------------------------------------*/
    status = pro_dist_manifolds (&sel[0], &sel[1], uvpar, xyz,
                   &distance);

    if (status)
        promsg_print (MSGFIL, "USER Distance is: %0f", &distance);
    else
        promsg_print (MSGFIL, "USER %0s", "No min/max distances found");

    return (1);
}

/*================================================================*\
FUNCTION: user_part_interference
PURPOSE:  Compute the interference between two parts.
\*================================================================*/

int user_part_interference (p_model)
Prohandle p_model;
{
    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];

promsg_print (MSGFIL, "USER %0s", "Select two parts");
if (pro_select ("part", 2, &sel, 0, 0) <= 1)
    return (0);

/*----------------------------------------------------------------*
  Compute the interference between the two parts.
  *----------------------------------------------------------------*/
err = pro_compute_interference (sel, 0, &intf_surfs, 
    &n_surfs);
if (err != PRODEV_NO_ERROR)
    return (-1);
if (n_surfs == 0)
{
    promsg_print (MSGFIL, "USER %0s", "No interference was 
detected.");
    return (0);
}

sprintf (str, "%d interference surfaces found. Display? [Y]:", 
    n_surfs);
promsg_print (MSGFIL,"USER %0s", str);
if (!promsg_getstring (w_str, 4))
{
    pro_wstr_to_str (a_str, w_str);
    if (a_str[0] != 'y' || a_str[0] != 'Y')
        return (0);
}
user_gen_filename (p_model, ".intf", fname);
if ((fp = fopen (fname, "w")) == NULL)
    return (-1);
fprintf (fp, "Interference in ");
user_dump_mdl (fp, p_model);
fprintf (fp, "\n\n");

/*----------------------------------------------------------------*
  Compute the interference volume.
  *----------------------------------------------------------------*/
err = pro_compute_volume (intf_surfs, n_surfs, &volume);
user_dump_interferences (fp, sel, volume);

/*----------------------------------------------------------------*
  Highlight the interference regions.
  *----------------------------------------------------------------*/
err = pro_display_interf_volume (intf_surfs, n_surfs, 
    EDGE_HIGHLITE_COLOR);
sprintf (str, "Total Interference Volume = %lf", volume);
promsg_print (MSGFIL, "USER %0s", str);

/*----------------------------------------------------------------*
  Release the memory for the interference volume.
  *----------------------------------------------------------------*/
err = pro_interference_volume_release (intf_surfs, n_surfs);
fclose (fp);
pro_show_file (pro_str_to_wstr (w_fname, fname));
return (0);
}

/*================================================================*
FUNCTION: user_global_interference
PURPOSE:  Compute the global interference within an assembly.
/*================================================================*/

int user_global_interference (p_model)
Prohandle p_model;
{
    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];

    /*----------------------------------------------------------------*
    Compute the interference within the assembly.
    /*----------------------------------------------------------------*/
    err = pro_compute_global_interference (p_model, 0, &n_intf_parts,
                                   &intf_parts, &intf_part_surfs, &n_part_surfs);
    if (err != PRODEV_NO_ERROR)
        return (-1);

    if (n_intf_parts == 0)
        promsg_print (MSGFIL, "USER %0s", "No interference was
detected.");

    sprintf (str, "%d interfering part pairs found. Display
        interference pair volumes? [Y]:", n_intf_parts);
    promsg_print (MSGFIL, "USER %0s", str);

    if (!promsg_getstring (w_str, 4))
    {
        pro_wstr_to_str (a_str, w_str);
        if (a_str[0] != 'y' || a_str[0] != 'Y')
            return (0);
    }
    user_gen_filename (p_model, ".intf", fname);
    if ((fp = fopen (fname, "w")) == NULL)
        return (-1);
    fprintf (fp, "Interference in ");
    user_dump_mdl (fp, p_model);
    fprintf (fp, "\n\n");
for (i = 0; i < n_intf_parts; i++)
{
    /*----------------------------------------------------------------*
     * Compute the interference volumes.
     *----------------------------------------------------------------*/
    err = pro_compute_volume (intf_part_surfs[i],
                             n_part_surfs[i], &volume);
    user_dump_interferences (fp,
                             (Select3d *)&(intf_parts[i].part1_sel), volume);
    /*----------------------------------------------------------------*
     * Highlight the interference regions.
     *----------------------------------------------------------------*/
    err = pro_display_interf_volume (intf_part_surfs[i],
                                     n_part_surfs[i], EDGE_HIGHLITE_COLOR);
    sprintf (str, "Interference pair %d of %d: Volume = %0.2lf:
              Continue? [y]:", i+1, n_intf_parts, volume);
    promsg_print (MSGFIL, "USER %0s", str);
    if (!promsg_getstring (w_str, 4))
    {
        pro_wstr_to_str (a_str, w_str);
        if (a_str[0] != 'y' || a_str[0] != 'Y')
        {
            fclose (fp);
            err = pro_interference_volume_release(
                   intf_part_surfs[i], n_part_surfs[i]);
            return (0);
        }
    }
    /*----------------------------------------------------------------*
     * Release the memory for the interference volume.
     *----------------------------------------------------------------*/
    err = pro_interference_volume_release (intf_part_surfs[i],
                                           n_part_surfs[i]);
    fclose (fp);
    pro_show_file (pro_str_to_wstr (w_fname, fname));
    return (0);
}
fprintf (fp, "Interfering Parts:\n");
user_dump_mdl (fp, intf[0].part_ptr);
for (m = 0; m < intf[0].memb_num; m++)
{
    fprintf (fp, "\t\tmemb_id_tab[%d],\n",
               intf[0].memb_id_tab[m]);
    if (m < intf[0].memb_num-1)
        fprintf (fp, ",\n");
}
user_dump_mdl (fp, intf[1].part_ptr);
for (m = 0; m < intf[1].memb_num; m++)
{
    fprintf (fp, "\t\tmemb_id_tab[%d],\n",
               intf[1].memb_id_tab[m]);
    if (m < intf[1].memb_num-1)
        fprintf (fp, ",\n");
}
fprintf (fp, "Interfering Volume: %0.2lf\n\n", volume);
return(0);
}

user_dump_mdl (fp, p_model)

FILE          *fp;
Prohandle      p_model;
{
    Pro_object_info      info;
    char                 name[80], type[10];
    if (p_model == NULL)
        return (0);
    prodb_get_object_info (p_model, &info);
    fprintf (fp, "%.8s: %s, \n", pro_wstr_to_str (name, info.name),
             pro_wstr_to_str (type, info.type));
Sheet Metal

Geometry Analysis

Functions introduced:

- ProSmtPartThicknessGet()
- ProSmtedgeContourGet()
- ProSmtOppsurfGet()
- ProSmtOppedgeGet()
- ProSmtBendsrfParentGet()
- ProSmtBendsrfChildGet()
- ProSmtBendedgeChildGet()
- ProSmtBendedgeParentGet()
- ProSmtSurfaceTypeGet()
- ProSmtMdlIsFlatStateInstance()
- ProfaminstancelsFlatState()

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.

The function **ProSmtPartThicknessGet()** returns the dimension that defines the thickness of the specified sheet metal component. If you specify a non sheet metal part, this function returns PRO_TK_BAD_CONTEXT.

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. If the edge is not in the part geometry, the function returns `PRO_TK_BAD_CONTEXT`. If the edge is on a green or white part surface but does not belong to an outside or inside contour, the function returns `PRO_TK_BAD_INPUTS`. This function requires information about sheet metal surfaces and surfaces types. Obtain the surface type by passing the handle of the surface containing the edge as an argument to `ProSmtSurfaceTypeGet()`.

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. Function-specific definitions follow.

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. Function-specific definitions follow.

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. Function-specific definitions follow.
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. Function-specific definitions follow.

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 **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 5: Sheet Metal Geometry Analysis**

```c
#include "ProToolkit.h"
#include "ProMdl.h"
#include "ProSolid.h"
#include "ProSurface.h"
#include "ProSheetmetal.h"
#include "ProMode.h"

FLICTOOLKIT.HD
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()", "ProTestShtmtlSurfAct()",
                       status, status != PRO_TK_NO_ERROR );
     for (i = 0; i < surfaces_num; i++)
     {
         status = ProTestShtmtlSurfAct (surfaces[i],
                                         PRO_TK_NO_ERROR, (ProAppData)&app_data);
     }
     status = ProArrayFree ((ProArray*)&surfaces);
     TEST_CALL_REPORT( "ProArrayFree()", "ProTestShtmtlSurfAct()",
                       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)
{
  ProTestSmt1Surf *app_data = (ProTestSmt1Surf *) tmp_app_data;
  ProSmtSurfType smt_surf_type;
  ProMdl *model;
  int surf_id;
  ProError status;
  FILE *fp;

  fp     = app_data->fp;
  model  = app_data->model;

  /*-----------------------------------------*\n  \* Get the surface id. \*-----------------------------------------*/
  status = ProSurfaceIdGet(surface, &surf_id);
  TEST_CALL_REPORT("ProSurfaceIdGet()", "ProTestShtmtlSurfAct()",
                   status, status != PRO_TK_NO_ERROR);

  /*-----------------------------------------*\n  \* Get the sheetmetal surface type. \*-----------------------------------------*/
  status = ProSmtSurfaceTypeGet ((ProPart)(*model), surface,
                                &smt_surf_type);
  TEST_CALL_REPORT("ProSmtSurfaceTypeGet()", "ProTestShtmtlSurfAct()",
                   status, status != PRO_TK_NO_ERROR);

  /*-----------------------------------------*\n  \* 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 :
      break;
  }  

Geometry 9 - 39
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

Functions introduced:

- `ProSmtPartBendtableApply()`
- `ProSmtPartBendtableRemove()`
- `ProSmtFeatureBendtableApply()`
- `ProSmtFeatureBendtableRemove()`
- `ProSmtFeatureDevldimsGet()`
- `ProSmtDevldimIsDriven()`
- `ProSmtDevldimDrivenSet()`

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 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.

Sheet metal dimension functions find or set whether or not developed length dimensions are driven.

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.

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.
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 `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.

Function `ProSmtDevldimIsDriven()` specifies whether a developed length dimension is driven or not. Use the function `ProSmtDevldimDrivenSet()` to set a developed length dimension to driven.
This chapter describes the various coordinate systems used by Pro/ENGINEER and Pro/TOOLKIT, and how to transform coordinates from one to another.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coordinate Systems</td>
<td>10 - 2</td>
</tr>
<tr>
<td>Coordinate System Transformations</td>
<td>10 - 4</td>
</tr>
</tbody>
</table>
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:

```c
typedef double ProPoint3d[3]
```

Vectors are distinguished for clarity by a different, though equivalent, declaration:

```c
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:

```c
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 10-1: Transformation Matrix**

<table>
<thead>
<tr>
<th>rotation (3 x 3)</th>
<th>projection (3 x 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>translation (1 x 3)</td>
<td>scaling/zoom (1 x 1)</td>
</tr>
</tbody>
</table>

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

Function 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**

```c
void UserSolidToScreen(
    ProSolid   solid,
    ProPoint3d solid_point,
    ProPoint3d screen_point)
{
    ProMatrix  view_matrix;
    ProError   status;

    status = ProViewMatrixGet (solid, NULL, view_matrix);
    status = ProPntTrfEval (solid_point, view_matrix, screen_point);
}
```

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.

```c
void UserScreenToDwg (ProDrawing drawing, ProPoint3d scn_point, ProPoint3d drw_point)
{
    ProName w_size;
    ProMatrix matrix;

    ProDrawingSheetTrfGet (drawing, 1, w_size, matrix);

    ProPntTrfEval (scn_point, matrix, drw_point);
}
```
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 2: 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, &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;
    
    ProAsmcomppathTrfGet (&comp_path, PRO_B_TRUE, matrix);
    
    status = ProPntTrfEval(p3d, matrix, t_point);
}
else
{
    ProSelectionModelitemGet (p_sel[0], &item);
    top_model = item.owner;
    memcpy (t_point, p3d, sizeof (ProPoint3d));
}

ProViewMatrixGet (top_model, NULL, matrix);
ProPntTrfEval (t_point, matrix, scrpnt);

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 - scrnt[j]*scale;
}

zoom_matrix [3][3] = 1.0;

ProMdlWindowGet (top_model, &window);
status = ProWindowPanZoomMatrixSet (window, zoom_matrix);

/*---------------------------------------------------------------------*
Repaint the window
\*---------------------------------------------------------------------*/
ProWindowRepaint (window);

return (status);
}
This chapter describes how to use dimensions with Pro/TOOLKIT.

**Note:** Actions that are specific to drawing mode, such as the ability to create dimensions and conversion to ordinate dimensions are described in the chapter on 'Drawings'.

This chapter is divided into the following parts:

**Topic**

The ProDimension Object 11 - 2  
Reading Dimensions 11 - 3  
Modifying Dimensions 11 - 4  
ISO/DIN Tolerance Table Use 11 - 5  
Dimension Text 11 - 6  
Displaying Dimensions 11 - 6  
Dimension Display Modes 11 - 7  
Designating Dimensions and Symbols 11 - 7  
Dimension Tolerances 11 - 8  
Relations 11 - 8
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 fields 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()).

Note: In Release 2000i2 and later, pattern parameters are treated as ordinary dimensions and stored in Pro/ENGINEER as dimensions. Use regular Dimension functions to access them. This means individual pattern parameters in Pro/ENGINEER Release 2000i2 and later have different ids than in Pro/ENGINEER releases earlier than Release 2000i2.
Reading Dimensions

Functions introduced:

- ProSolidDimensionVisit()
- ProDrawingDimensionVisit()
- ProDimensionSymbolGet()
- ProDimensionValueGet()
- ProDimensionTypeGet()
- ProDimensionToleranceGet()
- ProDimensionIsFractional()
- ProDimensionDecimalsGet()
- ProDimensionDenominatorGet()
- ProDimensionIsReldriven()
- ProDimensionIsRegenednegative()
- ProDimensionNomvalueGet()
- ProDimensionBoundSet()
- ProDimensionBoundGet()

The two visit functions ProSolidDimensionVisit() and ProDrawingDimensionVisit() conform to the usual style of visit functions. (Refer to section Visit Functions in the ‘Fundamentals’ chapter.) 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() outputs the symbol (the name) of the specified dimension.

The function ProDimensionValueGet() outputs the value of the dimension.

The function ProDimensionTypeGet() outputs the type of the dimension in terms of the following values:

- LINEAR
- RADIUS
- DIAMETER
- ANGLE

The function ProDimensionToleranceGet() reports the deviation from the nominal value of the upper and lower tolerances.
The function `ProDimensionIsFractional()` outputs 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()` outputs the value of the denominator used to define the fractional value.

The function `ProDimensionIsRelDriven()` outputs whether the dimension is driven by a relation.

The function `ProDimensionIsRegenednegative()` outputs 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 output 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.)

Use the function `ProDimensionNomvalueGet()` to return 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.

The functions `ProDimensionBoundSet()` and `ProDimensionBoundGet()` set and return the dimension bound status, respectively.

### Modifying Dimensions

Functions introduced:

- `ProDimensionSymbolSet()`
- `ProDimensionValueSet()`
- `ProDimensionToleranceSet()`
- `ProDimensionDecimalsSet()`
- `ProDimensionDenominatorSet()`
- `ProDimensionDimensionReset()`

The function `ProDimensionSymbolSet()` allows you to change the symbol (the name) of a dimension. This function can be used 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 reference dimension or a dimension which is driven by a relation.

The function **ProDimensionToleranceGet()** sets the tolerance upper and lower limits. It is not applicable to reference dimensions.

The functions **ProDimensionDecimalsSet()** and **ProDimensionDenominatorSet()** set the displayed resolution of the dimension, depending upon whether the display is decimal or fractional. These functions can be used only with solid dimensions.

The function **ProDimensionDimensionReset()** sets the dimension to the value it had at the end of the last successful regeneration. It can be used to recover from a failed regeneration.

**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.

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

Function **ProSolidTolclassLoad** loads a hole or shaft ISO/DIN tolerance table into the current session memory.
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.

**Displaying Dimensions**

Functions introduced:

- `ProDimensionDisplayUpdate()`
- `ProDimensionShow()`
- `ProDimensionErase()`

The function `ProDimensionDisplayUpdate()` redispalyes a specified dimension, if it is currently displayed. If it is not currently displayed, there is no effect. You should always call this function after calling one of the `ProDimension*Set()` functions, unless you are sure that the dimension is not currently displayed.

To force the display of a single dimension, either standard or reference, use `ProDimensionShow()`. 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()`.
Dimension Display Modes

Functions introduced:

- **ProDisplaymodeGet()**
- **ProDisplaymodeSet()**

These functions tell you 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 `Relations, Switch Dim`.

Designating Dimensions and Symbols

Function described:

- **prodb_designate_param()**
- **ProSymbolDesignate()**
- **ProSymbolUndesignate()**
- **ProSymbolDesignationVerify()**

The function **prodb_designate_param()** designates model parameters and dimensions for Pro/PDM and Pro/INTRALINK. In the function call, the argument `param_name` specifies either the name of the parameter or the dimension symbol normally used in relations.

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.
Dimension Tolerances

Functions introduced:

- `ProToleranceDefaultGet()`
- `ProSolidToleranceGet()`
- `ProSolidToleranceSet()`
- `ProDimensionIsToleranceDisplayed()`

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).

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`.

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 access relations on all the types of model and model item that can have relations in Pro/ENGINEER. These are listed in the following table.

**Note:** Access to Sketcher relations is not supported in the current release.

<table>
<thead>
<tr>
<th>Model Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_PART</td>
<td>Part</td>
</tr>
<tr>
<td>PRO_ASSEMBLY</td>
<td>Assembly</td>
</tr>
<tr>
<td>PRO_DRAWING</td>
<td>Drawing</td>
</tr>
<tr>
<td>PRO_REPORT</td>
<td>Report</td>
</tr>
<tr>
<td>PRO_DIAGRAM</td>
<td>Diagram</td>
</tr>
<tr>
<td>PRO_DWGFORM</td>
<td>Format</td>
</tr>
<tr>
<td>PRO_UDF</td>
<td>User-defined feature</td>
</tr>
<tr>
<td>PRO_FEATURE</td>
<td>Feature</td>
</tr>
<tr>
<td>PRO_SURFACE</td>
<td>Surface</td>
</tr>
<tr>
<td>PRO_EDGE</td>
<td>Edge</td>
</tr>
<tr>
<td>PRO_WELD_PARAMS</td>
<td>Weld parameters</td>
</tr>
<tr>
<td>PRO_BND_TABLE</td>
<td>Bend table</td>
</tr>
<tr>
<td>PRO_EXTOBJ</td>
<td>External objects</td>
</tr>
<tr>
<td>PRO_PATREL_FIRST_DIR</td>
<td>Pattern direction 1</td>
</tr>
<tr>
<td>PRO_PATREL_SECOND_DIR</td>
<td>Pattern direction 2</td>
</tr>
<tr>
<td>PRO_RELOBJ_QUILT</td>
<td>Quilt</td>
</tr>
<tr>
<td>PRO_RELOBJ_CRV</td>
<td>Curve</td>
</tr>
<tr>
<td>PRO_RELOBJ_COMP_CRV</td>
<td>Compound curve</td>
</tr>
</tbody>
</table>

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 a 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 '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.
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 ‘Geometry’ chapter.

Access to the internal structure of a feature is described in the ‘Principles of Feature Creation’ chapter.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature Objects</td>
<td>12 - 2</td>
</tr>
<tr>
<td>Visiting Features</td>
<td>12 - 2</td>
</tr>
<tr>
<td>Feature Inquiry</td>
<td>12 - 3</td>
</tr>
<tr>
<td>Feature Geometry</td>
<td>12 - 6</td>
</tr>
<tr>
<td>Manipulating Features</td>
<td>12 - 6</td>
</tr>
<tr>
<td>Feature Dimensions</td>
<td>12 - 10</td>
</tr>
<tr>
<td>Manipulating Patterns</td>
<td>12 - 10</td>
</tr>
<tr>
<td>Creating Local Groups</td>
<td>12 - 17</td>
</tr>
<tr>
<td>Read Access to Groups</td>
<td>12 - 18</td>
</tr>
</tbody>
</table>
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 the next section) 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()`
- `ProSolidFeatstatusGet()`
- `ProFeatureIsNcseq()`
- `ProFeatureSolidGet()`
- `ProFeatureIsIncomplete()`
- `ProFeatureChildrenGet()`
- `ProFeatureParentsGet()`
- `ProFeatureSelectionGet()`
- `ProFeatureHasGeomchks()`
- `ProFeaturesIsReadonly()`
- `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 `ProFeatureTypenameGet()` 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 suppresses 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 **ProSolidFeatstatusGet()** provides a list of the integer identifiers and statuses of all features in the 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 **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`.

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 `prodb_feature_info()` structure returned by the Pro/Develop function `prodb_feature_info()`.
Feature Geometry

Functions introduced:

- `ProFeatureGeomitemVisit()`
- `ProGeomitemFeatureGet()`

For information about feature geometry, see the ‘Geometry’ chapter.

Manipulating Features

Functions introduced:

- `ProFeatureDelete()`
- `ProFeatureSuppress()`
- `ProFeatureRedefine()`
- `ProFeatureInsertModeActivate()`
- `ProInsertModeCancel()`
- `ProFeatureReadonlySet()`
- `ProFeatureReadonlyUnset()`
- `ProFeatureReorder()`
- `ProFeatureNumberGet()`
- `ProSolidFeatstatusGet()`

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 a Boolean input that indicates whether to also delete or suppress features dependent on those being acted on directly, and another Boolean that indicates whether the function interacts with Pro/ENGINEER to get a fix if the subsequent regeneration fails.

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.
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.

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.

```c
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, &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.

```c
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

```c
status = UserAxisRefs();
```

Deactivate insert mode.

```c
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 `ProFeatureReorder()` function enables you to change the position of one or more features in the feature regeneration sequence. Its 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.
- `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()`.
Example 2: Changing the Regeneration Sequence of a Solid

This example shows how to reorder features.

```c
int FeatureReorderTest()
{
    static wchar_t    msgfil[80];
    ProMd1            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,
                      &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,
                     &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 = ProMd1CurrentGet (&model)!= PRO_TK_NO_ERROR;
status = ProFeatureReorder ((ProSolid)model, sel_feat_ids, n_feats_sel, feat_num);
return (0);

Feature Dimensions

Function introduced:

• ProFeatureDimensionVisit()

The function ProFeatureDimensionVisit() visits all the dimensions which belong to the feature.

For more information about dimensions, see the chapter ‘Dimensions and Relations’.

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()
• proptn_get_leader()

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 ‘Creating 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()**.

Alternatively, the Pro/DEVELOP-style function **proptn_get_leader()** gives the identifier of the feature that leads the pattern containing the specified feature. The function can be used for both ordinary and group patterns. Note that a given feature can be a member of both types of pattern.

There are two ways to access pattern information:

• Use the pattern element tree described in the chapter ‘Creating Patterns’. You can access element tree information using the functions **ProElement*()**, described in the chapter Principles of Feature Creation.

• Use the Pro/DEVELOP-style function **proptn_get_leader()** to fill a data structure that describes the pattern led by the specified feature. This method is valid for reference and table-driven patterns, as well as for dimension-driven patterns. The corresponding data structure is called **Pro_Pattern**, which is declared in the file *prodevdim.h*.

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:

```c
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;
```

```c
typedef enum pro_pattern_type
{
  PRO_PTYPE_DIMENSION,
  PRO_PTYPE_REFERENCE,
  PRO_PTYPE_TABLE,
  PRO_PTYPE_FILL
} Pro_pattern_type;
```

```c
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:

```c
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:

```c
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 proptn_get_leader().

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 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 a ProFeature data handle:

```c
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()`
- `ProGroupIsTabledriven()`
- `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()`
- `ProUdfCreate()`

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).
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 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 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).

• 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.

• 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()
- 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.

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 Dimensions and Pattern Parameters

The data structure for variable dimensions and pattern parameters is **ProUdfvardim**.

Functions introduced:

- **ProUdfvardimAlloc()**
- **ProUdfdataUdfvardimAdd()**

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.

### UDF References

Functions introduced:

- `ProUdfreferenceAlloc()`
- `ProUdfdataReferenceAdd()`

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. 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()`.
- **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.
Assembly Intersections

The data structure used for assembly intersections is ProUdfIntersection.

Functions introduced:

- ProUdfIntersectionAlloc()
- ProUdfdataIntersectionAdd()

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 == 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.
Component 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.
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:

- The Selection Object
- Coordinate Systems and Transformations
- The Geometry section

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structure of Assemblies and Assembly Objects</td>
<td>13 - 2</td>
</tr>
<tr>
<td>Visiting Assembly Components</td>
<td>13 - 5</td>
</tr>
<tr>
<td>Regenerating an Assembly Component</td>
<td>13 - 11</td>
</tr>
<tr>
<td>Locations of Assembly Components</td>
<td>13 - 11</td>
</tr>
<tr>
<td>Assembling Components</td>
<td>13 - 13</td>
</tr>
<tr>
<td>Redefining and Rerouting Components</td>
<td>13 - 13</td>
</tr>
<tr>
<td>Deleting Components</td>
<td>13 - 14</td>
</tr>
<tr>
<td>Merge and Cutout</td>
<td>13 - 15</td>
</tr>
<tr>
<td>Automatic Interchange</td>
<td>13 - 16</td>
</tr>
</tbody>
</table>
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 'Geometry').

However, assemblies do not contain active geometry items other than those in datums—that is, no “solid” geometry as described in the Geometry chapter and in Pro/ENGINEER Feature Creation. 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 \textit{ProAsmcomppath} is as follows:

\begin{verbatim}
typedef struct pro_comp_path
{
    ProSolid    owner;
    ProIdTable  comp_id_table;
    int         table_num;
} ProAsmcomppath;
\end{verbatim}

The data structure fields are as follows:

- \textit{owner}—Identifies the root assembly
- \textit{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
- \textit{table_num}—Specifies the number of component identifiers in the \textit{comp_id_table} array

The following figure ‘Sample Assembly Hierarchy’ shows an assembly hierarchy with two examples of the contents of a \textit{ProAsmcomppath} object.
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        table_num = 4
comp_id_tab[0] = 2   comp_id_tab[0] = 11
comp_id_tab[4] = 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

- ProAsmcompIsBulkitem()
- ProAsmcompIsPackaged()

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

- ProAsmcompIsUnderconstrained()
- ProAsmcompIsFrozen()
- ProAsmcompIsUnplaced()
- ProAsmcompIsPlaced()
- ProAsmcompIsSubstitute()
- 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.

```c
/*-----------------------------*/
Pro/TOOLKIT includes
*/
#include <ProToolkit.h>
#include <ProMdl.h>
#include <ProFeature.h>
#include <ProFeatType.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);

Function to write out the members of the current assembly, and display the result in an information window.

ProMdl       asm;
char         name[PRO_NAME_SIZE];
char         type[PRO_TYPE_SIZE];
wchar_t      wname[PRO_NAME_SIZE];
ProMdlData   mdldata;
ProError     err;
UserAppdata  appdata;
FILE        *fp;

ProMdlCurrentGet (&asm);

Open the text file.

strcpy (name,FILENAME);
fp = fopen (name,"w");
ProMdlDataGet (asm, &mdldata);
ProWstringToString (name, mdldata.name);
ProWstringToString (type, mdldata.type);
fprintf (fp, "%s %s\n",name,type);
appdata.fp = fp;
appdata.level = 1;

List the assembly members.

ProSolidFeatVisit (asm, user_action, UserAsmCompFilter, &appdata);

Close the file and display it.

fclose (fp);
ProStringToWstring (wname, FILENAME);
ProInfoWindowDisplay (wname, NULL, NULL);
return (PRO_TK_NO_ERROR);
FUNCTION: UserAsmCompFilter()
PURPOSE: A filter used by ProSolidFeatVisit() to visit
features that are assembly components

ProError UserAsmCompFilter (const 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);
}

Write the information to a file.

ProError user_action (const 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;
    ProAsmcompMdlGet (feature, &mdl);
    ProMdldataGet (mdl, &mdldata);
    ProWstringToString (name, mdldata.name);
    ProWstringToString (type, mdldata.type);
    for (l = 0; l < level; l++)
fprintf(fp," ");
fprintf(fp,"%s  %s\n",name,type);
if (strncmp(type,"ASM",3) == 0)
{
    appd1.fp = appd->fp;
    appd1.level = appd->level+1;
    ProSolidFeatVisit(mdl, user_action, UserAsmCompFilter, &appd1);
}
if (feature != NULL)
    return(PRO_TK_NO_ERROR);
return (PRO_TK_CONTINUE);

Regenerating an Assembly Component

Function introduced:
• ProAsmcompRegenerate()

The function ProAsmcompRegenerate() regenerates an assembly component, given the component handle. The function regenerates the assembly component 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.

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.

```c
/*====================================================================*
 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;
   ProFeattypetype;
   FILE *fp_out=NULL;
   ProName w_name, wfile_name;
   ProCharNamename;

   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);
     }
   }
```
Assembling Components

To assemble components into an assembly, use the methods of feature creation. These methods are described in detail in the chapter ‘Assembling Components’.

Redefining and Rerouting Components

The functions used to redefine and reroute components are described in the chapter ‘Assembling Components’.
Deleting Components

Function introduced:

- **ProFeatureDelete()**

  The function **ProFeatureDelete()** deletes components. It has the same options as described in the chapter 'Basic Access to 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:

```c
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 ‘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()`.
This chapter describes the Pro/TOOLKIT functions that give access to parameters and geometric tolerances. Note that the functions in this chapter can also be used to add parameters to external objects (see the 'External Objects' chapter).

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter Objects</td>
<td>14 - 2</td>
</tr>
<tr>
<td>Parameter Values</td>
<td>14 - 3</td>
</tr>
<tr>
<td>Accessing Parameters</td>
<td>14 - 4</td>
</tr>
<tr>
<td>Designating Parameters for Pro/INTRALINK</td>
<td>14 - 7</td>
</tr>
</tbody>
</table>
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;
}
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.

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()`
- `ProParameterVisit()`

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` 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 outputs a `ProBoolean` 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, given the **ProModelitem** for the owner, the name, and a **ProParamvalue** structure for the value. The function outputs a valid **ProParameter** object for the new parameter.

The function **ProParameterDelete()** deletes a parameter, specified by a **ProParameter** object, from the Pro/ENGINEER database.

The function **ProParameterSelect()** asks the user to select a parameter from those belonging to a specified database item by displaying a menu of their names. The function outputs a **ProParameter** structure for the selected parameter.

The function **ProParameterVisit()** visits all the parameters on a specified database item.

**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()**.

```c
/*================================================================*
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_u12.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, &nsel);

if (nsel <= 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

Functions introduced:

- **ProParameterDesignationAdd()**
- **ProParameterDesignationVerify()**
- **ProParameterDesignationRemove()**

These functions control the designation of parameters for Pro/INTRALINK. A designated parameter will become visible within Pro/INTRALINK 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.

The function **ProParameterDesignationVerify()** tells you whether a parameter is currently designated for Pro/INTRALINK.
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 the reader 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.

This chapter describes the following:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geometric Tolerance Objects</td>
<td>15 - 2</td>
</tr>
<tr>
<td>Geometric Tolerance Datum References</td>
<td>15 - 2</td>
</tr>
<tr>
<td>Visiting Geometric Tolerances</td>
<td>15 - 3</td>
</tr>
<tr>
<td>Reading Geometric Tolerances</td>
<td>15 - 4</td>
</tr>
<tr>
<td>Creating a Geometric Tolerance</td>
<td>15 - 6</td>
</tr>
<tr>
<td>Editing a Geometric Tolerance</td>
<td>15 - 13</td>
</tr>
<tr>
<td>Deleting a Geometric Tolerance</td>
<td>15 - 14</td>
</tr>
</tbody>
</table>
Geometric Tolerance Objects

ProGtol

Geometric tolerances in a Pro/ENGINEER model are referenced by the D handle ProGtol. This is declared as a typedef to ProModelitem, where the type field is set to PRO_GTOL. ProGtol inherits from ProModelitem, so if you use ProSelect() with the option “gtol” to ask the user to select a gtol, you can then use ProSelectionModelitemGet() to get the ProGtol handle.

ProGtoldata

This is an O handle object that references an internal data structure containing a complete description of a gtol. During creation of a gtol, the description is in workspace; while editing a gtol, the object refers directly to the gtol description in the Pro/ENGINEER model. Creating and editing gtols are discussed in more detail below.

Geometric Tolerance Datum References

Functions introduced:

- ProGeomitemGtolrefSet()
- ProGeomitemGtolrefClear()
- ProGeomitemIsGtolref()

The function ProGeomitemGtolrefSet() turns a specified datum (either datum plane or axis) into a gtol reference. It is equivalent to the Pro/ENGINEER command Set Datum. The datum is identified by a ProGeomitem handle which must reference the geometry item itself, not the owning feature.

For a datum plane, the type field in the ProGeomitem must be PRO_DATUM_PLANE. If you find the datum by visiting, for example using ProFeatureGeomitemVisit(), or by selecting using ProSelect(), the resulting ProGeomitem will have its type field set to PRO_SURFACE. So before giving the ProGeomitem as an input to ProGeomitemGtolrefSet() you should reset the type field to PRO_DATUM_PLANE, preferably by calling ProModelitemInit(). See Example 1: Attaching a Position GTOL to Solid Planes Parallel to a Datum Plane.
ProGeomitemGtolrefClear() clears the gtol reference status of
the specified datum, and ProGeomitemIsGtolref() returns data
on whether a datum is currently a gtol reference.

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()
- ProGtolleaderGet()
- ProGtolleadersFree()
- ProGtoldataGtoldatumrefGet()
- ProGtoldatumrefGet()
- ProGtoldataCompositeGet()
- ProGtoldataValueGet()
- ProGtoldataPerUnitGet()
- ProGtoldataMatCondGet()
- ProGtoldataStatTolGet()
- ProGtoldataDiameterGet()
- ProGtoldataFreeStateGet()
- ProGtoldataAllAroundGet()
- ProGtoldataTangentPlaneGet()
- ProGtoldataProjZoneGet()
- ProGtoldataProfBoundaryGet()

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 to the contents of the Geometric Tolerance dialog in Pro/ENGINEER.

**ProGtoldataTypeGet()** provides the type of the gtol (straightness, flatness and so on). These values correspond to the 14 buttons at the left of the Pro/ENGINEER Geometric Tolerance dialog.
The functions **ProGtoldataModelGet()**, **ProGtoldataReferenceGet()**, **ProGtoldataPlacementGet()**, and **ProGtolleaderGet()** correspond to the information shown in the Model Refs tab of the Pro/ENGINEER Geometric Tolerance dialog.

**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()** provides 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. **ProSelection** structures provided by other **ProGtol*Get()** functions (such as **ProGtoldataReferenceGet()**) have the Owner as their root.

**ProGtoldataReferenceGet()** provides the type of entity referenced by the gtol and a **ProSelection** object which identifies the entity itself. **ProGtoldataPlacementGet()** provides the placement type and the items referenced by the placement. For example, if the placement type is “PROGTOLPTYPE_DIMENSION”, the function outputs a **ProDimension** object to show which dimension is used to define the gtol placement. If the placement type involves a leader, the function outputs an array of objects of the type **ProGtolleader**. To unpack the information in this opaque pointer, use function **ProGtolleaderGet()**. After reading the leaders, free the leader array (which contains copied objects) by calling **ProGtolleadersFree()**.

The functions **ProGtoldataGtoldatumrefGet()**, **ProGtoldatumrefGet()**, and **ProGtoldataCompositeGet()** correspond to the information on the Datum Refs tab of the Geometric Tolerance dialog. **ProGtoldataGtoldatumrefGet()** provides the primary, secondary, and tertiary datum references in terms of the opaque object **ProGtoldatumref**. 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()**, and **ProGtoldataMatCondGet()** correspond to the information on the Tol Value tab of the Pro/ENGINEER Geometric Tolerance dialog.

The functions **ProGtoldataStatTolGet()**, **ProGtoldataDiameterGet()**, **ProGtoldataFreeStateGet()**, **ProGtoldataAllAroundGet()**, **ProGtoldataTangentPlaneGet()**, **ProGtoldataProjZoneGet()**, and **ProGtoldataProfBoundaryGet()** correspond to the information on the Symbols tab of the Pro/ENGINEER Geometric Tolerance dialog.

---

**Creating a Geometric Tolerance**

Functions introduced:

- **ProGtoldataAlloc()**
- **ProGtoldataTypeSet()**
- **ProGtoldataModelSet()**
- **ProGtoldataReferenceSet()**
- **ProGtolleaderAlloc()**
- **ProGtolleaderPlacementSet()**
- **ProGtolleaderFree()**
- **ProGtoldatumrefAlloc()**
- **ProGtoldatumrefSet()**
- **ProGtoldatumrefFree()**
- **ProGtoldataCompositeSet()**
- **ProGtoldataValueSet()**
- **ProGtoldataPerUnitSet()**
- **ProGtoldataMatCondSet()**
- **ProGtoldataStatTolSet()**
- **ProGtoldataDiameterSet()**
- **ProGtoldataFreeStateSet()**
- **ProGtoldataAllAroundSet()**
The basic steps in creating a gtol are:

1. Allocate a ProGtolData structure using ProGtolDataAlloc().
2. Set tolerance properties using the ProGtolSet() functions.
3. Create the tolerance using ProGtolCreate().
4. Free the ProGtolData using ProGtolDataFree().

Be careful about the sequence in which applications call the ProGtolSet() 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 unset) 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 will become unset because it depends upon the reference.

The ProGtolSet() functions perform exactly the same operations that result from selecting the corresponding buttons in Pro/ENGINEER's Geometric Tolerance dialog. To ensure that you call the ProGtolSet() functions in the correct sequence, first experiment with creating the desired gtol interactively in Pro/ENGINEER, and then use a sequence of ProGtolSet() 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.
A useful feature of the ProGtol*Set functions is that each has an output argument of type ProGtoldataStatus, an enum 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:

<table>
<thead>
<tr>
<th>Function</th>
<th>GtoldataStatus</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProGtoldataReferenceSet()</td>
<td>PROGTOLSTATUS_PLACEMENT</td>
</tr>
<tr>
<td>ProGtoldataPlacementSet()</td>
<td>PROGTOLSTATUS_REF_DTM_NEEDED</td>
</tr>
<tr>
<td>ProGtoldataGtoldatumrefSet()</td>
<td>PROGTOLSTATUS_COMPLETE</td>
</tr>
</tbody>
</table>

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().

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, the user can choose to make it a reference.

```c
/*---------------------------------------------*/
Data structure for finding parallel solid plane surfaces

typedef struct planes_data
{
    ProGeomitem reference;
    ProVector normal;
    double tolerance;
} Planesdata_t;

/*---------------------------------------------*/
FUNCTION: UsrPlanePositiontolSet()
PURPOSE: To add a position gtol to the specified surface

int UsrPlanePositiontolSet(
```
Geometric Tolerancing

Geometric Tolerancing

ProSelection surface, /* The surface */
ProVector pos, /* The position of the gtol */
ProGeomItem *reference, /* The datum reference */
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);

/*--------------------------------------------------------------------*\
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);
sprintf(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->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;

/*-----------------------------------------------*/
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);
      if(UsrYesnoGet(msgfil,"YES", &ref_datum)==PRO_TK_MSG_USER_QUIT)
         return(0);
   }

/*-----------------------------------------------*/
If "no" then exit, else set the datum ad a gtol reference
/*-----------------------------------------------*/
if(!ref_datum)
   return(0);

   ProGeomitemGtolrefSet(&datum, NULL);

/*-----------------------------------------------*/
Remember the reference
Editing a Geometric Tolerance

Functions introduced:

- **ProGtolEditInit()**
- **ProGtolEditCommit()**
- **ProGtolEditCancel()**
- **ProGtoldataFree()**

A call to **ProGtolEditInit()** provides a *ProGtoldata* structure that describes the gtol being edited. Note that, in this case, the data is not in the Pro/TOOLKIT workspace, but is Pro/ENGINEER's own description of the gtol; calls to the **ProGtol*Set()** functions on this data 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()` function. This restores the gtol from a backup made during the call to `ProGtolEditInit()`.

Call `ProGtolEditCommit()` to tell Pro/ENGINEER you have finished modifying the gtol with `ProGtol*Set()` functions.

After calling `ProGtolEditCommit()` or `ProGtolEditCancel()`, call `ProGtoldataFree()` to free the workspace `ProGtoldata` structure.

As when creating 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, this will unset the placement. 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 this sequence:

`ProGtoldataReferenceGet()`
`ProGtoldataPlacementGet()`
`ProGtoldataReferenceSet()`
`ProGtoldataPlacementSet()`

This ensures you do not lose the placement information when `ProGtoldataReferenceSet()` is called.

### Deleting a Geometric Tolerance

Function introduced:

- `ProGtolDelete()`

The function `ProGtolDelete` permanently removes a gtol.
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.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview of Feature Creation</td>
<td>16 - 2</td>
</tr>
<tr>
<td>Feature Inquire</td>
<td>16 - 22</td>
</tr>
<tr>
<td>Feature Redefine</td>
<td>16 - 23</td>
</tr>
</tbody>
</table>
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:

- Creating Datum Features
- Creating Sketched Features
- Manufacturing

This chapter defines and describes the following Pro/TOOLKIT objects:

- **ProElement**
- **ProElempath**
- **ProValue**

For a definition of the **ProFeature** object, see the chapter ‘Basic Access to 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 Creating Sketched Features.

This method 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 \texttt{ProFeatureElemDimensionIdGet()}. This is explained in detail in the section Feature Inquire. For more general functions that access dimensions, see the chapter ‘Dimensions and 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 \texttt{ProElemId}, declared in \texttt{ProElemId.h}. Example values are as follows:

- \texttt{PRO_E_FEATURE_TYPE}
- \texttt{PRO_E_FEATURE_FORM}
- \texttt{PRO_E_EXT_DEPTH}
- \texttt{PRO_E_THICKNESS}
- \texttt{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 \texttt{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:

<table>
<thead>
<tr>
<th>Header File</th>
<th>Feature Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProAnalysis.h</td>
<td>External Analysis</td>
</tr>
<tr>
<td>ProAsmcomp.h</td>
<td>Assembly component</td>
</tr>
<tr>
<td>ProChamfer.h</td>
<td>Chamfer</td>
</tr>
<tr>
<td>ProDtmAxis.h</td>
<td>Datum axis</td>
</tr>
<tr>
<td>ProDtmCrv.h</td>
<td>Datum curve</td>
</tr>
<tr>
<td>ProDtmCsys.h</td>
<td>Datum coordinate system</td>
</tr>
<tr>
<td>ProDtmPln.h</td>
<td>Datum plane</td>
</tr>
<tr>
<td>ProDtmPnt.h</td>
<td>Datum Point</td>
</tr>
<tr>
<td>ProExtrude.h</td>
<td>Extruded protrusion, cut, surface, surface trim or thin</td>
</tr>
<tr>
<td>ProFixture.h</td>
<td>Fixture (for Pro/NC)</td>
</tr>
<tr>
<td>ProFlatSrf.h</td>
<td>Fill surface</td>
</tr>
<tr>
<td>ProForeignCurve.h</td>
<td>Foreign datum curve</td>
</tr>
<tr>
<td>ProHole.h</td>
<td>Hole</td>
</tr>
<tr>
<td>ProMfgoper.h</td>
<td>Manufacturing operation</td>
</tr>
<tr>
<td>ProNcseq.h</td>
<td>Manufacturing NC sequence</td>
</tr>
<tr>
<td>ProProcostep.h</td>
<td>Process step</td>
</tr>
<tr>
<td>ProReplace.h</td>
<td>Surface replacement feature</td>
</tr>
<tr>
<td>ProRevolve.h</td>
<td>Revolved protrusion, cut, surface, surface trim or thin</td>
</tr>
<tr>
<td>ProRib.h</td>
<td>Rib</td>
</tr>
<tr>
<td>ProSweep.h</td>
<td>Simple swept protrusion, cut</td>
</tr>
<tr>
<td>ProWcell.h</td>
<td>Manufacturing workcell</td>
</tr>
</tbody>
</table>
The feature element tree for a Fixture Setup feature, taken from *ProFixture.h*, looks like this:

Figure 16-1: Fixture Setup Feature Element Tree

```
PRO_E_FEATURE_TREE
  └── PRO_E_FEATURE_TYPE
  └── PRO_E_FEAT_NAME
      optional element
  └── PRO_E_FIXT_COMPONENTS
      optional element - multiple values
  └── PRO_E_SETUP_TIME
      optional element
```

The corresponding table of feature elements is as follows:

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Element Name</th>
<th>Data Type</th>
<th>Valid Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_FEATURE_TYPE</td>
<td>Feature type</td>
<td>INT</td>
<td>PRO_FEAT_FIXSETUP</td>
</tr>
<tr>
<td>PRO_E_FEAT_NAME</td>
<td>Feature name</td>
<td>WSTRING</td>
<td>Feature name</td>
</tr>
<tr>
<td>PRO_E_FIXT_COMPONENTS</td>
<td>Fixture components</td>
<td>INT ARRAY</td>
<td>Fixture component identifiers</td>
</tr>
<tr>
<td>PRO_E_SETUP_TIME</td>
<td>Setup time</td>
<td>DOUBLE</td>
<td>Number of minutes</td>
</tr>
</tbody>
</table>

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/TOLKIT 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
• 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()
- ProValueArrayToWorldStringArray()
- 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(). These functions are described later in this chapter.

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:

```c
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.

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 inorder 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/TOLKIT 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:

```c
typedef struct path
{
    ProElempathItem 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()`
- `ProElementValueGet()`
- `ProElementValuesGet()`
- `ProElementValueSet()`
- `ProElementValuesSet()`
- `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 `ProElementValueGet()` provides a `ProValue` object with the value of a single-valued element. The function `ProElementValuesGet()` provides a `ProValue` array of the values of a multivalued element. The array is an expandable array that must be allocated by a call to `ProArrayAlloc()` before you call the function.
The functions `ProElementValueSet()` and `ProElementValuesSet()` enable you to set the values of a single-valued element and a multivalued element, respectively. The inputs are in the form of `ProValue` objects.

The function `ProElementtreeElementGet()` 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 `ProElementtreeElementAdd()` 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 `ProElementtreeElementRemove()` 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 `ProElementtreeElementVisit()` visits the elements that are members of the specified array element in an element tree.
Calling ProFeatureCreate()

Function introduced:

- **ProFeatureCreate()**

  The syntax of **ProFeatureCreate()** is as follows:

  ```c
  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:

```c
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
{
```
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.

### 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 16-2: Example Element Tree: Offset Datum Plane**
The full element tree for datum plane features is described in the chapter 'Creating 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:

```c
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()`:

```c
/*---------------------------------------------------------------*\
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.

```c
/*---------------------------------------------------------------*\
Allocate the feature type element.
\*---------------------------------------------------------------*/
ProElementAlloc (PRO_E_FEATURE_TYPE, &elem_ftype);
/*---------------------------------------------------------------*\nSet 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);
/*---------------------------------------------------------------*\nAdd 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).

```c
/*---------------------------------------------------------------*/
Add a PRO_E_DTMPLN_CONSTRAINTS element to the root of the tree.

ProElementAlloc (PRO_E_DTMPLN_CONSTRAINTS, &elem_consts);
ProElemtreeElementAdd (elem_tree, NULL, elem_consts);
/*---------------------------------------------------------------*/
Add a PRO_E_DTMPLN_CONSTRAINT element to the constraints element.

ProElementAlloc (PRO_E_DTMPLN_CONSTRAINT, &elem_offset);
ProElemtreeElementAdd (elem_consts, NULL, elem_offset);

Inside the single constraint element, add an element of type PRO_E_DTMPLN_CONSTR_TYPE that specifies the constraint type to be PRO_DTMPLN_OFFS.

/*---------------------------------------------------------------*/
Allocate the constraint type element.

ProElementAlloc (PRO_E_DTMPLN_CONSTR_TYPE, &elem_const_type);
/*---------------------------------------------------------------*/
Set its value to be PRO_DTMPLN_OFFS.

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_DTMPLN_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_DTMPLN_CONSTR_REF, with value type ProSelection.

/*---------------------------------------------------------------*/
Allocate the offset reference element.

ProElementAlloc (PRO_E_DTMPLN_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);

Finally, you need an element of type
PRO_E_DTMPLN_CONSTR_REF_OFFSET to contain the double value of the offset distance.

Allocate the offset distance element.

ProElementAlloc (PRO_E_DTMPLN_CONSTR_REF_OFFSET,
                &elem_offset_dist);

Set its value to be the offset distance.

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.

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 Inquire

Functions introduced:

- `ProFeatureElemtreeCreate()`
- `ProFeatureElemValueGet()`
- `ProFeatureElemValuesGet()`
- `ProFeatureElemDimensionIdGet()`
- `ProFeatureElemIsVisible()`
- `ProFeatureIsIncomplete()`
- `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.

The function `ProFeatureElemtreeCreate()` creates a 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 described in the section Feature Elements and Feature Element Paths, such as `ProElement*Get()`, `ProElement*Visit()`, and `ProElementArrayCount()`.

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 `ProFeatureElemtreeCreate()` to get a copy of the element tree.

2. Analyze and modify the tree using functions `ProFeatureElem*()`, `ProElement*()`, and `ProElempath*()`.

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()`.
Creating 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 ‘Principles of Feature Creation’ provides necessary background for creating datum features; we recommend you read that material first.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datum Plane Features</td>
<td>17 - 2</td>
</tr>
<tr>
<td>Datum Point Features</td>
<td>17 - 12</td>
</tr>
<tr>
<td>Datum Axis Features</td>
<td>17 - 50</td>
</tr>
<tr>
<td>Datum Coordinate System Features</td>
<td>17 - 62</td>
</tr>
</tbody>
</table>
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.

Many elements forming the constraint element PRO_E_DTMPLN_CONSTRAINT are used only for the following constraint types:

- PRO_E_DTMPLN_CONSTRAINT_REF_OFFSET—Used if the constraint type is “offset.”
- PRO_E_DTMPLN_CONSTRAINT_REF_ANGLE—Used if the constraint type is “angle.”
- PRO_E_DTMPLN_CONSTRAINT_SEC_IND—Used if the constraint type is “section.”
- PRO_E_DTMPLN.Offset.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_DTMPN_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.

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Element Name</th>
<th>Data Type</th>
<th>Valid Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_FEATURE_TYPE</td>
<td>Feature type</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>PRO_FEAT_DATUM</td>
</tr>
<tr>
<td>PRO_E_STD_FEATURE_NAME</td>
<td>Feature Name</td>
<td>PRO_VALUE_TYPE_WSTRING</td>
<td></td>
</tr>
<tr>
<td>PRO_E_DTMPLN_CONSTRAINTS</td>
<td>Constraints</td>
<td>Array</td>
<td></td>
</tr>
<tr>
<td>PRO_E_DTMPLN_CONSTRAINT</td>
<td>Constraints</td>
<td>Compound</td>
<td></td>
</tr>
<tr>
<td>PRO_E_DTMPLN_CONSTR_TYPE</td>
<td>Type</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>See ProDtmplnConstrType</td>
</tr>
<tr>
<td>PRO_E_DTMPLN_CONSTR_REF</td>
<td>References</td>
<td>PRO_VALUE_TYPE_SELECTION</td>
<td>See Note : Constraint Reference Types</td>
</tr>
<tr>
<td>PRO_E_DTMPLN_CONSTR_REF_OFFSET</td>
<td>Offset</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
<td>Any</td>
</tr>
<tr>
<td>PRO_E_DTMPLN_CONSTR_REF_ANGLE</td>
<td>Angle</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
<td>(-360.0, 360.0)</td>
</tr>
<tr>
<td>PRO_E_DTMPLN_SEC_IND</td>
<td>Section index</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>[0, sec num - 1]</td>
</tr>
<tr>
<td>PRO_E_DTMPLN_OFF_CSYS</td>
<td>Offset coordinate system</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>See ProDtmplnOffCsyxAxis</td>
</tr>
<tr>
<td>Element ID</td>
<td>Element Name</td>
<td>Data Type</td>
<td>Valid Value</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-----------------------------------------</td>
<td>----------------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>PRO_E_DTMPLN_OFF_CSYS_OFFSET</td>
<td>Offset coordinate system value</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
<td>Any</td>
</tr>
<tr>
<td>PRO_E_DTMPLN_FLP_DIR</td>
<td>Flip direction</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>ProDtmplnFlipDir</td>
</tr>
<tr>
<td>PRO_E_DTMPLN_FIT</td>
<td>Fit</td>
<td>Compound</td>
<td>ProDtmplnFitType</td>
</tr>
<tr>
<td>PRO_E_DTMPLN_FIT_TYPE</td>
<td>Fit type</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>ProDtmplnFitType</td>
</tr>
<tr>
<td>PRO_E_DTMPLN_FIT_REF</td>
<td>Reference</td>
<td>PRO_VALUE_TYPE_SELECTION</td>
<td>See Note: Fit Reference Types</td>
</tr>
<tr>
<td>PRO_E_DTMPLN_FIT_DTM_RAD</td>
<td>Datum radius</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
<td>&gt;= 0.0</td>
</tr>
</tbody>
</table>
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:

**Note : Constraint Reference Types**

<table>
<thead>
<tr>
<th>Constraint Type</th>
<th>Valid Reference Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_DTMPLN_THRU</td>
<td>PRO_AXIS, PRO_EDGE, PRO_CURVE, Channel, PRO_POINT, PRO_EDGE_START, PRO_EDGE_END, PRO_CRV_END, PRO_SURFACE (Plane, Cylinder)</td>
</tr>
<tr>
<td>PRO_DTMPLN_NORM</td>
<td>PRO_AXIS, PRO_EDGE, PRO_CURVE, Channel PRO_SURFACE (plane)</td>
</tr>
<tr>
<td>PRO_DTMPLN_PRL</td>
<td>PRO_SURFACE (plane)</td>
</tr>
<tr>
<td>PRO_DTMPLN_OFFS</td>
<td>PRO_SURFACE (plane), PRO_CSYS</td>
</tr>
<tr>
<td>PRO_DTMPLN_ANG</td>
<td>PRO_SURFACE (plane)</td>
</tr>
<tr>
<td>PRO_DTMPLN_TANG</td>
<td>PRO_SURFACE (cylinder)</td>
</tr>
<tr>
<td>PRO_DTMPLN_SEC</td>
<td>PRO_FEATURE (blend)</td>
</tr>
<tr>
<td>PRO_DTMPLN_DEF_X</td>
<td>No reference needed</td>
</tr>
<tr>
<td>PRO_DTMPLN_DEF_Y</td>
<td>No reference needed</td>
</tr>
<tr>
<td>PRO_DTMPLN_DEF_Z</td>
<td>No reference needed</td>
</tr>
</tbody>
</table>
The following table describes the corresponding rules for the fit options in detail.

**Note: Fit Reference Types**

<table>
<thead>
<tr>
<th>Fit Type</th>
<th>Valid Reference Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_DTMPLN_FIT_DEFAULT</td>
<td>—</td>
</tr>
<tr>
<td>PRO_DTMPLN_FIT_PART</td>
<td>PRO_PART</td>
</tr>
<tr>
<td>PRO_DTMPLN_FIT_FEATURE</td>
<td>PRO_FEATURE</td>
</tr>
<tr>
<td>PRO_DTMPLN_FIT_SURFACE</td>
<td>PRO_SURFACE</td>
</tr>
<tr>
<td>PRO_DTMPLN_FIT_EDGE</td>
<td>PRO_EDGE</td>
</tr>
<tr>
<td>PRO_DTMPLN_FIT_AXIS</td>
<td>PRO_AXIS</td>
</tr>
<tr>
<td>PRO_DTMPLN_FIT_RADIUS</td>
<td>—</td>
</tr>
</tbody>
</table>

**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.

```c
/*================================================================*/
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, "USR 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_DTMPLN_CONSTRAINTS, &elem_consts);
ProElemtreeElementAdd (elem_tree, NULL, elem_consts);

/*----------------------------------------------------------------*\
Allocate the constraint element. Add it under constraints element.
\*----------------------------------------------------------------*/
status = ProElementAlloc (PRO_E_DTMPLN_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_DTMPLN_CONSTR_TYPE, elem_consts,
&elem_const_type);
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_DTMPLN_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_DTMPLN_CONSTR_REF, elem_offset,
&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_DTMPLN_CONSTR_REF_OFFSET, elem_offset_ref,
&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.

```
PRO_E_FEATURE_TYPE
  PRO_E_DPOINT_TYPE
  PRO_E_STD_FEATURE_NAME
  PRO_E_STD_SECTION
```

Define the following sub elements of PRO_E_STD_SECTION to complete the sketched datum point feature.

```
PRO_E_STD_SECTION
  PRO_E_STD_SEC_SETUP_PLANE
    PRO_E_STD_SEC_PLANES
      PRO_E_STD_SEC_PLANES_VIEW_DIR
      PRO_E_STD_SEC_PLANES_ORIENT_DIR
      PRO_E_STD_SEC_PLANES_ORIENT_REF
    PRO_E_SKETCHER
```
See the chapter ‘Creating 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.

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Element Name</th>
<th>Data Type</th>
<th>Valid Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_FEATURE_TYPE</td>
<td>Feature Type</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>PRO_FEAT_DATUM_POINT</td>
</tr>
<tr>
<td>PRO_E_DPOINT_TYPE</td>
<td>Datum Point Type</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>PRO_DPOINT_TYPE_SKETCHED</td>
</tr>
<tr>
<td>PRO_E_STD_FEATURE_NAME</td>
<td>Feature Name</td>
<td>PRO_VALUE_TYPE_WSTRING</td>
<td></td>
</tr>
<tr>
<td>PRO_E_STD_SECTION</td>
<td>Section</td>
<td>Compound</td>
<td>See ProStdSection.h</td>
</tr>
</tbody>
</table>

**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.

```c
/*====================================================================*\
Creating a Sketched Datum Point
\*====================================================================*/
#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_PRINT(a) printf ( "%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 );

="/="/---------------------------------------------------------------*/
    Populating element PRO_E_FEATURE_TYPE
"*/="/---------------------------------------------------------------*/
    C_PRINT( " *** Processing Element PRO_E_FEATURE_TYPE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TYPE, &pro_e_feature_type );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_FEAT_DATUM_POINT; /* 931 */
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_feature_type, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_feature_type );

="/="/---------------------------------------------------------------*/
    Populating element PRO_E_DPOINT_TYPE
"*/="/---------------------------------------------------------------*/
    C_PRINT( " *** Processing Element PRO_E_DPOINT_TYPE *** " );
status = ProElementAlloc ( PRO_E_DPOINT_TYPE, &pro_e_dpoint_type );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_DPOINT_TYPE_SKETCHED; /* 22 ProDPointType */
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_dpoint_type, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                pro_e_dpoint_type );

="/="/---------------------------------------------------------------*/
    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 );
ProStringToWstring ( wide_string, "MY_PNT1" );
value_data.type = PRO_VALUE_TYPE_WSTRING;
value_data.v.w = wide_string;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_feature_name, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                      pro_e_std_feature_name );
/*-----------------------------------------------*/
  // 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 );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                      pro_e_std_section );
/*-----------------------------------------------*/
  // 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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_SEC_SKETCH; ** 25  ProSecMethod **
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_sec_method, value );
status = ProElemtreeElementAdd ( pro_e_std_section, NULL,
                      pro_e_std_sec_method );
*/

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 );
status = ProElemtreeElementAdd ( pro_e_std_section, NULL,
                      pro_e_std_sec_setup_plane );
/*-----------------------------------------------*/
  // 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");
printf ( "Please select datum,surface,sldface,qltface_ID_5 type of
status = ProSelect ( "datum,surface,sldface,qltface", -1, NULL, NULL, NULL, NULL, &p_select, &n_select );
if ( n_select <= 0 ) return -1;
else
{
    status = ProSelectionCopy( p_select[0], &sketch_refs[0]);
}
status = ProElementAlloc ( PRO_E_STD_SEC_PLANE, &pro_e_std_sec_plane );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_sec_plane, value );
status = ProElementreeElementAdd ( pro_e_std_sec_setup_plane, NULL, pro_e_std_sec_plane );

/*---------------------------------------------------------------*
| 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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_SEC_VIEW_DIR_SIDE_ONE; /* 1 */
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_sec_plane_view_dir, value );
status = ProElementreeElementAdd ( pro_e_std_sec_setup_plane, NULL, pro_e_std_sec_plane_view_dir );

/*---------------------------------------------------------------*
| 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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_SEC_ORIENT_DIR_UP; /* 1 */
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_sec_plane_orient_dir, value );
status = ProElementreeElementAdd ( pro_e_std_sec_setup_plane, NULL, pro_e_std_sec_plane_orient_dir );
/*---------------------------------------------------------------*

Creating Datum Features 17 - 17
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");
printf ( "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);
if ( n_select <= 0 ) return -1;
else
{
  status = ProSelectionCopy( p_select[0], &sketch_refs[1]);
}

status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_ORIENT_REF,
  &pro_e_std_sec_plane_orient_ref );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_sec_plane_orient_ref, value );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
  pro_e_std_sec_plane_orient_ref );

/*---------------------------------------------------------------*\
  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);
/* Using the element tree from created feature */
status = ProFeatureElemtreeCreate ( &feature, &created_elemtree );

/*---------------------------------------------------------------*/
/* 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;
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.

```
  PRO_E_FEATURE_TYPE
  ├── PRO_E_DPOINT_TYPE
  │    └── PRO_E_STD_FEATURE_NAME
  │                    │
  │                    └── PRO_E_DPOINT_FIELD_REF
```

**Feature Elements**

The following table describes the elements of the element tree for the field datum points.

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Element Name</th>
<th>Data Type</th>
<th>Valid Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_FEATURE_TYPE</td>
<td>Feature Type</td>
<td>PRO_VALUE_TYP</td>
<td>PRO_FEAT_DATUM_POINT</td>
</tr>
<tr>
<td>PRO_E_DPOINT_TYPE</td>
<td>Datum Point Type</td>
<td>PRO_VALUE_TYP</td>
<td>PRO_DPOINT_TYPE_FIELD</td>
</tr>
<tr>
<td>PRO_E_STD_FEATURE_NAME</td>
<td>Feature Name</td>
<td>PRO_VALUE_TYP</td>
<td>Surface, Edge, Curve, or Quilt.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PRO_VALUE_TYP</td>
<td>Note: UV is used to specify exact</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PRO_VALUE_TYP</td>
<td>location.</td>
</tr>
<tr>
<td>PRO_E_DPOINT_FIELD_REF</td>
<td>Placement reference</td>
<td>PRO_VALUE_TYP</td>
<td>Surface, Edge, Curve, or Quilt.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PRO_VALUE_TYP</td>
<td>Note: UV is used to specify exact</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PRO_VALUE_TYP</td>
<td>location.</td>
</tr>
</tbody>
</table>

**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.

```c
/*====================================================================*/
Creating a Field Datum Curve
/*====================================================================*/

#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_PRINT(a) printf ( "%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];
    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_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

Populating element PRO_E_FEATURE_TYPE

Populating element PRO_E_DPOINT_TYPE

Populating element PRO_E_STD_FEATURE_NAME

Populating element PRO_E_DPOINT_FIELD_REF
Creating Datum Features

```c
C_PRINT(" *** Processing Element PRO_E_DPOINT_FIELD_REF *** ");
status = ProMessageDisplay(message_file,"Select a reference Surface");
printf("Please select datum,surface,sldface,qltface_ID_5 type of Modelitem\n");
status = ProSelect("curve,surface",-1,NULL,NULL,NULL,&p_select,&n_select);
if (n_select <= 0) return -1;
status = ProElementAlloc(PRO_E_DPOINT_FIELD_REF,
&pro_e_dpoint_field_ref);
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc(&value);
status = ProValueDataSet(value,&value_data);
status = ProElementValueSet(pro_e_dpoint_field_ref,value);
status = ProElemtreeElementAdd(pro_e_feature_tree,NULL,
pro_e_dpoint_field_ref);
/*-----------------------------------------------*/
    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);
status = ProElementFree(&pro_e_feature_tree);
return (status);
}
```

Offset Csrs Datum Point

An Offset Csrs 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, θ, Z axis.
- **Spherical**—Requires values along r, Φ, θ axis.
The following figure shows the element tree for Offset Csys Datum Point.

```
PRO_E_FEATURE_TYPE
  PRO_E_DPOINT_TYPE
  PRO_E_STD_FEATURE_NAME
  PRO_E_DPOINT_OFST_CSYS_TYPE
  PRO_E_DPOINT_OFST_CSYS_REF
  PRO_E_DPOINT_OFST_CSYS_WITH_DIMS
  PRO_E_DPOINT_OFST_CSYS_PNTS_ARRAY
    PRO_E_DPOINT_OFST_CSYS_PNT
      PRO_E_DPOINT_OFST_CSYS_PNT_NAME
      PRO_E_DPOINT_OFST_CSYS_DIR1_VAL
      PRO_E_DPOINT_OFST_CSYS_DIR2_VAL
      PRO_E_DPOINT_OFST_CSYS_DIR3_VAL
```

**Feature Elements**

The following table describes the elements in the element tree for datum points.

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Element Name</th>
<th>Data Type</th>
<th>Valid Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_FEATURE_TYPE</td>
<td>Feature Type</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>PRO_FEAT_DATUM_POINT</td>
</tr>
<tr>
<td>PRO_E_DPOINT_TYPE</td>
<td>Datum Point Type</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>PRO_DPOINT_TYPE_OFFSET_CSYS</td>
</tr>
<tr>
<td>PRO_E_STD_FEATURE_NAME</td>
<td>Feature Name</td>
<td>PRO_VALUE_TYPE_WSTRING</td>
<td></td>
</tr>
<tr>
<td>PRO_E_DPOINT_OFST_CSYS_TYPE</td>
<td>Reference Csys Type</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>See ProDtmpntOff CsysType</td>
</tr>
<tr>
<td>PRO_E_DPOINT_OFST_CSYS_REF</td>
<td>Reference Csys</td>
<td>PRO_VALUE_TYPE_SELECTION</td>
<td>Csys</td>
</tr>
</tbody>
</table>
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.

```c
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_ARR
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 \( \rho \)
PRO_VALUE_TYPE _DOUBLE
Depends on PRO_E_DPOINT_TYPE

PRO_E_DPOINT_OF ST_CSYS_DIR2_VAL
Y, \( \theta \), or \( \Phi \)
PRO_VALUE_TYPE _DOUBLE
Depends on PRO_E_DPOINT_TYPE

PRO_E_DPOINT_OF ST_CSYS_DIR3_VAL
Z, Z, or \( \theta \)
PRO_VALUE_TYPE _DOUBLE
Depends on PRO_E_DPOINT_TYPE
```

/*-----------------------------------------------*/
Creating an Offset Csys 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"

Creating Datum Features 17 - 25
#include "ProFeatType.h"
#include "ProFeatForm.h"
#include "ProSelection.h"
#include "ProSection.h"
#include "ProDtmPnt.h"

#define C_PRINT(a) printf ( "%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 );

/*---------------------------------------------------------------*/
Populating element PRO_E_FEATURE_TYPE
/*---------------------------------------------------------------*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TYPE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TYPE, &pro_e_feature_type );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_FEAT_DATUM_POINT; /* 931 */
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_feature_type, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL, pro_e_feature_type );

/*---------------------------------------------------------------*/
Populating element PRO_E_DPOINT_TYPE
/*---------------------------------------------------------------*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_TYPE *** " );
status = ProElementAlloc ( PRO_E_DPOINT_TYPE, &pro_e_dpoint_type );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_DPOINT_TYPE_OFFSET_CSYS; /* 6 */
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_dpoint_type, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL, pro_e_dpoint_type );

/*---------------------------------------------------------------*/
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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_DTMNT_OFFCYS_CARTESIAN; /* 0 */
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_dpoint_ofst_csys_type, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL, pro_e_dpoint_ofst_csys_type );

/*-------------------------------*/
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");
printf("Please select csys_ID_25 type of Modelitem\n");
status = ProSelect("csys", -1, NULL, NULL, NULL, &p_select, &n_select);
if (n_select <= 0) return -1;
status = ProElementAlloc (PRO_E_DPOINT_OFST_CSYS_REF, &pro_e_dpoint_ofst_csys_ref);
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc (&value);
status = ProValueDataSet (value, &value_data);
status = ProElementValueSet (pro_e_dpoint_ofst_csys_ref, value);
status = ProElemtreeElementAdd (pro_e_feature_tree, NULL, pro_e_dpoint_ofst_csys_ref);

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);
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_B_TRUE;
status = ProValueAlloc (&value);
status = ProValueDataSet (value, &value_data);
status = ProElementValueSet (pro_e_dpoint_ofst_csys_with_dims, value);
status = ProElemtreeElementAdd (pro_e_feature_tree, NULL, pro_e_dpoint_ofst_csys_with_dims);

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);
status = ProElemtreeElementAdd (pro_e_feature_tree, NULL, pro_e_dpoint_ofst_csys_pnts_array);

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);
status = ProElemtreeElementAdd ( pro_e_dpoint_ofst_cssys_pnts_array, NULL, pro_e_dpoint_ofst_cssys_pnt );

/*---------------------------------------------------------------*/
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_cssys_pnt_name );
ProStringToWstring ( wide_string, "PNT2" );
value_data.type = PRO_VALUE_TYPE_WSTRING;
value_data.v.w = wide_string;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_dpoint_ofst_cssys_pnt_name, value );
status = ProElemtreeElementAdd ( pro_e_dpoint_ofst_cssys_pnt, NULL, pro_e_dpoint_ofst_cssys_pnt_name );

/*---------------------------------------------------------------*/
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_cssys_dir1_val );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 100.000000;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_dpoint_ofst_cssys_dir1_val, value );
status = ProElemtreeElementAdd ( pro_e_dpoint_ofst_cssys_pnt, NULL, pro_e_dpoint_ofst_cssys_dir1_val );

/*---------------------------------------------------------------*/
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_cssys_dir2_val );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 200.000000;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_dpoint_ofst_csys_dir2_val, value );
status = ProElemtreeElementAdd ( pro_e_dpoint_ofst_csys_pnt, NULL,
 pro_e_dpoint_ofst_csys_dir2_val );

/**********************************************************************
 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 );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 300.000000;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_dpoint_ofst_csys_dir3_val, value );
status = ProElemtreeElementAdd ( pro_e_dpoint_ofst_csys_pnt, NULL,
 pro_e_dpoint_ofst_csys_dir3_val );

/**********************************************************************
 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);
status = ProElementFree (&pro_e_feature_tree );

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.

```
PRO_E_FEATURE_TYPE
  PRO_E_DPOINT_TYPE
    PRO_E_STD_FEATURE_NAME
      PRO_E_DPOINT_POINTS_ARRAY
        PRO_E_DPOINT_POINT
          PRO_E_DPOINT_POINT_NAME
        PRO_E_DPOINT_PLA_CONSTRAINTS
          PRO_E_DPOINT_PLA_CONSTRAINT
            PRO_E_DPOINT_PLA_CONSTRAINT_REF
            PRO_E_DPOINT_PLA_CONSTRAINT_TYPE
            PRO_E_DPOINT_PLA_CONSTRAINT_VAL
        PRO_E_DPOINT_DIM_CONSTRAINTS
          PRO_E_DPOINT_DIM_CONSTRAINT
            PRO_E_DPOINT_DIM_CONSTRAINT_REF
            PRO_E_DPOINT_DIM_CONSTRAINT_TYPE
            PRO_E_DPOINT_DIM_CONSTRAINT_VAL
```

**Feature Elements**

The following table describes the elements of the element tree for datum points.

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Element Name</th>
<th>Data Type</th>
<th>Valid Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_FEATURE_TYPE</td>
<td>Feature Type</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>PRO_FEAT_DIMENSION</td>
</tr>
<tr>
<td>PRO_E_DPOINT_TYPE</td>
<td>Datum Point Type</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>PRO_DPOINT_TYPE_GENERAL</td>
</tr>
<tr>
<td>PRO_E_STD_FEATURE_NAME</td>
<td>Feature Name</td>
<td>PRO_VALUE_TYPE_WSTRING</td>
<td></td>
</tr>
<tr>
<td>PRO_E_DPOINT_POINTS_ARRAY</td>
<td>Points List</td>
<td>Array</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

Pro/TOOLKIT User's Guide
### 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`

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Element Name</th>
<th>Data Type</th>
<th>Valid Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PRO_E_DPOINT_POINT</code></td>
<td>One Point</td>
<td>Compound</td>
<td>Not applicable</td>
</tr>
<tr>
<td><code>PRO_E_DPOINT_POINT_NAME</code></td>
<td>Point Name</td>
<td>PRO_VALUE_TYP E_WSTRING</td>
<td></td>
</tr>
<tr>
<td><code>PRO_E_DPOINT_PLA_CONSTRAINTS</code></td>
<td>Placement Constraints</td>
<td>Array</td>
<td>Not applicable</td>
</tr>
<tr>
<td><code>PRO_E_DPOINT_PLA_CONSTRAINT</code></td>
<td>One Placement Constraint</td>
<td>Compound</td>
<td>Not applicable</td>
</tr>
<tr>
<td><code>PRO_E_DPOINT_PLA_CONSTR_REF</code></td>
<td>Placement Reference</td>
<td>PRO_VALUE_TYP E_SELECTION</td>
<td>Depends on the context. See <code>PRO_E_DPOINT_PLA_CONSTR_REF</code>.</td>
</tr>
<tr>
<td><code>PRO_E_DPOINT_PLA_CONSTR_TYPE</code></td>
<td>Constraint Type</td>
<td>PRO_VALUE_TYP E_INT</td>
<td>See <code>ProDtmpntConstrType</code>.</td>
</tr>
<tr>
<td><code>PRO_E_DPOINT_DIM_CONSTRAINTS</code></td>
<td>Value</td>
<td>PRO_VALUE_TYP E_DOUBLE</td>
<td></td>
</tr>
<tr>
<td><code>PRO_E_DPOINT_DIM_CONSTRAINT</code></td>
<td>Dimension Constraint</td>
<td>Array</td>
<td>Not applicable</td>
</tr>
<tr>
<td><code>PRO_E_DPOINT_DIM_CONSTR_REF</code></td>
<td>One Dimension Constraint</td>
<td>Compound</td>
<td>Not applicable</td>
</tr>
<tr>
<td><code>PRO_E_DPOINT_DIM_CONSTR_TYPE</code></td>
<td>Dimension Reference</td>
<td>PRO_VALUE_TYPE_SELECTION</td>
<td>See Placement Constraint References.</td>
</tr>
<tr>
<td><code>PRO_E_DPOINT_DIM_CONSTR_REF</code></td>
<td>Constraint Type</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>Depends on the context. See Constraint Type.</td>
</tr>
<tr>
<td><code>PRO_E_DPOINT_DIM_CONSTRAINT</code></td>
<td>Value</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
<td>See <code>ProDtmpntConstrType</code>.</td>
</tr>
</tbody>
</table>
• 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_ALONG_X
- PRO_DTMPNT_CONSTR_TYPE_ALONG_Y
- PRO_DTMPNT_CONSTR_TYPE_ALONG_Z

**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
- CSYS—SEL_3D_CSYS
- Surface—SEL_3D_SRF, SEL_SRF_PNT, SEL_3D_SRF_LIST

**Constraint Type**

Valid values for PRO_E_DPOINT_DIM_CONSTR_TYPE are as follows:

- PRO_DTMPNT_CONSTR_TYPE_OFFSET
- PRO_DTMPNT_CONSTR_TYPE_LENGTH
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.

```c
#include "ProFeature.h"
#include "ProExtrude.h"
#include "ProElement.h"
#include "ProSelection.h"
#include "ProSection.h"
#include "ProDtmPnt.h"
#define C_PRINT(a) printf ( "%s\n", a);

static ProFileName message_file;

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" );

>Status = ProElementAlloc ( PRO_E_FEATURE_TREE, &pro_e_feature_tree );

C_PRINT(" *** Processing Element PRO_E_FEATURE_TREE *** ");
status = ProElementAlloc ( PRO_E_FEATURE_TREE, &pro_e_feature_tree );

C_PRINT(" *** Processing Element PRO_E_FEATURE_TYPE *** ");
status = ProElementAlloc ( PRO_E_FEATURE_TYPE, &pro_e_feature_type );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_FEAT_DATUM_POINT; /* 931 */
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_feature_type, value );
status = ProElementValueSet ( pro_e_feature_type, value );
status = ProElementValueSet ( pro_e_feature_type, value );
status = ProElementValueSet ( pro_e_feature_type, value );
status = ProElementValueSet ( pro_e_feature_type, value );
status = ProElementValueSet ( pro_e_feature_type, value );
status = ProElementValueSet ( pro_e_feature_type, value );
status = ProElementValueSet ( pro_e_feature_type, value );
Populating element PRO_E_DPOINT_TYPE
\*---------------------------------------------------------------*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_TYPE *** " );
status = ProElementAlloc ( PRO_E_DPOINT_TYPE, &pro_e_dpoint_type );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_DPOINT_TYPE_GENERAL; /* -1 */
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_dpoint_type, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_dpoint_type );

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 );
ProStringToWstring ( wide_string, "MY_PNT0" );
value_data.type = PRO_VALUE_TYPE_WSTRING;
value_data.v.w = wide_string;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_feature_name, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_std_feature_name );

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 );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_dpoint_points_array );

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 );
status = ProElemtreeElementAdd ( pro_e_dpoint_points_array, NULL,
pro_e_dpoint_point );

Populating element PRO_E_DPOINT_POINTS_ARRAY
-> PRO_E_DPOINT_POINT
-> PRO_E_DPOINT_POINT_NAME
\*---------------------------------------------------------------*/

Creating Datum Features 17 - 37
C_PRINT(" *** Processing Element PRO_E_DPOINT_POINT_NAME *** ");
status = ProElementAlloc ( PRO_E_DPOINT_POINT_NAME,
&pro_e_dpoint_point_name );
ProStringToWstring ( wide_string, "PNT0" );
value_data.type = PRO_VALUE_TYPE_WSTRING;
value_data.v.w = wide_string;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_dpoint_point_name, value );
status = ProElemtreeElementAdd ( pro_e_dpoint_point, NULL,
pro_e_dpoint_point_name );

/*---------------------------------------------------------------*
| 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 );
status = ProElemtreeElementAdd ( pro_e_dpoint_point, NULL,
pro_e_dpoint_pla_constraints );

/*---------------------------------------------------------------*
| 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 );
status = ProElemtreeElementAdd ( pro_e_dpoint_pla_constraints, NULL,
pro_e_dpoint_pla_constraint );

/*---------------------------------------------------------------*
| 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" );
printf("Please select datum,surface,slidfce,qltface_ID_5 type of
Modelitem\n");
status = ProSelect ( "datum,surface,slidfce,qltface", -1, NULL, NULL,
NULL, NULL, &p_select, &n_select );
if ( n_select <= 0 ) return -1;
status = ProElementAlloc ( PRO_E_DPOINT_PLA_CONSTR_REF,
Creating Datum Features

&pro_e_dpoint_pla_constr_ref );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_dpoint_pla_constr_ref, value );
status = ProElemtreeElementAdd ( pro_e_dpoint_pla_constraint, NULL,
pro_e_dpoint_pla_constr_ref );

/*---------------------------------------------------------------*/
Populating element PRO_E_DPOINT_POINTS_ARRAY
    -> PRO_E_DPOINT_POINT
    -> PRO_E_DPOINT_PLA_CONSTRAINTS
    -> PRO_E_DPOINT_PLACONSTRAINT
    -> 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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_DTMPNT_CONSTR_TYPE_ON; /* 0  ProDtmpntConstrType */
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_dpoint_pla_constr_type, value );
status = ProElemtreeElementAdd ( pro_e_dpoint_pla_constraint, NULL,
pro_e_dpoint_pla_constr_type );

/*---------------------------------------------------------------*/
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 );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 0.000000;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_dpoint_pla_constr_val, value );
status = ProElemtreeElementAdd ( pro_e_dpoint_pla_constraint, NULL, 
        pro_e_dpoint_pla_constr_val );

/*---------------------------------------------------------------*/
| 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 );
status = ProElemtreeElementAdd ( pro_e_dpoint_pla_constraints, NULL, 
        pro_e_dpoint_pla_constraint );

/*---------------------------------------------------------------*/
| 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 = ProMessageDisplay ( message_file, "Select a reference Surface");
printf ( "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 );
if ( n_select <= 0 ) return -1;
status = ProElementAlloc ( PRO_E_DPOINT_PLA_CONSTRAINT, 
        &pro_e_dpoint_pla_constr_ref );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_dpoint_pla_constr_ref, value );
status = ProElemtreeElementAdd ( pro_e_dpoint_pla_constraint, NULL, 
        pro_e_dpoint_pla_constr_ref );

/*---------------------------------------------------------------*/
| 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_TYPE, 
        &pro_e_dpoint_pla_constr_type );
value_data.type = PRO_VALUE_TYPE_INT;

17 - 40 Pro/TOOLKIT User's Guide
value_data.v.i = PRO_DTMPTN_CONST_TYPE_ON; /* 0 ProDtmpntConstrType */
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_dpoint_pla_constr_type, value );
status = ProElementTreeElementAdd ( pro_e_dpoint_pla_constr_type, NULL,
pro_e_dpoint_pla_constr_type );

/*---------------------------------------------------------------*
     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 );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 0.000000;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_dpoint_pla_constr_val, value );
status = ProElementTreeElementAdd ( pro_e_dpoint_pla_constraint, NULL,
pro_e_dpoint_pla_constr_val );

/*---------------------------------------------------------------*
     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_CONSTRAINT *** " );
status = ProElementAlloc ( PRO_E_DPOINT_PLA_CONSTRAINT,
&pro_e_dpoint_pla_constr );
status = ProElementTreeElementAdd ( pro_e_dpoint_pla_constraints, NULL,
pro_e_dpoint_pla_constr );

/*---------------------------------------------------------------*
     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_CONSTRAINT *** " );
status = ProElementAlloc ( PRO_E_DPOINT_PLA_CONSTRAINT,
&pro_e_dpoint_pla_constr );
status = ProElementTreeElementAdd ( pro_e_dpoint_pla_constraints, NULL,
pro_e_dpoint_pla_constr );

status = ProMessageDisplay ( message_file, "Select a reference Surface" );
printf ( "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 );
if ( n_select <= 0 ) return -1;

status = ProElementAlloc ( PRO_E_DPOINT_PLA_CONSTRAINT_REF,
                        &pro_e_dpoint_pla_constr_ref );

value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];

status = ProValueAlloc ( &value );

status = ProValueDataSet ( value, &value_data );

status = ProElementValueSet ( pro_e_dpoint_pla_constr_ref, value );

status = ProElemtreeElementAdd ( pro_e_dpoint_pla_constraint, NULL,
                        pro_e_dpoint_pla_constr_ref );

/*---------------------------------------------------------------*/
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_CONSTRAINT_TYPE
/*---------------------------------------------------------------*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_PLA_CONSTRAINT *** ");

status = ProElementAlloc ( PRO_E_DPOINT_PLA_CONSTRAINT_TYPE,
                        &pro_e_dpoint_pla_constr_type );

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_DTMPNT CONSTRAIN TYPE ON; /* 0 ProDtmpntConstrType */

status = ProValueAlloc ( &value );

status = ProValueDataSet ( value, &value_data );

status = ProElementValueSet ( pro_e_dpoint_pla_constr_type, value );

status = ProElemtreeElementAdd ( pro_e_dpoint_pla_constraint, NULL,
                        pro_e_dpoint_pla_constr_type );

/*---------------------------------------------------------------*/
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_CONSTRAINT_VAL
/*---------------------------------------------------------------*/
C_PRINT( " *** Processing Element PRO_E_DPOINT_PLA_CONSTRAINT *** ");

status = ProElementAlloc ( PRO_E_DPOINT_PLA_CONSTRAINT_VAL,
                        &pro_e_dpoint_pla_constr_val );

value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 0.000000;

status = ProValueAlloc ( &value );

status = ProValueDataSet ( value, &value_data );

status = ProElementValueSet ( pro_e_dpoint_pla_constr_val, value );

status = ProElemtreeElementAdd ( pro_e_dpoint_pla_constraint, NULL,
                        pro_e_dpoint_pla_constr_val );

/*---------------------------------------------------------------*/
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);

status = ProElementFree (&pro_e_feature_tree );
return (status);
}

Examples

Point on a Vertex

To create a datum point on the vertex, the following constraints are required.

<table>
<thead>
<tr>
<th>Placement Constraint Element</th>
<th>Reference Element</th>
<th>Valid Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_DPOINT_PLA_CONSTRAINT</td>
<td>PRO_E_DPOINT_PLLONSTR_REF</td>
<td>Vertex</td>
</tr>
<tr>
<td>(Constraint 1)</td>
<td>PRO_E_DPOINT_PLLONSTR_TYPE</td>
<td>PRO_DTMPNT_PLLONSTR_TYPE_ON</td>
</tr>
<tr>
<td></td>
<td>PRO_E_DPOINT_PLLONSTR_VAL</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

Offset Point

To create one or more datum points at an offset, the following constraints are required.

<table>
<thead>
<tr>
<th>Placement Constraint Element</th>
<th>Reference Element</th>
<th>Valid Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_DPOINT_PLLONSTR_CONSTRAINT</td>
<td>PRO_E_DPOINT_PLLONSTR_REF</td>
<td>Vertex, Csys, or DPnt</td>
</tr>
<tr>
<td>(Constraint 1)</td>
<td>PRO_E_DPOINT_PLLONSTR_TYPE</td>
<td>PRO_DTMPNT_PLLONSTR_TYPE_OFFSET</td>
</tr>
<tr>
<td></td>
<td>PRO_E_DPOINT_PLLONSTR_VAL</td>
<td>Offset value.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Placement Constraint Element</th>
<th>Reference Element</th>
<th>Valid Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_DPOINT_PLA_CONSTRAINT</td>
<td>PRO_E_DPOINT_PLA_CONSTR_REF</td>
<td>Curve, Edge or Axis</td>
</tr>
<tr>
<td></td>
<td>PRO_E_DPOINT_PLA_CONSTR_TYPE</td>
<td>PRO_DTMPNT_CONSTR_TYPE_PARALLEL</td>
</tr>
<tr>
<td></td>
<td>PRO_E_DPOINT_PLA_CONSTR_VAL</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

OR

<table>
<thead>
<tr>
<th>Placement Constraint Element</th>
<th>Reference Element</th>
<th>Valid Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_DPOINT_PLA_CONSTRAINT</td>
<td>PRO_E_DPOINT_PLA_CONSTR_REF</td>
<td>Surface</td>
</tr>
<tr>
<td></td>
<td>PRO_E_DPOINT_PLA_CONSTR_TYPE</td>
<td>PRO_DTMPNT_CONSTR_TYPE_NORMAL</td>
</tr>
<tr>
<td></td>
<td>PRO_E_DPOINT_PLA_CONSTR_VAL</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

OR

<table>
<thead>
<tr>
<th>Placement Constraint Element</th>
<th>Reference Element</th>
<th>Valid Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_DPOINT_PLA_CONSTRAINT</td>
<td>PRO_E_DPOINT_PLA_CONSTR_REF</td>
<td>Cs ys</td>
</tr>
<tr>
<td></td>
<td>PRO_E_DPOINT_PLA_CONSTR_TYPE</td>
<td>PRO_DTMPNT_CONSTR_TYPE_ALONG_X or PRO_DTMPNT_CONSTR_TYPE_ALONG_Y or PRO_DTMPNT_CONSTR_TYPE_ALONG_Z</td>
</tr>
<tr>
<td></td>
<td>PRO_E_DPOINT_PLA_CONSTR_VAL</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>
**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.

<table>
<thead>
<tr>
<th>Placement Constraint Element</th>
<th>Reference Element</th>
<th>Valid Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_DPOINT_PLA_CONSTRAINT</td>
<td>PRO_E_DPOINT_PLA_CO NSTR_REF</td>
<td>Surface</td>
</tr>
<tr>
<td>(Constraint 1)</td>
<td>PRO_E_DPOINT_PLA_CO NSTR_TYPE</td>
<td>PRO_DTMPNT_CONSTR_TYPE_ON</td>
</tr>
<tr>
<td></td>
<td>PRO_E_DPOINT_PLA_CO NSTR_VAL</td>
<td>Not applicable</td>
</tr>
<tr>
<td>PRO_E_DPOINT_PLA_CONSTRAINT</td>
<td>PRO_E_DPOINT_PLA_CO NSTR_REF</td>
<td>Surface</td>
</tr>
<tr>
<td>(Constraint 2)</td>
<td>PRO_E_DPOINT_PLA_CO NSTR_TYPE</td>
<td>PRO_DTMPNT_CONSTR_TYPE_ON</td>
</tr>
<tr>
<td></td>
<td>PRO_E_DPOINT_PLA_CO NSTR_VAL</td>
<td>Not applicable</td>
</tr>
<tr>
<td>PRO_E_DPOINT_PLA_CONSTRAINT</td>
<td>PRO_E_DPOINT_PLA_CO NSTR_REF</td>
<td>Surface</td>
</tr>
<tr>
<td>(Constraint 3)</td>
<td>PRO_E_DPOINT_PLA_CO NSTR_TYPE</td>
<td>PRO_DTMPNT_CONSTR_TYPE_ON</td>
</tr>
<tr>
<td></td>
<td>PRO_E_DPOINT_PLA_CO NSTR_VAL</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>Placement Constraint Element</th>
<th>Reference Element</th>
<th>Valid Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_DPOINT_PLA_CONSTRAINT</td>
<td>PRO_E_DPOINT_PLA_CO NSTR_REF</td>
<td>Surface</td>
</tr>
<tr>
<td>(Constraint 1)</td>
<td>PRO_E_DPOINT_PLA_CO NSTR_TYPE</td>
<td>PRO_DTMPNT_CONSTR_TYPE_ON or PRO_DTMPNT_CONSTR_TYPE_OFFSET</td>
</tr>
<tr>
<td></td>
<td>PRO_E_DPOINT_PLA_CO NSTR_VAL</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Placement Constraint Element</th>
<th>Reference Element</th>
<th>Valid Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_DPOINT_PLA_CONSTRAINT (Constraint 2)</td>
<td>PRO_E_DPOINT_PLA_CONSTRAINT_REF</td>
<td>Edge or Surface</td>
</tr>
<tr>
<td></td>
<td>PRO_E_DPOINT_PLA_CONSTRAINT_TYPE</td>
<td>PRO_DTMPNT_CONSTRAINT_TYPE_OFFSET</td>
</tr>
<tr>
<td></td>
<td>PRO_E_DPOINT_PLA_CONSTRAINT_VAL</td>
<td>Offset value</td>
</tr>
<tr>
<td>PRO_E_DPOINT_PLA_CONSTRAINT (Constraint 3)</td>
<td>PRO_E_DPOINT_PLA_CONSTRAINT_REF</td>
<td>Edge or Surface</td>
</tr>
<tr>
<td></td>
<td>PRO_E_DPOINT_PLA_CONSTRAINT_TYPE</td>
<td>PRO_DTMPNT_CONSTRAINT_TYPE_OFFSET</td>
</tr>
<tr>
<td></td>
<td>PRO_E_DPOINT_PLA_CONSTRAINT_VAL</td>
<td>Offset value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Placement Constraint Element</th>
<th>Reference Element</th>
<th>Valid Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_DPOINT_PLA_CONSTRAINT (Constraint 1)</td>
<td>PRO_E_DPOINT_PLA_CONSTRAINT_REF</td>
<td>Curve, axis, edge, or surface</td>
</tr>
<tr>
<td></td>
<td>PRO_E_DPOINT_PLA_CONSTRAINT_TYPE</td>
<td>PRO_DTMPNT_CONSTRAINT_TYPE_ON</td>
</tr>
<tr>
<td></td>
<td>PRO_E_DPOINT_PLA_CONSTRAINT_VAL</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Placement Constraint Element</th>
<th>Reference Element</th>
<th>Valid Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_DPOINT_PLACONSTRAINT (Constraint 2)</td>
<td>PRO_E_DPOINT_PLA_CONSTRAINT</td>
<td>- 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.</td>
</tr>
<tr>
<td>PRO_E_DPOINT_PLA_CONSTRAINT</td>
<td>PRO_DTMPNT_CONSTRAINT_TYPE</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Placement Constraint Element</th>
<th>Reference Element</th>
<th>Valid Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_DPOINT_PLA_CONSTRAINT (Constraint 1)</td>
<td>PRO_E_DPOINT_PLA_CONSTRAINT</td>
<td>Curve, edge, or surface (Sphere)</td>
</tr>
<tr>
<td>PRO_E_DPOINT_PLA_CONSTRAINT</td>
<td>PRO_DTMPNT_CONSTRAINT_TYPE_CENTER</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>
Point at Intersection of Two Curves

To create a point at intersection of two curves, use the following constraints.

<table>
<thead>
<tr>
<th>Placement Constraint Element</th>
<th>Reference Element</th>
<th>Valid Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_DPOINT_PLA_CONSTRAINT</td>
<td>PRO_E_DPOINT_PLA_CONSTRAINT</td>
<td>Curve, edge, or axis</td>
</tr>
<tr>
<td>(Constraint 1)</td>
<td>PRO_E_DPOINT_PLA_CONSTRAINT</td>
<td>PRO_DTMPNT_CONSTRAINT_TYPE_ON</td>
</tr>
<tr>
<td></td>
<td>PRO_E_DPOINT_PLA_CONSTRAINT</td>
<td>Not applicable</td>
</tr>
<tr>
<td>PRO_E_DPOINT_PLA_CONSTRAINT</td>
<td>PRO_E_DPOINT_PLA_CONSTRAINT</td>
<td>Curve, edge, or axis</td>
</tr>
<tr>
<td>(Constraint 2)</td>
<td>PRO_E_DPOINT_PLA_CONSTRAINT</td>
<td>PRO_DTMPNT_CONSTRAINT_TYPE_ON</td>
</tr>
<tr>
<td></td>
<td>PRO_E_DPOINT_PLA_CONSTRAINT</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Placement Constraint Element</th>
<th>Reference Element</th>
<th>Valid Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_DPOINT_PLA_CONSTRAINT</td>
<td>PRO_E_DPOINT_PLA_CONSTRAINT</td>
<td>Curve, edge, or axis (It is valid with offset plane)</td>
</tr>
<tr>
<td>(Constraint 1)</td>
<td>PRO_E_DPOINT_PLA_CONSTRAINT</td>
<td>PRO_DTMPNT_CONSTRAINT_TYPE_ON</td>
</tr>
<tr>
<td></td>
<td>PRO_E_DPOINT_PLA_CONSTRAINT</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Placement Constraint Element</th>
<th>Reference Element</th>
<th>Valid Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_DPOINT_PLA_CONSTRAINT (Constraint 2)</td>
<td>PRO_E_DPOINT_PLA_CONSTRAINT_REF</td>
<td>Curve (Use the same curve as used in Constraint 1)</td>
</tr>
<tr>
<td>PRO_E_DPOINT_PLA_CONSTRAINT_TYPE</td>
<td>PRO_DTMPNT_CONSTRAINT_TYPE_LENGTH or PRO_DTMPNT_CONSTRAINT_TYPE_LENGTH_END</td>
<td></td>
</tr>
<tr>
<td>PRO_E_DPOINT_PLA_CONSTRAINT_VAL</td>
<td>Length value (from curve start point or end point)</td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Placement Constraint Element</th>
<th>Reference Element</th>
<th>Valid Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_DPOINT_PLA_CONSTRAINT (Constraint 2)</td>
<td>PRO_E_DPOINT_PLA_CONSTRAINT_REF</td>
<td>Curve (Use the same curve as used in contrarian 1)</td>
</tr>
<tr>
<td>PRO_E_DPOINT_PLA_CONSTRAINT_TYPE</td>
<td>PRO_DTMPNT_CONSTRAINT_TYPE_RATIO or PRO_DTMPNT_CONSTRAINT_TYPE_RATIO_END</td>
<td></td>
</tr>
<tr>
<td>PRO_E_DPOINT_PLA_CONSTRAINT_VAL</td>
<td>Ratio value (from curve start or end)</td>
<td></td>
</tr>
</tbody>
</table>

Use the following values for constraint 2 if the offset surface is used to locate the point on curve.

<table>
<thead>
<tr>
<th>Placement Constraint Element</th>
<th>Reference Element</th>
<th>Valid Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_DPOINT_PLA_CONSTRAINT (Constraint 2)</td>
<td>PRO_E_DPOINT_PLA_CONSTRAINT_REF</td>
<td>Surface</td>
</tr>
<tr>
<td>PRO_E_DPOINT_PLA_CONSTRAINT_TYPE</td>
<td>PRO_DTMPNT_CONSTRAINT_TYPE_OFFSET</td>
<td></td>
</tr>
<tr>
<td>PRO_E_DPOINT_PLA_CONSTRAINT_VAL</td>
<td>Offset value</td>
<td></td>
</tr>
</tbody>
</table>
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_E_FEATURE_TREE
  └── PRO_E_FEATURE_TYPE
  └── PRO_E_STD_FEATURE_NAME
      └── PRO_E_DTMAXIS_CONSTRAINTS
          └── PRO_E_DTMAXIS_CONSTRAINT
              └── PRO_E_DTMAXIS_CONSTR_TYPE
              └── PRO_E_DTMAXIS_CONSTR_REF
          └── PRO_E_DTMAXIS_DIM_CONSTRAINTS
              └── PRO_E_DTMAXIS_DIM_CONSTRAINT
                  └── PRO_E_DTMAXIS_DIM_CONSTR_REF
                  └── PRO_E_DTMAXIS_DIM_CONSTR_VAL
```

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.

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Data Type</th>
<th>Valid Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_DTMAXIS_CONSTRAINT (Constraint 1)</td>
<td>PRO_E_DTMAXIS_CONSTRAINT_TYPE</td>
<td>PRO_DTMAXIS_CONSTRAINT_TYPE_THRU</td>
</tr>
<tr>
<td></td>
<td>PRO_E_DTMAXIS_CONSTRAINT_REF</td>
<td>PRO_POINT</td>
</tr>
<tr>
<td>PRO_E_DTMAXIS_CONSTRAINT (Constraint 2)</td>
<td>PRO_E_DTMAXIS_CONSTRAINT_TYPE</td>
<td>PRO_DTMAXIS_CONSTRAINT_TYPE_NORMAL</td>
</tr>
<tr>
<td></td>
<td>PRO_E_DTMAXIS_CONSTRAINT_REF</td>
<td>PRO_SURFACE</td>
</tr>
</tbody>
</table>
Tangent

The element tree structure of the axis, created with type as Tangent, is shown in the following figure.

![Element Tree Diagram]

The following table specifies the constraints for the PRO_E_DTMAXIS_CONSTRAINT elements in the element tree for the tangent type of axis.

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Data Type</th>
<th>Valid Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_DTMAXIS_CONSTRAINT (Constraint 1)</td>
<td>PRO_E_DTMAXIS_CONSTRAINT_TYPE</td>
<td>PRO_DTMAXIS_CONSTRAINT_TYPE_TANGENT</td>
</tr>
<tr>
<td></td>
<td>PRO_E_DTMAXIS_CONSTRAINT_REF</td>
<td>PRO_EDGE, PRO_CURVE</td>
</tr>
<tr>
<td>PRO_E_DTMAXIS_CONSTRAINT (Constraint 2)</td>
<td>PRO_E_DTMAXIS_CONSTRAINT_TYPE</td>
<td>PRO_DTMAXIS_CONSTRAINT_TYPE_THRU</td>
</tr>
<tr>
<td></td>
<td>PRO_E_DTMAXIS_CONSTRAINT_REF</td>
<td>PRO_POINT, PRO_EDGE_START, PRO_EDGE_END, PRO_CRV_START, PRO_CRV_END</td>
</tr>
</tbody>
</table>
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.

![Element Tree Structure]

The following table specifies the constraints for the PRO_E_DTMAXIS_CONSTRAINT elements in the element tree for the through type of axis.

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Data Type</th>
<th>Valid Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_DTMAXIS_CONSTRAINT (Constraint 1)</td>
<td>PRO_E_DTMAXIS_CONSTRAINT_TYPE</td>
<td>PRO_DTMAXIS_CONSTRAINT_TYPE_THRU</td>
</tr>
<tr>
<td></td>
<td>PRO_E_DTMAXIS_CONSTRAINT_REF</td>
<td>PRO_EDGE (Straight), PRO_SURFACE (Cylinder)</td>
</tr>
</tbody>
</table>
Two Planes

The element tree structure of the axis, created using the type as two planes, is as shown in the following figure.

![Element Tree Structure](image)

The following table specifies the constraints for the `PRO_E_DTMAXIS_CONSTRAINT` elements in the element tree for the two planes reference scheme.

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Data Type</th>
<th>Valid Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_DTMAXIS_CONSTRAINT (Constraint 1)</td>
<td>PRO_E_DTMAXIS_CONSTR_TYPE</td>
<td>PRO_DTMAXIS_CONSTRAINT_TYPE_THRU</td>
</tr>
<tr>
<td>PRO_E_DTMAXIS_CONSTRAINT (Constraint 1)</td>
<td>PRO_E_DTMAXIS_CONSTR_REF</td>
<td>PRO_SURFACE (Planar)</td>
</tr>
<tr>
<td>PRO_E_DTMAXIS_CONSTRAINT (Constraint 2)</td>
<td>PRO_E_DTMAXIS_CONSTR_TYPE</td>
<td>PRO_DTMAXIS_CONSTRAINT_TYPE_THRU</td>
</tr>
<tr>
<td>PRO_E_DTMAXIS_CONSTRAINT (Constraint 2)</td>
<td>PRO_E_DTMAXIS_CONSTR_REF</td>
<td>PRO_SURFACE (Planar)</td>
</tr>
</tbody>
</table>
Two Points

The element tree structure of the axis, created using the type as two points, is shown in the following figure.

```
PRO_E_DTMAXIS_CONSTRAINT
  ├── PRO_E_DTMAXIS_CONSTRAINT_TYPE
  │    └── PRO_E_DTMAXIS_CONSTRAINT_REF
  │        └── PRO_E_DTMAXIS_CONSTRAINT
  │            └── PRO_E_DTMAXIS_CONSTRAINT_TYPE
  │                └── PRO_E_DTMAXIS_CONSTRAINT_REF
```

The following table specifies the constraints for the PRO_E_DTMAXIS_CONSTRAINT elements in the element tree for the two points type of axis.

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Data Type</th>
<th>Valid Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_DTMAXIS_CONSTRAINT (Constraint 1)</td>
<td>PRO_E_DTMAXIS_CONSTRAINT_TYPE</td>
<td>PRO_DTMAXIS_CONSTRAINT_TYPE_THRU</td>
</tr>
<tr>
<td></td>
<td>PRO_E_DTMAXIS_CONSTRAINT_REF</td>
<td>PRO_POINT, PRO_EDGE_START, PRO_EDGE_END, PRO_CRV_START, PRO_CRV_END</td>
</tr>
<tr>
<td>PRO_E_DTMAXIS_CONSTRAINT (Constraint 2)</td>
<td>PRO_E_DTMAXIS_CONSTRAINT_TYPE</td>
<td>PRO_DTMAXIS_CONSTRAINT_TYPE_THRU</td>
</tr>
<tr>
<td></td>
<td>PRO_E_DTMAXIS_CONSTRAINT_REF</td>
<td>PRO_POINT, PRO_EDGE_START, PRO_EDGE_END, PRO_CRV_START, PRO_CRV_END</td>
</tr>
</tbody>
</table>
Normal Planes

The element tree structure of the axis, created using the type as Normal Planes, is shown in the following figure.

```
PRO_E_DTMAXIS_CONSTRAINT
  └── PRO_E_DTMAXIS_CONSTR_TYPE
  └── PRO_E_DTMAXIS_CONSTR_REF

PRO_E_DTMAXIS_DIM_CONSTRAINT
  └── PRO_E_DTMAXIS_DIM_CONSTR_REF
  └── PRO_E_DTMAXIS_DIM_CONSTR_VAL

PRO_E_DTMAXIS_CONSTRAINT
  └── PRO_E_DTMAXIS_DIM_CONSTR_REF
  └── PRO_E_DTMAXIS_DIM_CONSTR_VAL
```

The following table specifies the constraints for the PRO_E_DTMAXIS_CONSTRAINT elements in the element tree for the normal plane type of axis.

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Data Type</th>
<th>Valid Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_DTMAXIS_CONSTRAINT (Constraint 1)</td>
<td>PRO_E_DTMAXIS_CONSTR_TYPE</td>
<td>PRO_DTMAXIS_CONSTR_TYPE_NORMAL</td>
</tr>
<tr>
<td></td>
<td>PRO_E_DTMAXIS_CONSTR_REF</td>
<td>PRO_SURFACE (Planar)</td>
</tr>
<tr>
<td>PRO_E_DTMAXIS_DIM_CONSTRAINT (Constraint 2)</td>
<td>PRO_E_DTMAXIS_DIM_CONSTR_REF</td>
<td>PRO_SURFACE (Planar), PRO_AXIS, PRO_EDGE</td>
</tr>
<tr>
<td></td>
<td>PRO_E_DTMAXIS_DIM_CONSTR_VAL</td>
<td>Valid dimension</td>
</tr>
<tr>
<td>PRO_E_DTMAXIS_DIM_CONSTRAINT (Constraint 3)</td>
<td>PRO_E_DTMAXIS_DIM_CONSTR_REF</td>
<td>PRO_SURFACE (Planar), PRO_AXIS, PRO_EDGE</td>
</tr>
<tr>
<td></td>
<td>PRO_E_DTMAXIS_DIM_CONSTR_VAL</td>
<td>Valid dimension</td>
</tr>
</tbody>
</table>
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.

```c
/*====================================================================*
Creating a Datum Axis feature
\*====================================================================*/
#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_PRINT(a) printf ( "%s\n", a);
static ProFileName message_file;

logfile = "\"DatumAxis.log\"

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];
```

Creating Datum Features
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 );
/*---------------------------------------------------------------*/
/*- Populating element PRO_E_FEATURE_TYPE */
C_PRINT( " *** Processing Element PRO_E_FEATURE_TYPE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TYPE, &pro_e_feature_type );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_FEAT_DATUM_AXIS; /* 926 */
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_feature_type, value );

/*---------------------------------------------------------------*/
/*- 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 );
width_string = "MY_A_1";
value_data.type = PRO_VALUE_TYPE_WSTRING;
value_data.v.w = width_string;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_feature_name, value );
status = ProElementTreeElementAdd ( pro_e_feature_tree, NULL, pro_e_std_feature_name );

/*---------------------------------------------------------------*
Populating element PRO_E_DTMAXIS_CONSTRAINTS
*/
status = ProElementAlloc ( PRO_E_DTMAXIS_CONSTRAINTS, &pro_e_dtmaxis_constraints );
status = ProElementTreeElementAdd ( pro_e_feature_tree, NULL, pro_e_dtmaxis_constraints );

/*---------------------------------------------------------------*
Populating element PRO_E_DTMAXIS_CONSTRAINTS
-> PRO_E_DTMAXIS_CONSTRAINT
*/
status = ProElementAlloc ( PRO_E_DTMAXIS_CONSTRAINT, &pro_e_dtmaxis_constraint );
status = ProElementTreeElementAdd ( pro_e_dtmaxis_constraints, NULL, pro_e_dtmaxis_constraint );

/*---------------------------------------------------------------*
Populating element PRO_E_DTMAXIS_CONSTRAINTS
-> PRO_E_DTMAXIS_CONSTRAINT
-> PRO_E_DTMAXIS_CONSTR_TYPE
*/

C_PRINT (" *** Processing Element PRO_E_DTMAXIS_CONSTRAINT *** ");
status = ProElementAlloc ( PRO_E_DTMAXIS_CONSTR_TYPE, &pro_e_dtmaxis_constr_type );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_DTMAXIS_CONSTR_TYPE_THRU; /* 1 ProDtmaxisConstrType */
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_dtmaxis_constr_type, value );
status = ProElementTreeElementAdd ( pro_e_dtmaxis_constraint, NULL, pro_e_dtmaxis_constr_type );

/*---------------------------------------------------------------*
Populating element PRO_E_DTMAXIS_CONSTRAINTS
-> PRO_E_DTMAXIS_CONSTRAINT
-> PRO_E_DTMAXIS_CONSTR_REF
*/
status = ProMessageDisplay ( message_file, "Select a reference Surface");
printf ( "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 );
if ( n_select <= 0 ) return -1;
status = ProElementAlloc ( PRO_E_DTMAXIS_CONSTRAINT_REF,
    &pro_e_dtmaxis_constr_ref );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_dtmaxis_constr_ref, value );
status = ProElemtreeElementAdd ( pro_e_dtmaxis_constr_ref, NULL,
    pro_e_dtmaxis_constr_ref );

/*---------------------------------------------------------------*
   Populating element PRO_E_DTMAXIS_CONSTRAINTS
      -> PRO_E_DTMAXIS_CONSTRAINT
         -> PRO_E_DTMAXIS_CONSTRAINT_REF

   */---------------------------------------------------------------*/
C_PRINT( " *** Processing Element PRO_E_DTMAXIS_CONSTRAINT *** " );
status = ProElementAlloc ( PRO_E_DTMAXIS_CONSTRAINT,
    &pro_e_dtmaxis_constraint );
status = ProElemtreeElementAdd ( pro_e_dtmaxis_constraints, NULL,
    pro_e_dtmaxis_constraint );

/*---------------------------------------------------------------*
   Populating element PRO_E_DTMAXIS_CONSTRAINTS
      -> PRO_E_DTMAXIS_CONSTRAINT
         -> PRO_E_DTMAXIS_CONSTRAINT_TYPE

   */---------------------------------------------------------------*/
C_PRINT( " *** Processing Element PRO_E_DTMAXIS_CONSTRAINT_TYPE *** " );
status = ProElementAlloc ( PRO_E_DTMAXIS_CONSTRAINT_TYPE,
    &pro_e_dtmaxis_constr_type );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_DTMAXIS_CONSTR_TYPE_THRU; /* 1 ProDtmaxisConstrType */
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_dtmaxis_constr_type, value );
status = ProElemtreeElementAdd ( pro_e_dtmaxis_constraint, NULL,
    pro_e_dtmaxis_constr_type );

/*---------------------------------------------------------------*/
C_PRINT( " *** Processing Element PRO_E_DTMAXIS_CONSTRAINT *** " );
status = ProMessageDisplay ( message_file, "Select a reference Surface" );
printf ( "Please select datum,surface,sldface,qltface_ID_5 type of Modelitem\n" );
status = ProSelect ( "datum,surface,sldface,qltface_ID_5", -1, NULL,
    NULL,
Creating Datum Features

    NULL, NULL, &p_select, &n_select );
    if ( n_select <= 0 ) return -1;
    status = ProElementAlloc ( PRO_E_DTMAXIS_CONSTR_REF,
                               &pro_e_dtmaxis_constr_ref);
    value_data.type = PRO_VALUE_TYPE_SELECTION;
    value_data.v.r = p_select[0];
    status = ProValueAlloc ( &value );
    status = ProValueDataSet ( value, &value_data );
    status = ProElementValueSet ( pro_e_dtmaxis_constr_ref, value );
    status = ProElemtreeElementAdd ( pro_e_dtmaxis_constraint, NULL,
                                     pro_e_dtmaxis_constr_ref );

    /*-----------------------------------------------------------------------*
    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);

    status = ProElementFree (pro_e_feature_tree );

    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_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
```

Feature Elements

The following table describes the elements in the element tree for coordinate system feature.

<table>
<thead>
<tr>
<th>Element Id</th>
<th>Element Name</th>
<th>Data Type</th>
<th>Valid Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_FEATURE_TYPE</td>
<td>Feature Type</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>PRO_FEAT_CSYS</td>
</tr>
<tr>
<td>PRO_E_STD_FEATURE_NAME</td>
<td>Feature Name</td>
<td>PRO_VALUE_TYPE_WSTRING</td>
<td>-</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIGIN_CONSTRAINTS</td>
<td>Origin Constraints</td>
<td>Array</td>
<td>-</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIGIN_CONSTRAINT</td>
<td>Compound</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIGIN_CONSTRAINT_RE</td>
<td>Origin Reference</td>
<td>PRO_VALUE_TYPE_SELECTION</td>
<td>-</td>
</tr>
<tr>
<td>PRO_E_CSYS_OFFSET_TYPE</td>
<td>Origin Offset Type</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>ProCsSysOffsetType</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIENTMOVES</td>
<td>Orientation Moves</td>
<td>Array</td>
<td>-</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIENTATION</td>
<td>Compound</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIENTATION_MOVE_TYPE</td>
<td>Move Type</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>ProCsSysOrientMoveMoveOpt</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIENTATION_MOVE_VAL</td>
<td>Move Value</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
<td>-</td>
</tr>
<tr>
<td>PRO_E_CSYS_NORMAL_TO_SCREEN</td>
<td>Set Z Normal To Screen</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>ProCsSysOrientMovesNrmScrnOpt</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIENTATION_BY_METHOD</td>
<td>Orient By Method</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>ProCsSysOrientByMethod</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIENTATION_SELAXIS1_REF</td>
<td>First Axis Reference</td>
<td>PRO_VALUE_TYPE_SELECTION</td>
<td>-</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIENTATION_SELAXIS1_REF_OPT</td>
<td>First Axis Ref Option</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>ProCsSysDirecSysRefOpt</td>
</tr>
</tbody>
</table>
The following elements are common for all the cases of the coordinate system feature creation:

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_FEATURE_TYPE</td>
<td>PRO_FEAT_CSYS</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_STD_FEATURE_NAME</td>
<td>Feature Name</td>
<td>Optional</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_CSYS_ORIGIN_CONSTRS</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIGIN_CONSTR</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIGIN_CONSTR_REF</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIENTSELEXIS1_REF</td>
<td>Optional, using default if not set</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIENTSELEXIS1_REF_OPT</td>
<td>Optional, using default if not set</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_CSYS_ORIGIN_CONSTRS</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIGIN_CONSTR</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIGIN_CONSTR_REF</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIENTSELAXIS1_REF</td>
<td>Optional, using default if not set</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIENTSELAXIS1_REF_OPT</td>
<td>Optional, using default if not set</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIENTSELAXIS1_OPT</td>
<td>Mandatory if PRO_E_CSYS_ORIENTSELAXIS 1_REF is a Csys reference</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIENTSELAXIS1_FLIP</td>
<td>Optional</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIENTSELAXIS2_REF</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIENTSELAXIS2_REF_OPT</td>
<td>Mandatory, using default if not set</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIENTSELAXIS1_OPT</td>
<td>Mandatory if PRO_E_CSYS_ORIENTSELAXIS 2_REF is a Csys reference</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIENTSELAXIS2_FLIP</td>
<td>Optional</td>
</tr>
<tr>
<td>Others</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_CSYS_ORIGIN_CONSTRAINTS</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIGIN_CONSTRAINT</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIGIN_CONSTRAINT_REF</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIENTATIONSELAXIS1_REF</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIENTATIONSELAXIS1_REF_OPT</td>
<td>Optional, using default if not set</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIENTATIONSELAXIS1_OPT</td>
<td>Mandatory if PRO_E_CSYS_ORIENTATIONSELAXIS 1_REF is a Csys reference</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIENTATIONSELAXIS1_FLIP</td>
<td>Optional</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIENTATIONSELAXIS2_REF</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIENTATIONSELAXIS2_REF_OPT</td>
<td>Optional, using default if not set</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIENTATIONSELAXIS2_OPT</td>
<td>Mandatory if PRO_E_CSYS_ORIENTATIONSELAXIS 2_REF is a Csys reference</td>
</tr>
<tr>
<td>Others</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

Orienting by Selecting References

Use the following elements if

PRO_E_CSYS_ORIENTATION_METHOD is

PRO_CSYS_ORIENTATION_SEL_REFS:

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_CSYS_ORIENTATIONSELAXIS1_REF</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIENTATIONSELAXIS1_REF_OPT</td>
<td>Optional, using default if not set</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIENTATIONSELAXIS1_OPT</td>
<td>Optional, using default if not set</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIENTATIONSELAXIS1_FLIP</td>
<td>Optional, using default if not set</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIENTATIONSELAXIS2_REF</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIENTATIONSELAXIS2_REF_OPT</td>
<td>Optional, using default if not set</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIENTATIONSELAXIS2_OPT</td>
<td>Optional, using default if not set</td>
</tr>
</tbody>
</table>
Orienting by Selecting Csys Axes

Use the following elements if:
- PRO_E_CSYS_ORIENT_BY_METHOD is
- PRO_CSYS_ORIENT_BY_SEL_CSYS_AXES:

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_CSYS_NORMAL_TO_SCREEN</td>
<td>Optional, valid only if PRO_E_CSYS_ORIENT_BY_METHOD = PRO_CSYS_ORIENT_BY_SEL_CSYS_AXES. Otherwise, it is ignored.</td>
</tr>
<tr>
<td>Others</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

Using a Csys

Use the following elements if the origin of the coordinate system is determined using a Csys:

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_CSYS_ORIGIN_CONSTR</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIGIN_CONSTR_REF</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_CSYS_OFFSET_TYPE</td>
<td>Optional, using default PRO_CSYS_OFFSET_CARTESIAN if not set</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIENTMOVES</td>
<td>Mandatory for non PRO_CSYS_OFFSET_CARTESIAN</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIENTMOVE</td>
<td>Mandatory for non PRO_CSYS_OFFSET_CARTESIAN</td>
</tr>
<tr>
<td>PRO_E_CSYS_ORIENTMOVE_MOVE_TYPE</td>
<td>Mandatory for non PRO_CSYS_OFFSET_CARTESIAN</td>
</tr>
</tbody>
</table>
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, without the extension. The name must be lowercase only. The file should contain a coordinate transform such as:

\[
\begin{array}{cccc}
X1 & X2 & X3 & Tx \\
Y1 & Y2 & Y3 & Ty \\
Z1 & Z2 & Z3 & Tz \\
\end{array}
\]

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.

```c
/*====================================================================*\
Creating a Datum Coordinate System
\*====================================================================*/
#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_PRINT(a) printf ( "%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;

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 );
/*---------------------------------------------------------------*/
/*------------------- Populating element PRO_E_FEATURE_TYPE
                     */
C_PRINT( " *** Processing Element PRO_E_FEATURE_TYPE *** ");
status = ProElementAlloc ( PRO_E_FEATURE_TYPE, &pro_e_feature_type );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_FEAT_CSYS; /* 979 */
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_feature_type, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL, pro_e_feature_type );
/*---------------------------------------------------------------*/
/*------------------- 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 );
ProStringToWstring ( wide_string, "MY_CS0" );
value_data.type = PRO_VALUE_TYPE_WSTRING;
value_data.v.w = wide_string;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_feature_name, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_std_feature_name );

/*---------------------------------------------------------------*
| 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 );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_csys_origin_constrs );

/*---------------------------------------------------------------*
| 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 );
status = ProElemtreeElementAdd ( pro_e_csys_origin_constrs, NULL,
pro_e_csys_origin_constr );

/*---------------------------------------------------------------*
| 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");
printf ( "Please select csys_ID_25 type of Modelitem\n"");
status = ProSelect ( "csys", -1, NULL, NULL, NULL, &p_select,
&n_select );
if ( n_select <= 0 ) return -1;
status = ProElementAlloc ( PRO_E_CSYS_ORIGIN_CONSTR_REF,
&pro_e_csys_origin_constr_ref );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_csys_origin_constr_ref, value );
status = ProElemtreeElementAdd ( pro_e_csys_origin_constr, NULL,
pro_e_csys_origin_constr_ref );

/*--------------------------------*/

Creating Datum Features

17 - 71
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  );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = 0;
status = ProValueAlloc ( &value  );
status = ProValueDataSet ( value, &value_data  );
status = ProElementValueSet ( pro_e_csys_offset_type, value  );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                 pro_e_csys_offset_type  );

/*---------------------------------------------------------------*/
Populating element PRO_E_CSYS_ORIENTMOVES

/*---------------------------------------------------------------*/
C_PRINT( " *** Processing Element PRO_E_CSYS_ORIENTMOVES *** " );
status = ProElementAlloc ( PRO_E_CSYS_ORIENTMOVES,
                           &pro_e_csys_orientmoves  );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                 pro_e_csys_orientmoves  );

/*---------------------------------------------------------------*/
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  );
status = ProElemtreeElementAdd ( pro_e_csys_orientmoves, NULL,
                                 pro_e_csys_orientmove  );

/*---------------------------------------------------------------*/
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  );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_CSYS_ORIENTMOVE_MOVE_OPT_ROT_X; /* 3 */
status = ProValueAlloc ( &value  );
status = ProValueDataSet ( value, &value_data  );
status = ProElementValueSet ( pro_e_csys_orientmove_move_type, value  );
status = ProElemtreeElementAdd ( pro_e_csys_orientmove, NULL,
                                 pro_e_csys_orientmove_move_type  );

/*---------------------------------------------------------------*/
Populating element PRO_E_CSYS_ORIENTMOVES

Pro/TOOLKIT User's Guide
-> 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 );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 0.000000;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_csys_orientmove_move_val, value );
status = ProElemtreeElementAdd ( pro_e_csys_orientmove, NULL,
  pro_e_csys_orientmove_move_val );

/*---------------------------------------------------------------*/
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 );
status = ProElemtreeElementAdd ( pro_e_csys_orientmoves, NULL,
  pro_e_csys_orientmove );

/*---------------------------------------------------------------*/
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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_CSYS_ORIENTMOVE_MOVE_OPT_ROT_Y; /* 4 */
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_csys_orientmove_move_type, value );
status = ProElemtreeElementAdd ( pro_e_csys_orientmove, NULL,
  pro_e_csys_orientmove_move_type );

/*---------------------------------------------------------------*/
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 );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = -90.000000;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_csys_orientmove_move_val, value );
status = ProElementValueSet ( pro_e_csys_orientmove, NULL,
    pro_e_csys_orientmove_move_val );

/*---------------------------------------------------------------*
  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 );
status = ProElementAdd ( pro_e_csys_orientmoves, NULL,
    pro_e_csys_orientmove );

/*---------------------------------------------------------------*
  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 *** " );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_CSYS_ORIENTMOVE_MOVE_OPT_ROT_Z; /* 5 */
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_csys_orientmove_move_type, value );
status = ProElementAdd ( pro_e_csys_orientmove, NULL,
    pro_e_csys_orientmove_move_type );

/*---------------------------------------------------------------*
  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 *** " );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 0.000000;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_csys_orientmove_move_val, value );
status = ProElementAdd ( pro_e_csys_orientmove, NULL,
    pro_e_csys_orientmove_move_val );

17 - 74 Pro/TOOLKIT User's Guide
Creating Datum Features

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 );
status = ProElemtreeElementAdd ( pro_e_csys_orientmoves, NULL,
                               pro_e_csys_orientmove );

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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_CSYS.ORIENTMOVE_MOVE_OPT_TRAN_X; /* 0 */
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_csys_orientmove_move_type, value );
status = ProElemtreeElementAdd ( pro_e_csys_orientmove, NULL,
                               pro_e_csys_orientmove_move_type );

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 );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 100.000000;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_csys_orientmove_move_val, value );
status = ProElemtreeElementAdd ( pro_e_csys_orientmove, NULL,
                               pro_e_csys_orientmove_move_val );

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 );
status = ProEltreeElementAdd ( pro_e_csys_orientmoves, NULL,
    pro_e_csys_orientmove );

_REPEAT_DELIM
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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_CSYS_ORIENTMOVE_MOVE_OPT_TRAN_Y; /* 1 */
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_csys_orientmove_move_type, value );
status = ProEltreeElementAdd ( pro_e_csys_orientmove, NULL,
    pro_e_csys_orientmove_move_type );

_REPEAT_DELIM
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 );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 200.000000;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_csys_orientmove_move_val, value );
status = ProEltreeElementAdd ( pro_e_csys_orientmove, NULL,
    pro_e_csys_orientmove_move_val );

_REPEAT_DELIM
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 );
status = ProEltreeElementAdd ( pro_e_csys_orientmoves, NULL,
    pro_e_csys_orientmove );

_REPEAT_DELIM
Populating element PRO_E_CSYS_ORIENTMOVES
Creating Datum Features

- 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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_CSYS_ORIENTMOVE_MOVE_OPT_TRAN_Z; /* 2 */
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_csys_orientmove_move_type, value );
status = ProElemtreeElementAdd ( pro_e_csys_orientmove, NULL,
pro_e_csys_orientmove_move_type );

/*---------------------------------------------------------------*
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 );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 300.000000;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_csys_orientmove_move_val, value );
status = ProElemtreeElementAdd ( pro_e_csys_orientmove, NULL,
pro_e_csys_orientmove_move_val );

/*---------------------------------------------------------------*
Creating the feature in the current model.

/*------------------------------------------*/
status = ProMd1CurrentGet (&model);
if ( status != PRO_TK_NO_ERROR ) return ( status );
status = ProMd1ToModelitem( 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);

status = ProElementFree (&pro_e_feature_tree );
return ( status );
}
Creating Datum Curves

This chapter describes how to create, redefine, and access data for datum curve features using Pro/TOOLKIT. The chapter ‘Creating Datum Features’ provides necessary background for creating features; we recommend you read that material first.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datum Curve Features</td>
<td>18 - 2</td>
</tr>
<tr>
<td>Datum Curve Types</td>
<td>18 - 3</td>
</tr>
<tr>
<td>Other Datum Curve Types</td>
<td>18 - 8</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_FEATURE_TYPE</td>
<td>PRO_FEAT_CURVE</td>
</tr>
<tr>
<td>PRO_E_CURVE_TYPE</td>
<td>As listed in ProCurveType. This element identifies the subtree to be used.</td>
</tr>
<tr>
<td>PRO_E_STD_FEATURE_NAME</td>
<td>Wstring (feature name)</td>
</tr>
</tbody>
</table>
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 ‘Creating Sketched Features’.

<table>
<thead>
<tr>
<th>Table 2: Elements for Sketched Datum Curve Features</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Element ID</strong></td>
</tr>
<tr>
<td>PRO_E_CURVE_TYPE</td>
</tr>
<tr>
<td>PRO_E_STD_SECTION</td>
</tr>
<tr>
<td>PRO_E_DTMCRV_DISPLAY_HATCH</td>
</tr>
<tr>
<td>PRO_E_DTMCRV_HATCH_DENSITY</td>
</tr>
</tbody>
</table>

Trim Datum Curves

Pro/TOOLKIT provides complete element tree access to trim datum curves (previously called Split datum curve).

<table>
<thead>
<tr>
<th>Table 3: Elements for Trim Datum Curve Features</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Element ID</strong></td>
</tr>
<tr>
<td>PRO_E_CURVE_TYPE</td>
</tr>
<tr>
<td>PRO_E_STD_CRV_SPLIT_CURVE</td>
</tr>
</tbody>
</table>
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 ‘Creating Sketched Features’.

### Table 3: Elements for Trim Datum Curve Features

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_STD_CRV_SPLIT_SIDE</td>
<td>One of the ProSplitSides enumerations</td>
</tr>
</tbody>
</table>

### Table 4: Elements for Intersection of Surfaces Datum Curve Features

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_CURVE_TYPE</td>
<td>PRO_CURVE_TYPE_INTSRF</td>
</tr>
<tr>
<td>PRO_E_CRV_IP_REF_TYPE</td>
<td>PRO_CURVE_TYPE_INTSRF</td>
</tr>
<tr>
<td>PRO_E_CRV_IP_COMP_REF1</td>
<td>Compound</td>
</tr>
<tr>
<td>PRO_E_CRV_IP_REF_SEL1_TYPE</td>
<td>PRO_CURVE_TYPE_WHOLE for the whole surface selection; PRO_CURVE_TYPE_MULTIPLE_SEL for multiple independent surface selections.</td>
</tr>
</tbody>
</table>
Creating Datum Curves

Based on the value of `PRO_E_CRV_IP_SEL1_TYPE`.

- If the value is *whole*, 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 4: Elements for Intersection of Surfaces Datum Curve**

<table>
<thead>
<tr>
<th>Features</th>
<th>Element ID</th>
<th>Value</th>
</tr>
</thead>
</table>
|                                   | PRO_E_CRV_IP_REF1   | Based on the value of `PRO_E_CRV_IP_SEL1_TYPE`.
|                                   |                     | • If the value is *whole*, 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. |
|                                   | PRO_E_CRV_IP_COMP_REF2 | Compound                                                           |
|                                   | PRO_E_CRV_IP_REF_SEL2_TYPE | PRO_CURVE_TYPE_WHOLE for the *whole* surface selection; PRO_CURVE_TYPE_MULTIPLE_SEL for multiple independent surface selections. |
|                                   | PRO_E_CRV_IP_REF2   | Based on the value of `PRO_E_CRV_IP_SEL1_TYPE`.
|                                   |                     | • If the value is *whole*, 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**

<table>
<thead>
<tr>
<th>Features</th>
<th>Element ID</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PRO_E_CURVE_TYPE</td>
<td>PRO_CURVE_TYPE_TWO_PROJ</td>
</tr>
<tr>
<td></td>
<td>PRO_E_CRV_IP_REF_TYPE</td>
<td>PRO_CURVE_TYPE_TWO_PROJ</td>
</tr>
<tr>
<td></td>
<td>PRO_E_CRV_IP_COMP_SEC1</td>
<td>Compound</td>
</tr>
<tr>
<td></td>
<td>PRO_E_STD_SECTION</td>
<td>Section element tree</td>
</tr>
<tr>
<td></td>
<td>PRO_E_CRV_IP_COMP_SEC2</td>
<td>Compound</td>
</tr>
<tr>
<td></td>
<td>PRO_E_STD_SECTION</td>
<td>Section element tree</td>
</tr>
</tbody>
</table>
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 ‘Creating Sketched Features’.

Table 6: Elements for Wrap Datum Curve Features

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_CURVE_TYPE</td>
<td>PRO_CURVE_TYPE_WRAP</td>
</tr>
<tr>
<td>PRO_E_CRV_WRAP_SRF_TYPE</td>
<td>One of ProWrapSrfType</td>
</tr>
<tr>
<td>PRO_E_CRV_WRAP_SRF</td>
<td>Selection containing the wrap surface (surface, quilt or solid geometry)</td>
</tr>
<tr>
<td>PRO_E_STD_SECTION</td>
<td>Section</td>
</tr>
<tr>
<td>PRO_E_CRV_WRAP_FLIP</td>
<td>One of ProWrapFlip</td>
</tr>
<tr>
<td>PRO_E_CRV_WRAP_COORD_SYS</td>
<td>ID of the section coordinate system</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_CURVE_TYPE</td>
<td>PRO_CURVE_TYPE_OFFSET</td>
</tr>
<tr>
<td>PRO_E_CRV_OFFS_FEAT_TYPE</td>
<td>PRO_OFFSET_FROM_SURF ACE</td>
</tr>
<tr>
<td>PRO_E_CRV_OFFS_SRF_REF</td>
<td>Selection of surface or quilt</td>
</tr>
<tr>
<td>PRO_E_CRV_OFFS_DIR_FLIP</td>
<td>One of ProOffsetDirFlip</td>
</tr>
<tr>
<td>PRO_DATUM_CURVE_OFFSET_VAL</td>
<td>The offset value or scale if a graph is used for offset</td>
</tr>
<tr>
<td>PRO_E_CRV_OFFS_CRV_REF</td>
<td>Selection of datum curve to be offset</td>
</tr>
<tr>
<td>PRO_E_CRV_OFFS_GRAPH_REF</td>
<td>Selection of graph used for offset calculation (optional)</td>
</tr>
<tr>
<td>PRO_E_CRV_OFFS_ST_END</td>
<td>One of ProOffsetStEnd</td>
</tr>
</tbody>
</table>

**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**

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_CURVE_TYPE</td>
<td>PRO_CURVE_TYPE_TANGENT_OFFSET</td>
</tr>
<tr>
<td>PRO_E_CRV_TANG_OFFSET_CURVE</td>
<td>Selection of curve to be offset</td>
</tr>
<tr>
<td>PRO_E_CRV_TANG_OFFSET_SURF</td>
<td>Selection of surface in which to create the offset</td>
</tr>
<tr>
<td>PRO_E_CRV_TANG_OFFSET_DIR</td>
<td>One of ProOffsetDirection</td>
</tr>
</tbody>
</table>
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.

Table 8: Elements for Tangent Offset Datum Curve Features

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_CRV_TANG_OFFSET_DIST</td>
<td>Offset value</td>
</tr>
</tbody>
</table>
Creating Chamfer Features

This chapter describes the basic principles of programmatic creation of chamfer features. The chapter ‘Principles of Feature Creation’ is a necessary background for this topic. Read that chapter before this one.

Note that this release of Pro/TOOLKIT supports programmatic creation and access of edge and corner chamfers. See the Part Modeling User’s Guide for details on chamfer creation in Pro/ENGINEER.

The element tree for edge chamfer features is documented in the header file ProChamfer.h. The following sections describe the feature creation process for edge chamfers in detail.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edge Chamfers in Pro/ENGINEER Wildfire</td>
<td>19 - 2</td>
</tr>
<tr>
<td>Corner Chamfers</td>
<td>19 - 11</td>
</tr>
</tbody>
</table>
Edge Chamfers in Pro/ENGINEER Wildfire

In Pro/ENGINEER Wildfire, the element tree for edge chamfers is redesigned. As some radical changes are made to this element tree, existing edge chamfers are still stored using the Release 2001 element tree (they are not redefined to use the new chamfer elements). Also, the new element tree contains application elements, those cannot be currently accessed by Pro/TOOLKIT.

Consider the following points while working with edge chamfers:

• Pro/TOOLKIT allows creation of chamfers using the release 2001 element tree. These chamfers use the old Pro/ENGINEER interface for redefinition.

• To access and redefine chamfers created in Release 2001, use the Release 2001 element tree.

• Pro/TOOLKIT may not currently create chamfers using the new Wildfire element tree.

• Pro/TOOLKIT may not currently access or redefine the element trees for Wildfire created chamfer features. The function `ProFeatureElemtreeCreate()` returns `PRO_TK_INVALID_TYPE` when attempting to access a Wildfire based chamfer.

The following descriptions and examples are relevant for the Release 2001 chamfer element tree.

The Feature Element Tree for Edge Chamfers in Pro/ENGINEER Release 2001

The same four dimension schemes used to create edge chamfers interactively in Pro/ENGINEER are available in Pro/TOOLKIT:

• 45 x d
• d x d
• d1 x d2
• Ang x d

The general element tree for edge chamfer features contains all the elements required to create chamfers using any of the four dimension schemes.

The element tree for an edge chamfer feature is documented in the header file `ProChamfer.h`, and is shown in the following figure.
For a given dimension scheme, you need only a subset of the elements in the general tree. The following table describes the necessary elements for each of the four dimension schemes used to create edge chamfers.

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Data Type</th>
<th>Values and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>For dimension schemes 45 x d and d x d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRO_E_FEATURE_TYPE</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>PRO_FEAT_CHAMFER</td>
</tr>
<tr>
<td>PRO_E_STD_EDGE_CHAMF_SCHEME</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>CHM_45_X_D or CHM_D_X_D</td>
</tr>
<tr>
<td>PRO_E_STD_EDGE_CHAMF_DIM</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
<td>d</td>
</tr>
<tr>
<td>PRO_E_STD_EDGE_CHAMF_EDGES</td>
<td>PRO_VALUE_TYPE_SELECTION (multiple)</td>
<td>One or more edges in an array of ProSelection objects</td>
</tr>
<tr>
<td>For dimension scheme d1 x d2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 19-1: Edge Chamfer Element Tree
<table>
<thead>
<tr>
<th>Element ID</th>
<th>Data Type</th>
<th>Values and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_FEATURE_TYPE</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>PRO_FEAT_CHAMFER</td>
</tr>
<tr>
<td>PRO_E_STD_EDGE_CHAMF_SCHEME</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>CHM_D1_X_D2</td>
</tr>
<tr>
<td>PRO_E_STD_EDGE_CHAMF_DIM1</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
<td>d1</td>
</tr>
<tr>
<td>PRO_E_STD_EDGE_CHAMF_DIM2</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
<td>d2</td>
</tr>
<tr>
<td>PRO_E_STD_EDGE_CHAMF_SRF</td>
<td>PRO_VALUE_TYPE_SELECTION</td>
<td>Reference surface</td>
</tr>
<tr>
<td>PRO_E_STD_EDGE_CHAMF_EDGES</td>
<td>PRO_VALUE_TYPE_SELECTION</td>
<td>One or more edges in an array of ProSelection object</td>
</tr>
</tbody>
</table>

For dimension scheme Ang x d

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Data Type</th>
<th>Values and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_FEATURE_TYPE</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>PRO_FEAT_CHAMFER</td>
</tr>
<tr>
<td>PRO_E_STD_EDGE_CHAMF_SCHEME</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>CHM_ANG_X_D</td>
</tr>
<tr>
<td>PRO_E_STD_EDGE_CHAMF_DIM</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
<td>d</td>
</tr>
<tr>
<td>PRO_E_STD_EDGE_CHAMF_ANGLE</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
<td>Angle</td>
</tr>
<tr>
<td>PRO_E_STD_EDGE_CHAMF_SRF</td>
<td>PRO_VALUE_TYPE_SELECTION</td>
<td>Reference surface</td>
</tr>
<tr>
<td>PRO_E_STD_EDGE_CHAMF_EDGES</td>
<td>PRO_VALUE_TYPE_SELECTION</td>
<td>One or more edges in an array of ProSelection objects</td>
</tr>
</tbody>
</table>

The value of the PRO_E_STD_EDGE_CHAMF_EDGES element is an array of one or more ProSelection objects, with each member in the array representing an edge on which to create a chamfer. Note that the PRO_E_STD_EDGE_CHAMF_EDGES element is a single-valued element (not an array element), even though its value is an array of ProSelection objects.
Creating Chamfer Features

Example 1: Creating an Edge Chamfer Feature in Pro/ENGINEER Release 2001

The following example code illustrates how to create a chamfer feature programmatically.

```cpp
#include <ProToolkit.h>
#include <ProObjects.h>
#include <ProChamfer.h>
#include <ProFeature.h>
#include <ProFeatType.h>
#include <ProElement.h>
#include <ProValue.h>
#include <ProMenu.h>
#include <ProUtil.h>
#include <UtilMessage.h>
#include <UtilString.h>
#include <UtilNames.h>
#include <UtilMath.h>

/*====================================================================*
FUNCTION:    UserDemoChamferCreate()
PURPOSE:     Create a chamfer feature.
/*====================================================================*/
static int UserDemoChamferCreate(
    ProAppData   app_data,
    int    feat_type)
{
    int                     menu_id;
    ProError                status;
    ProFeature              feature;
    ProElement              parent_element, *feat_elem;
    UserDemoTreeElemdata    tree_elem_data;
    ProFeatureCreateOptions opts[1];
    ProMdl                  model;
    ProModelitem            mdlitem;
    ProSelection            model_sel;
    ProErrorlist            errs;

    if (feat_type == -1)
        return (-1);
    /********************************************************************************
Create an element tree and add the feature type element.
**********************************************************************************/
    status = ProElementAlloc (PRO_E_FEATURE_TREE,
        &tree_elem_data.tree);
    feat_elem = &tree_elem_data.tree;
    tree_elem_data.parent_element = feat_elem;
    UserDemoFeatTypeElemAdd (&tree_elem_data, PRO_FEAT_CHAMFER);
    return (status);  // return status if creation fails
}
```
tree_elem_data.parent_element = tree_elem_data.tree;
/*--------------------------------------------------------------------*/
Add the remaining elements to the tree.
/*--------------------------------------------------------------------*/
status = UserDemoEdgeChamfer ((ProAppData)&tree_elem_data,
feat_type);
if (status != PRO_TK_NO_ERROR)
return (status);
/*--------------------------------------------------------------------*/
Display the tree in the information window to verify the tree structure.
/*--------------------------------------------------------------------*/
ProUtilShowTreeInInfo (*feat_elem);
/*--------------------------------------------------------------------*/
Create the chamfer.
/*--------------------------------------------------------------------*/
status = ProMdICurrentGet (&model);
status = ProModelitemInit (model, -1, -1, &mdlitem);
status = ProSelectionAlloc (NULL, (ProModelitem *)&mdlitem,
&model_sel);
opts[0] = PRO_FEAT_CR_NO_OPTS;
status = ProFeatureCreate (model_sel, *feat_elem, opts, 1, &feature,
&errs);
return (status);
/*====================================================================*/
FUNCTION:  UserDemoEdgeChamfer()
PURPOSE:   Create an edge chamfer feature.
/*====================================================================*/
static int UserDemoEdgeChamfer (ProAppData app_data,
int       scheme)
{
ProError   status;
UserDemoTreeElemdata  *tree_elem_data;
ProElement  parent_element;
ProSelection  *sel, *edges, surface;
int         n, n_sel;
ProCharLine  str;
double      d1, d2, d, ang = 45.0;
ProValue     *values;
ProValueData value_data;
ProElement   element;

d1 = d2 = d = 0.5;
if (scheme == CHM_D1_X_D2)
{
    ProMessageDisplay (msgfil, "USER %0s",
        "Enter chamfer dimension along the reference
ProMessageDisplay (msgfil, "USER %0s",
"Enter distance from edge along the other
surface [0.5] : ");
status = ProMessageDoubleRead (NULL, &d1);

ProMessageDisplay (msgfil, "USER %0s",
"Enter distance from edge along the other
surface [0.5] : ");
status = ProMessageDoubleRead (NULL, &d2);
}
else
{
    ProMessageDisplay (msgfil, "USER %0s",
"Enter chamfer dimension [0.5] : ");
    status = ProMessageDoubleRead (NULL, &d);
}
if (scheme == CHM_ANG_X_D)
{
    ProMessageDisplay (msgfil, "USER %0s",
"Enter angle from reference surface [45.0] : ");
    status = ProMessageDoubleRead (NULL, &ang);
}
if ((scheme == CHM_D1_X_D2) || (scheme == CHM_ANG_X_D))
{
    ProMessageDisplay (msgfil, "USER %0s", "Select the reference
surface");
    status = ProSelect (*"sldface", 1, NULL, NULL, NULL, NULL,
&sel, &n_sel);
    if (n_sel < 1)
        return (-1);
    status = ProSelectionCopy (sel[0], &surface);
}
ProMessageDisplay (msgfil, "USER %0s", "Select one or more edges to
chamfer");
status = ProSelect (*"sldedge", -1, NULL, NULL, NULL, NULL, 
&edges, 
&n_sel);
if (n_sel < 1)
    return (-1);

tree_elem_data = (UserDemoTreeElemdata *) app_data;
tree_elem_data->elem_id = PRO_E_STD_EDGE_CHAMF_SCHEME;
tree_elem_data->value_data.type = PRO_VALUE_TYPE_INT;
tree_elem_data->value_data.v.i = scheme;
UserDemoFeatElemAdd (tree Elem_data);

tree_elem_data->parent_element = tree_elem_data->tree;
if ((scheme == CHM_45_X_D) || (scheme == CHM_D_X_D))
{
    tree_elem_data->elem_id = PRO_E_STD_EDGE_CHAMF_DIM;
    tree_elem_data->value_data.type = PRO_VALUE_TYPE_DOUBLE;
    tree_elem_data->value_data.v.d = d;
    UserDemoFeatElemAdd (tree_elem_data);
    tree_elem_data->parent_element = tree_elem_data->tree;
}
else if (scheme == CHM_D1_X_D2)
{
    tree_elem_data->elem_id = PRO_E_STD_EDGE_CHAMF_DIM1;
    tree_elem_data->value_data.type = PRO_VALUE_TYPE_DOUBLE;
    tree_elem_data->value_data.v.d = d1;
    UserDemoFeatElemAdd (tree_elem_data);
    tree_elem_data->parent_element = tree_elem_data->tree;

    tree_elem_data->elem_id = PRO_E_STD_EDGE_CHAMF_DIM2;
    tree_elem_data->value_data.type = PRO_VALUE_TYPE_DOUBLE;
    tree_elem_data->value_data.v.d = d2;
    UserDemoFeatElemAdd (tree_elem_data);
    tree_elem_data->parent_element = tree_elem_data->tree;
}
else if (scheme == CHM_ANG_X_D)
{
    tree_elem_data->elem_id = PRO_E_STD_EDGE_CHAMF_DIM;
    tree_elem_data->value_data.type = PRO_VALUE_TYPE_DOUBLE;
    tree_elem_data->value_data.v.d = d;
    UserDemoFeatElemAdd (tree_elem_data);
    tree_elem_data->parent_element = tree_elem_data->tree;

    tree_elem_data->elem_id = PRO_E_STD_EDGE_CHAMF_ANGLE;
    tree_elem_data->value_data.type = PRO_VALUE_TYPE_DOUBLE;
    tree_elem_data->value_data.v.d = ang;
    UserDemoFeatElemAdd (tree_elem_data);
    tree_elem_data->parent_element = tree_elem_data->tree;
}
if ((scheme == CHM_D1_X_D2) || (scheme == CHM_ANG_X_D))
{
    tree_elem_data->elem_id = PRO_E_STD_EDGE_CHAMF_SRF;
    tree_elem_data->value_data.type = PRO_VALUE_TYPE_SELECTION;
    tree_elem_data->value_data.v.r = surface;
    UserDemoFeatElemAdd (tree_elem_data);
    tree_elem_data->parent_element = tree_elem_data->tree;
}

    tree_elem_data->elem_id = PRO_E_STD_EDGE_CHAMF_EDGES;
    tree_elem_data->value_data.type = PRO_VALUE_TYPE_SELECTION;
    UserDemoElemValuesAdd (tree_elem_data, n_sel, edges);
    return (status);
}
FUNCTION: UserDemoFeatElemAdd  
\*-----------------------------------------------*\
static ProError UserDemoFeatElemAdd(  
    UserDemoTreeElemdata *elem)  
{
    ProValue     value;  
    ProElement   element;  
    ProError     status;

    if (elem->value_data.type != -1)  
    {
        status = ProValueAlloc (&value);  
        status = ProValueDataSet (value, &elem->value_data);  
    }
    status = ProElementAlloc (elem->elem_id, &element);  
    if (elem->value_data.type != -1)  
    status = ProElementValueSet (element, value);  
    status = ProElemtreeElementAdd (elem->parent_element, NULL, element);  
    elem->parent_element = element;
    return(status);
}  
\*-----------------------------------------------*\
FUNCTION: UserDemoElemValuesAdd  
\*-----------------------------------------------*\
static ProError UserDemoElemValuesAdd (  
    UserDemoTreeElemdata  *elem,  
    int                    n_sel,  
    ProSelection          *sel)  
{
    ProError               status;  
    ProValue              *values;  
    ProValueData           value_data;  
    ProElement             element;  
    int                    n;

    values = (ProValue *) calloc (n_sel, sizeof(ProValue));  
    for (n = 0; n < n_sel; n++)  
    {
        value_data.type = PRO_VALUE_TYPE_SELECTION;  
        value_data.v.r = sel[n];  
        status = ProValueAlloc (&values[n]);  
        status = ProValueDataSet (values[n], &value_data);  
    }
    status = ProElementAlloc (elem->elem_id, &element);  
    status = ProElementValuesSet (element, values, n_sel);  
    status = ProElemtreeElementAdd (elem->parent_element, NULL, element);  
}

Creating Chamfer Features

19 - 9
free(values);
return (status);
} /*===============================================*/

FUNCTION: UserDemoFeatTypeElemAdd
PURPOSE: Add the feature type element to the tree.
/*===============================================*/
static ProError UserDemoFeatTypeElemAdd (  
    UserDemoTreeElem data *elem,
    int feat_type)
{
    ProError status;
    elem->elem.id = PRO_E_FEATURE_TYPE;
    elem->value_data.type = PRO_VALUE_TYPE_INT;
    elem->value_data.v.i = feat_type;

    status = UserDemoFeatElemAdd (elem);
    return (status);
}
Corner Chamfers

The element tree for corner chamfers is described in the following figure and table.

Figure 19-2: Corner Chamfer Element Tree

```
PRO_E_FEATURE_TREE
  |---PRO_E_FEATURE_TYPE
  |---PRO_E_STD_CORNER_CHAMF_CORNER
  |---PRO_E_STD_CORNER_CHAMF_EDGE1
  |---PRO_E_STD_CORNER_CHAMF_EDGE2
  |---PRO_E_STD_EDGE_CHAMF_EDGE3
```

The following table describes the necessary elements to create corner chamfers.

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Data Type</th>
<th>Values and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_FEATURE_TYPE</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>PRO_FEAT_CORN_CHAMF</td>
</tr>
<tr>
<td>PRO_E_STD_CORNER_CHAMF_CORNER</td>
<td>PRO_VALUE_TYPE_SELECTION</td>
<td>Selection of PRO_EDGE_START or PRO_EDGE_END</td>
</tr>
<tr>
<td>PRO_E_STD_CORNER_CHAMF_EDGE1</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
<td>Offset of first corner edge</td>
</tr>
<tr>
<td>PRO_E_STD_CORNER_CHAMF_EDGE2</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
<td>Offset of second corner edge</td>
</tr>
<tr>
<td>PRO_E_STD_CORNER_CHAMF_EDGE3</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
<td>Offset of third corner edge</td>
</tr>
</tbody>
</table>
Creating Hole Features

This chapter describes programmatic creation of hole features with the Pro/TOOLKIT include file ProHole.h. Chapter ‘Principles of Feature Creation’ provides necessary background for creating holes; we recommend you read that material first.

**Topic** | **Page**
--- | ---
Overview | 20 - 2
The Element Tree for Hole Features | 20 - 2
Feature Element Data Types | 20 - 4
Common Element Values | 20 - 8
PRO_E_HLE_COM Values | 20 - 8
Valid PRO_E_HLE_COM Sub-Elements | 20 - 62
Hole Placement Types | 20 - 67
Miscellaneous Information | 20 - 71

**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

The element tree for the hole feature is documented in the Pro/TOOLKIT header file ProHole.h.

Pro/TOOLKIT supports three types of holes:

- Straight
- Standard
- Sketched

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 these cases.

All hole types and placement types require entry of specific elements during element tree creation. Elements must be entered in the order specified.

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 Creating Hole Features).

Note: All angle elements are specified in degrees.

The Element Tree for Hole Features

The Pro/TOOLKIT header file ProHole.h contains the element tree for hole features. ProHole.h describes:

- The basic elements representing feature type and feature form
- Common elements defining hole types
- Common elements defining hole placement

The following figures shows the tree structure of the element trees.
Figure 20-1: Common Elements for Hole Types

```
PRO_E_HOLE_COM
  | PRO_E_HOLE_TYPE_NEW
  | PRO_E_HOLE_STANDARD
  | PRO_E_HOLE_THROATED
  | PRO_E_HOLE_TAPER
  | PRO_E_HOLE_NUT
  | PRO_E_HOLE_SCREW
  | PRO_E_HOLE_ADD_THREADED
  | PRO_E_HOLE_ADD_CBORE
  | PRO_E_HOLE_ADD_CSINK

  | PRO_E_DIAMETER
  | PRO_E_HOLE_STD_DEPTH
    | PRO_E_HOLE_DEPTH_TO
      | PRO_E_HOLE_DEPTH_TO_TYPE
      | PRO_E_HOLE_DEPTH_TO_VALUE
      | PRO_E_HOLE_DEPTH_TO_REF
    | PRO_E_HOLE_DEPTH_FROM
      | PRO_E_HOLE_DEPTH_FROM_TYPE
      | PRO_E_HOLE_DEPTH_FROM_VALUE
      | PRO_E_HOLE_DEPTH_FROM_REF

PRO_E_HOLE_HOLEDIAM
PRO_E_HOLE_DRILLADIUS
PRO_E_HOLE_CSINKANGLE
PRO_E_HOLE_CBOREDTH
PRO_E_HOLE_CBOREDIAM
PRO_E_HOLE_CSINKDIAM
PRO_E_HOLE_THROATED
PRO_E_HOLE_DILLDEPTH
PRO_E_HOLE_THRO_DEPTH
PRO_E_HOLE_DEPTH
PRO_E_SKECHER
PRO_E_HOLE_CBORE_DIR
PRO_E_HOLE_ADD_EXIT_CSINK
PRO_E_HOLE_EXIT_CSINKANGLE
PRO_E_HOLE_EXIT_CSINKDIAM
```
Feature Element Data Types

The following table lists data types for hole type and placement elements.

Element values must be of the specified type.

<table>
<thead>
<tr>
<th>Element Id</th>
<th>Element Name</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_FEATURE_TYPE</td>
<td>Feature Type</td>
<td>PRO_VALUE_TYPE_INT</td>
</tr>
<tr>
<td>PRO_E_FEATURE_FORM</td>
<td>Feature Form</td>
<td>PRO_VALUE_TYPE_INT</td>
</tr>
<tr>
<td>PRO_E_STD_FEATURE_NAME</td>
<td>Feature Name</td>
<td>PRO_VALUE_TYPE_WSTRING</td>
</tr>
<tr>
<td>PRO_E_HLE_COM</td>
<td>Hole</td>
<td>Compound</td>
</tr>
<tr>
<td>PRO_E_HLE_TYPE_NEW</td>
<td>Hole Type</td>
<td>PRO_VALUE_TYPE_INT</td>
</tr>
<tr>
<td>PRO_E_HLE_STAN_TYPE</td>
<td>Standard Type</td>
<td>PRO_VALUE_TYPE_INT</td>
</tr>
<tr>
<td>PRO_E_HLE_THIRD_SERIS</td>
<td>Thread Series</td>
<td>PRO_VALUE_TYPE_INT</td>
</tr>
<tr>
<td>Element Id</td>
<td>Element Name</td>
<td>Data Type</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>PRO_E_HLE_FITTYPE</td>
<td>Fit Type</td>
<td>PRO_VALUE_TYPE_INT</td>
</tr>
<tr>
<td>PRO_E_HLE_SCREWSIZE</td>
<td>Screw Size</td>
<td>PRO_VALUE_TYPE_INT</td>
</tr>
<tr>
<td>PRO_E_HLE_DEPTH</td>
<td>Depth</td>
<td>PRO_VALUE_TYPE_INT</td>
</tr>
<tr>
<td>PRO_E_HLE_HOLEDIAM</td>
<td>Diameter</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
</tr>
<tr>
<td>PRO_E_HLE_DRILLANGLE</td>
<td>Drillhead Angle</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
</tr>
<tr>
<td>PRO_E_HLE_ADD_THREAD</td>
<td>Add Thread</td>
<td>PRO_VALUE_TYPE_INT</td>
</tr>
<tr>
<td>PRO_E_HLE_THRD DEPTH</td>
<td>Thread Depth</td>
<td>PRO_VALUE_TYPE_INT</td>
</tr>
<tr>
<td>PRO_E_HLE_THRDDEPTH</td>
<td>Thread Depth</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
</tr>
<tr>
<td>PRO_E_HLE_ADD_CBORE</td>
<td>Add Counterbore</td>
<td>PRO_VALUE_TYPE_INT</td>
</tr>
<tr>
<td>PRO_E_HLE_CBOREDEPTH</td>
<td>Counterbore Depth</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
</tr>
<tr>
<td>PRO_E_HLE_CBOREDIAM</td>
<td>Counterbore Diameter</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
</tr>
<tr>
<td>PRO_E_HLE_ADD_CSINK</td>
<td>Add Countersink</td>
<td>PRO_VALUE_TYPE_INT</td>
</tr>
<tr>
<td>PRO_E_HLE_CSINKANGLE</td>
<td>Csink Angle</td>
<td>PRO_VALUE_TYPE_INT</td>
</tr>
<tr>
<td>PRO_E_HLE_CSINKDIAM</td>
<td>Csink Diameter</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
</tr>
<tr>
<td>PRO_E_HLE_DRILLDEPTH</td>
<td>Drill Depth</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
</tr>
<tr>
<td>PRO_E_HLE_ADD_EXIT_CSINK</td>
<td>Add Exit Csink</td>
<td>PRO_VALUE_TYPE_INT</td>
</tr>
<tr>
<td>PRO_E_HLE_EXIT_CSINKANGLE</td>
<td>Exit Csink Angle</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
</tr>
<tr>
<td>PRO_E_HLE_EXIT_CSINKDIAM</td>
<td>Exit Csink Diameter</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
</tr>
<tr>
<td>PRO_E_DIAMETER</td>
<td>Diameter</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
</tr>
<tr>
<td>PRO_E_HOLE_STD_DEPTH</td>
<td>Depth Element</td>
<td>Compound</td>
</tr>
<tr>
<td>PRO_E_HOLE_DEPTH_TO</td>
<td>Depth Two</td>
<td>Compound</td>
</tr>
<tr>
<td>PRO_E_HOLE_DEPTH_TO_TYPE</td>
<td>Depth Two</td>
<td>PRO_VALUE_TYPE_INT</td>
</tr>
<tr>
<td>PRO_E_EXTDEPTH_TO_VALUE</td>
<td>Depth Value</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
</tr>
<tr>
<td>Element Id</td>
<td>Element Name</td>
<td>Data Type</td>
</tr>
<tr>
<td>------------</td>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>PRO_E_EXT_DEPTH_TO_REF</td>
<td>Reference</td>
<td>PRO_VALUE_TYPE_SELECTION</td>
</tr>
<tr>
<td>PRO_E_HOLE_STD_DEPTH</td>
<td>Depth</td>
<td>Compound</td>
</tr>
<tr>
<td>PRO_E_HOLE_DEPTH_FROM</td>
<td>Depth One</td>
<td>Compound</td>
</tr>
<tr>
<td>PRO_E_HOLE_DEPTH_FROM_TYPE</td>
<td>Depth One</td>
<td>PRO_VALUE_TYPE_INT</td>
</tr>
<tr>
<td>PRO_E_EXT_DEPTH_FROM_VALUE</td>
<td>Depth Value</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
</tr>
<tr>
<td>PRO_E_EXT_DEPTH_FROM_REF</td>
<td>Reference</td>
<td>PRO_VALUE&gt;Type_SELECTION</td>
</tr>
<tr>
<td>PRO_E_HLE_SKETCHER</td>
<td>Sketcher</td>
<td>N/A</td>
</tr>
<tr>
<td>PRO_E_HLE_CRDIR_FLIP</td>
<td>Creation Direction</td>
<td>PRO_VALUE_TYPE_INT</td>
</tr>
<tr>
<td>PRO_E_HLE_PLACEMENT</td>
<td>Placement</td>
<td>N/A</td>
</tr>
<tr>
<td>PRO_E_HLE_PRIM_REF</td>
<td>Primary Reference</td>
<td>PRO_VALUE_TYPE_SELECTION</td>
</tr>
<tr>
<td>PRO_E_HLE_PL_TYPE</td>
<td>Placement Options</td>
<td>PRO_VALUE_TYPE_INT</td>
</tr>
<tr>
<td>PRO_E_HLE_DIM_REF1</td>
<td>DimensionRef 1</td>
<td>PRO_VALUE_TYPE_SELECTION</td>
</tr>
<tr>
<td>PRO_E_HLE_PLC_ALIGN_OPT1</td>
<td>Alignment for placement reference 1</td>
<td>PRO_VALUE_TYPE_INT</td>
</tr>
<tr>
<td>PRO_E_HLE_DIM_DIST1</td>
<td>Distance 1</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
</tr>
<tr>
<td>PRO_E_HLE_PLC_ALIGN_OPT2</td>
<td>Alignment for placement reference 2</td>
<td>PRO_VALUE_TYPE_INT</td>
</tr>
<tr>
<td>PRO_E_HLE_DIM_REF2</td>
<td>DimensionRef 2</td>
<td>PRO_VALUE_TYPE_SELECTION</td>
</tr>
<tr>
<td>PRO_E_HLE_DIM_DIST2</td>
<td>Distance 2</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
</tr>
<tr>
<td>PRO_E_HLE_AXIS</td>
<td>Axis</td>
<td>PRO_VALUE_TYPE_SELECTION</td>
</tr>
<tr>
<td>PRO_E_HLE_REF_PLANE</td>
<td>Reference Plane</td>
<td>PRO_VALUE_TYPE_SELECTION</td>
</tr>
<tr>
<td>PRO_E_HLE_REF_ANG</td>
<td>Angle</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
</tr>
<tr>
<td>PRO_E_HLE_DIM_TYPE</td>
<td>Dimension Type</td>
<td>PRO_VALUE_TYPE_INT</td>
</tr>
<tr>
<td>Element Id</td>
<td>Element Name</td>
<td>Data Type</td>
</tr>
<tr>
<td>---------------------</td>
<td>------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>PRO_E_HLE_DIM_DIA</td>
<td>Diameter</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
</tr>
<tr>
<td>PRO_E_HLE_DIM_RAD</td>
<td>Radius</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
</tr>
<tr>
<td>PRO_E_HLE_DIM_LIN</td>
<td>Linear Distance</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
</tr>
<tr>
<td>PRO_E_HLE_NORM_PlA</td>
<td>Normal Plane</td>
<td>PRO_VALUE_TYPE_SELECTION</td>
</tr>
<tr>
<td>PRO_E_HLE_NORM_OFFSET</td>
<td>Offset</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
</tr>
<tr>
<td>PRO_E_HLE_PLCMNT_PLANE</td>
<td>Placement Plane</td>
<td>PRO_VALUE_TYPE_SELECTION</td>
</tr>
<tr>
<td>PRO_E_HLE_REF_PLANE_1</td>
<td>Reference Plane</td>
<td>PRO_VALUE_TYPE_SELECTION</td>
</tr>
<tr>
<td>PRO_E_HLE_REF_ANG_1</td>
<td>Angle</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
</tr>
<tr>
<td>PRO_E_INT_PARTS</td>
<td>Intsect Parts</td>
<td>N/A</td>
</tr>
<tr>
<td>PRO_E_PATTERN</td>
<td>Pattern</td>
<td>N/A</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_FEATURE_TYPE</td>
<td>PRO_FEAT_HOLE</td>
</tr>
<tr>
<td>PRO_E_FEATURE_FORM</td>
<td>PRO_HLE_TYPE_STRAIGHT (for straight holes)</td>
</tr>
<tr>
<td></td>
<td>PRO_HLE_TYPE_SKETCHED (for other hole types)</td>
</tr>
<tr>
<td>PRO_E_STD_FEATURE_NAME</td>
<td>Wstring (feature name)</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Element</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_HLE_TYPE_NEW</td>
<td>PRO_HLE_NEW_TYPE_STRAIGHT</td>
</tr>
<tr>
<td>PRO_E_DIAMETER</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_HOLE_STD_DEPTH</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_HOLE_DEPTH_TO</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_HOLE_DEPTH_TO_TYPE</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_EXT_DEPTH_TO_VALUE</td>
<td>Depends on PRO_E_HOLE_DEPTH_TO_TYPE</td>
</tr>
<tr>
<td>PRO_E_EXT_DEPTH_TO_REF</td>
<td>Depends on PRO_E_HOLE_DEPTH_TO_TYPE</td>
</tr>
<tr>
<td>PRO_E_HOLE_DEPTH_FROM</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_HOLE_DEPTH_FROM_TYPE</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_EXT_DEPTH_FROM_VALUE</td>
<td>Depends on PRO_E_HOLE_DEPTH_FROM_TYPE</td>
</tr>
<tr>
<td>PRO_E_EXT_DEPTH_FROM_REF</td>
<td>Depends on PRO_E_HOLE_DEPTH_FROM_TYPE</td>
</tr>
</tbody>
</table>
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

```c
/*---------------------- 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 function 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 : ProDemoHoleCreate
```

Creating Hole Features
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 */
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_STRAIGHT ;

status = UserElemtreeElementAdd ( elem_hle_com,
                                elem_hle_type_new, value_data );
/* Specifying the Diameter of the Hole */

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 */

status = ProElementAlloc ( PRO_E_HOLE_STD_DEPTH, &elem_hole_std_depth );
UG_HOLE_LOG ( " ProElementAlloc PRO_E_HOLE_STD_DEPTH ", status );

status = ProElemtreeElementAdd ( elem_hole_std_depth, NULL,
                                 elem_hole_std_depth );
UG_HOLE_LOG ( " ProElemtreeElementAdd ", status );

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 );
printf(" PRO_E_HOLE_DEPTH_TO_TYPE \n");
printf(" ****************** \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);

printf(" PRO_E_HOLE_DEPTH_FROM \n");
printf(" ****************** \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);

printf(" PRO_E_HOLE_DEPTH_FROM_TYPE \n");
printf(" ****************** \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 to the placement details

|--PRO_E_HLE_PLACEMENT
  |  |--PRO_E_HLE_PRIM_REF
  |  |--PRO_E_HLE_PL_TYPE
  |  |--PRO_E_HLE_DIM_REF1

Creating Hole Features
|   |--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(" ****************** 
");
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);

printf(" PRO_E_HLE_DIM_REF2 \n");
printf(" ****************** 
");
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(" ****************** 
");
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 'Creating Sketched Features', the second uses the function ProFeatureCreate() more directly.

The following table describes the required elements for sketched hole features.

<table>
<thead>
<tr>
<th>Sketched Hole Elements</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_HLE_TYPE_NEW</td>
<td>PRO_HLE_NEW_TYPE_SKETCH</td>
</tr>
<tr>
<td>PRO_E_SKETCHER</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_HLE_CRDIR_FLIP</td>
<td>Mandatory</td>
</tr>
</tbody>
</table>

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.

```c
#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(" ****************** ");

status = ProElementAlloc ( PRO_E_HOLE_COM, &elem_hle_com );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HOLE_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(" ****************** ");

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(" ****************** ");

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

backup

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);

status = ProElementAlloc ( PRO_E_HLE_PRIM_REF, &elem_hle_prim_ref );
UG_HOLE_LOG ("ProElementAlloc PRO_E_HLE_PRIM_REF ", status);

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);

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);

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);

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);
status = ProElementFree( &feat_elemtree );
UG_HOLE_LOG (" ProElementFree ", status);
status = ProSelectionFree( &model_selection );
UG_HOLE_LOG (" ProSelectionFree ", 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

return ( status );
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()
="/""="="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""="="/""=" JFactory: 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()
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);

/*---------------------------------------------------------------------*\nAdding 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 );
}

/*===============================================================*\nFUNCTION : 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);
Creating Hole Features

```c
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

```c
#include "ProToolkit.h"
#include "ProFeature.h"
#include "ProElemId.h"
#include "ProStdSection.h"
#include "ProElement.h"
```
```c
#include "ProFeatType.h"
#include "ProFeatForm.h"
#include "ProSelection.h"
#include "ProHole.h"

/*---------------------------------------------------------------------*\nMACRO
*---------------------------------------------------------------------*/
/*---------------------------------------------------------------------*\nUG_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 );
/*---------------------------------------------------------------------*\nExternal Data
*---------------------------------------------------------------------*/
/*----------------------------- Global Data -----------------------------*/
/*===============================================================*\nFUNCTION : ProDemoSketchedModiHoleCreate
PURPOSE  : Demonstrates the creation of a Standard Threaded hole
            with linear placement
*==================================================================*/
ProError ProDemoSketchedModiHoleCreate()
{
ProError status;

ProElement feat_elemtree;
    ProElement elem_featype;
    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_elmtree );
/*------------------------------------------*/
Adding the element for feature type
/*------------------------------------------*/

printf(" PRO_E_FEATURE_TREE \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_featform, value_data );
UG_HOLE_LOG (" ProElementAlloc ", status );
/*---------------------------------------------------------------------*/

Adding the element for feature form
/*---------------------------------------------------------------------*/

printf(" PRO_E_FEATURE_FORM \
");
printf(" ****************** \
");
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 \
");
printf(" ****************** \
");
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 \
");
printf(" ****************** \
");
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_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(" **************** 
");

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  
");
printf(" **************** 
");

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 = ProElementAdd ( feat Elemtree, NULL,
elem_hle_placement );
UG_HOLE_LOG (" ProElementAdd ", status);

status = ProElementAlloc ( PRO_E_HLE_PRIM_REF,
&elem_hle_prim_ref );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HLE_PRIM_REF ", 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);

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);

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 Standard Threaded Hole for a code example on creating this type of hole.

<table>
<thead>
<tr>
<th>Standard Hole Elements</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_HLE_TYPE_NEW</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_HLE_STAN_TYPE</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_HLE_THRDSERIS</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_HLE_FITTYPE</td>
<td>Mandatory: set to PRO_HLE_CLOSE_FIT</td>
</tr>
<tr>
<td>PRO_E_HLE_SCREWSIZE</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_HLE_DEPTH</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_HLE_HOLEDIAM</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_HLE_DRILLANGLE</td>
<td>Required for variable depth hole</td>
</tr>
</tbody>
</table>
### Standard Hole Elements

<table>
<thead>
<tr>
<th>Element</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_HLE_DRILLDEPTH</td>
<td>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.</td>
</tr>
<tr>
<td>PRO_E_HLE_ADD_THREAD</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_HLE_THRD_DEPTH</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_HLE_THRDDEPTH</td>
<td>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.</td>
</tr>
<tr>
<td>PRO_E_HLE_ADD_CBORE</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_HLE_CBOREDEPTH</td>
<td>Required for counterbore option</td>
</tr>
<tr>
<td>PRO_E_HLE_CBOREDIAM</td>
<td>Required for counterbore option</td>
</tr>
<tr>
<td>PRO_E_HLE_ADD_CSINK</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_HLE_CSINKANGLE</td>
<td>Required for countersink option</td>
</tr>
<tr>
<td>PRO_E_HLE_CSINKDIAM</td>
<td>Required for countersink option</td>
</tr>
<tr>
<td>PRO_E_HLE_ADD_EXIT_CSINK</td>
<td>Required for Thru all hole</td>
</tr>
<tr>
<td>PRO_E_HLE_EXIT_CSINKANGLE</td>
<td>Required for exit countersink option</td>
</tr>
<tr>
<td>PRO_E_HLE_EXIT_CSINKDIAM</td>
<td>Required for exit countersink option</td>
</tr>
</tbody>
</table>

### Standard Threaded Hole

**Example 2: Creating a Standard Threaded Hole with Linear Placement**

```cpp
#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_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_screwsise;
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

Start of Element Tree Creation

Start of Element Tree Creation

Pro/TOOLKIT User's Guide
Creating Hole Features

/*-------------------------------*/
status = ProElementAlloc ( PRO_E_FEATURE_TREE, &feat_elemtree );
/*-------------------------------*/
Adding the root element

status = ProElementAlloc ( PRO_E_FEATURE_TREE, &feat_elemtree );

/*-------------------------------*/
Adding the element for feature type

printf(" PRO_E_FEATURE_TYPE \
");
printf(" ****************** \
");
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

status = ProElementAlloc ( PRO_E_FEATURE_FORM , &elem_featform );

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

status = ProElementAlloc ( PRO_E_HLE_COM, &elem_hle_com );

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

```c
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

```c
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 */
/* 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_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 
");
printf(" ****************** 
");

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 
");
printf(" ****************** 
");

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 
");
printf(" ****************** 
");

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 
");
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);
Creating Hole Features

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 
");
printf(" ****************** 
");

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,
                    &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 
");
printf(" ****************** 
");

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 
");
printf(" ****************** 
");

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 
");
printf(" ****************** 
");

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 
");
printf(" ****************** 
");

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 
");
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 
");
printf(" ****************** 
");

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 );
UGE_HOLE_LOG (" UserElemtreeElementAdd ", status);

/*---------------------------------------------------------------------*\
 | End of Element Tree Creation                                    |
\*---------------------------------------------------------------------*/

/*---------------------------------------------------------------------*\
 | Start of Feature Creation                                       |
\*---------------------------------------------------------------------*/

status = ProMdlCurrentGet ( &model );
UGE_HOLE_LOG (" ProMdlCurrentGet ", status);

status = ProMdlToModelitem( model, &model_item );
UGE_HOLE_LOG (" ProMdlToModelitem ", status);

status = ProSelectionAlloc( NULL, &model_item, &model_selection );
UGE_HOLE_LOG (" ProSelectionAlloc ", status);

status = ProFeatureCreate ( model_selection, feat_elemtree,
               options, 1,
               &created_feature, &p_errors );
UGE_HOLE_LOG (" ProFeatureCreate ", status);

/*---------------------------------------------------------------------*\
 | End of Feature Creation                                          |
\*---------------------------------------------------------------------*/

/*---------------------------------------------------------------------*\n | Freeing the resources                                             |
\*---------------------------------------------------------------------*/

status = ProElementFree( &feat_elemtree );
UGE_HOLE_LOG (" ProElementFree ", status);

status = ProSelectionFree( &model_selection );
UGE_HOLE_LOG (" ProSelectionFree ", status);

/*---------------------------------------------------------------------*\n | End of Freeing                                                    |
\*---------------------------------------------------------------------*/

return ( status );

/*---------------------------------------------------------------------*\n FUNCTION : UserElemtreeElementAdd                               |
 PURPOSE : Adding an element to the parent element / tree         |
 input : parent_element, child_element, value_data               |
 output : none                                                   |

20 - 50
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 Standard Clearance Hole for a code example on creating this type of hole.

**Standard Clearance Hole Elements**

<table>
<thead>
<tr>
<th>Element</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_HLE_TYPE_NEW</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_HLE_STAN_TYPE</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_HLE_THRDSERIS</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_HLE_FITTYPE</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_HLE_SCREWSIZE</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_HLE_DEPTH</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_HLE_HOLEDIAM</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_HLE_DRILLANGLE</td>
<td>Required for variable depth hole</td>
</tr>
<tr>
<td>PRO_E_HLE_ADD_THREAD</td>
<td>Mandatory</td>
</tr>
</tbody>
</table>
Standard Clearance Hole Elements

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_HLE_ADD_CBORE</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_HLE_CBOREDETH</td>
<td>Required for counterbore option</td>
</tr>
<tr>
<td>PRO_E_HLE_CBOREDIAM</td>
<td>Required for counterbore option</td>
</tr>
<tr>
<td>PRO_E_HLE_ADD_CSINK</td>
<td>Mandatory</td>
</tr>
<tr>
<td>PRO_E_HLE_CSINKANGLE</td>
<td>Required for countersink option</td>
</tr>
<tr>
<td>PRO_E_HLE_CSINKDIAM</td>
<td>Required for countersink option</td>
</tr>
<tr>
<td>PRO_E_HLE_ADD_EXIT_CSINK</td>
<td>Required for Thru all hole</td>
</tr>
<tr>
<td>PRO_E_HLE_EXIT_CSINKANGLE</td>
<td>Required for exit countersink option</td>
</tr>
<tr>
<td>PRO_E_HLE_EXIT_CSINKDIAM</td>
<td>Required for exit countersink option</td>
</tr>
</tbody>
</table>

Standard Clearance Hole

**Example 3: Creating a Standard Clearance Hole with Linear Placement**

```c
#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,
```
Creating Hole Features

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_fittype;
  ProElement elem_hle_screwszie;
  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                                       |
---------------------------------------------------------------------*/

printf(" PRO_E_FEATURE_TREE 
");
UG_HOLE_LOG (" ProElementAlloc ", status);
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_FEAT_HOLE ;

/*---------------------------------------------------------------------*
| Adding the root element                                             |
---------------------------------------------------------------------*/

status = ProElementAlloc ( PRO_E_FEATURE_TREE, &feat_eletree );

 /*---------------------------------------------------------------------*
 | Adding the element for feature type                                |
 /*---------------------------------------------------------------------*/

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_element, elem_featuretype, value_data );
UG_HOLE_LOG (" ProElementAlloc ", status );

/*---------------------------------------------------------------*
   Adding the element for feature form
---------------------------------------------------------------*/
status = ProElementAlloc ( PRO_E_FEATURE_FORM , &elem_featureform );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_HOLE_TYPE_STRAIGHT ;
status = UserElemtreeElementAdd( feat_element, elem_featureform, value_data );
UG_HOLE_LOG (" ProElementAlloc ", status );

/*---------------------------------------------------------------*
   Adding the common element for hole information
---------------------------------------------------------------*/
status = ProElementAlloc ( PRO_E_HOLE_COM, &elem_hole_com );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HOLE_COM ", status);
status = ProElemtreeElementAdd ( feat_element, NULL, elem_hole_com );
UG_HOLE_LOG (" ProElemtreeElementAdd ", status);

/*---------------------------------------------------------------*
   Adding the element for hole type: Straight, Standard, Sketched
---------------------------------------------------------------*/
status = ProElementAlloc ( PRO_E_HOLE_TYPE_NEW, &elem_hole_type_new );
UG_HOLE_LOG (" ProElementAlloc PRO_E_HOLE_TYPE_NEW ", status);
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_HOLE_TYPE_STANDARD ;
status = UserElemtreeElementAdd ( elem_hole_com, elem_hole_type_new, value_data );
UG_HOLE_LOG (" UserElemtreeElementAdd ", status);
Elements for Standard Clearance Hole
/*---------------------------------------------------------------*/

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);

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);

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);

status = ProElementAlloc ( PRO_E_HLE_SCREWSIZE, &elem_hle_screwsize );
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_screwsize, 
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);

status = ProSelect ( "datum,surface", 1, 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, 
   &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

20 - 60 ProTOOLKIT User's Guide
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_eletree,
options, 1,
&created_feature, &p_errors );
UG_HOLE_LOG (" ProFeatureCreate ", status);

/*-----------------------------------------------*/
End of Feature Creation
/*-----------------------------------------------*/

status = ProElementFree( &feat_eletree );
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,
Valid PRO_E_HLE_COM Sub-Elements

The following table gives the description of all the elements for all the hole types.

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Comment/Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_HOLE_TYPE_NEW = PRO_HLE_NEW_TYPE_STRAIGHT</td>
<td>Stores the diameter double value</td>
</tr>
<tr>
<td>PRO_E_DIAMETER</td>
<td>Depth (compound element)</td>
</tr>
<tr>
<td>PRO_E_HOLE_STD_DEPTH</td>
<td>First Side depth info (compound element)</td>
</tr>
<tr>
<td>PRO_E_HOLE_DEPTH_TO</td>
<td></td>
</tr>
</tbody>
</table>
## Hole Elements Description

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_HOLE_DEPTH_TO_TYPE</td>
<td>Type ProHleStraightDepType</td>
</tr>
<tr>
<td></td>
<td>/* Blind*/</td>
</tr>
<tr>
<td></td>
<td>PRO_HLE_STRGHT_BLIND_DEPTH</td>
</tr>
<tr>
<td></td>
<td>/* Thru Next*/</td>
</tr>
<tr>
<td></td>
<td>PRO_HLE_STRGHT_THRU_NEXT_DEPTH</td>
</tr>
<tr>
<td></td>
<td>/* Thru All*/</td>
</tr>
<tr>
<td></td>
<td>PRO_HLE_STRGHT_THRU_ALL_DEPTH</td>
</tr>
<tr>
<td></td>
<td>/* Thru Until */</td>
</tr>
<tr>
<td></td>
<td>PRO_HLE_STRGHT_THRU_UNTILDEPTH</td>
</tr>
<tr>
<td></td>
<td>/* Upto Ref */</td>
</tr>
<tr>
<td></td>
<td>PRO_HLE_STRGHT_UPTO_REF_DEPTH</td>
</tr>
<tr>
<td></td>
<td>/* None */</td>
</tr>
<tr>
<td></td>
<td>PRO_HLE_STRGHT_NONE_DEPTH</td>
</tr>
<tr>
<td></td>
<td>/* Symmetric */</td>
</tr>
<tr>
<td></td>
<td>PRO_HLE_STRGHT_SYM_DEPTH</td>
</tr>
<tr>
<td>PRO_E_EXT_DEPTH_TO_VALUE</td>
<td>Stores variable depth double value when PRO_E_HOLE_DEPTH_TO_TYPE equals</td>
</tr>
<tr>
<td></td>
<td>PRO_HLE_STRGHT_BLIND_DEPTH</td>
</tr>
<tr>
<td>PRO_E_EXT_DEPTH_TO_REF</td>
<td>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.</td>
</tr>
<tr>
<td>PRO_E_HOLE_DEPTH_FROM</td>
<td>Second Side depth info (compound element)</td>
</tr>
<tr>
<td>PRO_E_HOLE_DEPTH_FROM_TYPE</td>
<td>Type ProHleStraightDep</td>
</tr>
<tr>
<td></td>
<td>/* Blind*/</td>
</tr>
<tr>
<td></td>
<td>PRO_HLE_STRGHT_BLIND_DEPTH</td>
</tr>
<tr>
<td></td>
<td>/* Thru Next*/</td>
</tr>
<tr>
<td></td>
<td>PRO_HLE_STRGHT_THRU_NEXT_DEPTH</td>
</tr>
<tr>
<td></td>
<td>/* Thru All*/</td>
</tr>
<tr>
<td></td>
<td>PRO_HLE_STRGHT_THRU_ALL_DEPTH</td>
</tr>
<tr>
<td></td>
<td>/* Thru Until */</td>
</tr>
<tr>
<td></td>
<td>PRO_HLE_STRGHT_THRU_UNTILDEPTH</td>
</tr>
<tr>
<td></td>
<td>/* Upto Ref */</td>
</tr>
<tr>
<td></td>
<td>PRO_HLE_STRGHT_UPTO_REF_DEPTH</td>
</tr>
<tr>
<td></td>
<td>/* None */</td>
</tr>
<tr>
<td></td>
<td>PRO_HLE_STRGHT_NONE_DEPTH</td>
</tr>
<tr>
<td></td>
<td>/* Symmetric */</td>
</tr>
<tr>
<td></td>
<td>PRO_HLE_STRGHT_SYM_DEPTH</td>
</tr>
<tr>
<td>Hole Elements Description</td>
<td></td>
</tr>
<tr>
<td>---------------------------</td>
<td></td>
</tr>
<tr>
<td><strong>PRO_E_EXT_DEPTH_FROM_VALUE</strong></td>
<td>Stores variable depth double value when <strong>PRO_E_HOLE_DEPTH_FROM_TYPE</strong> equals <strong>PRO_HLE_STRGHT_BLIND_DEPTH</strong></td>
</tr>
<tr>
<td><strong>PRO_E_EXT_DEPTH_FROM_REF</strong></td>
<td>Stores the upto reference when <strong>PRO_E_HOLE_DEPTH_FROM_TYPE</strong> is not <strong>PRO_HLE_STRGHT_BLIND_DEPTH</strong> and is not <strong>PRO_HLE_STRGHT_NONEDEPTH</strong> and not <strong>PRO_HLE_STRGHT_SYM_DEPTH</strong></td>
</tr>
</tbody>
</table>

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

**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**
### Hole Elements Description

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description and Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_HLE_SCREWSIZE</td>
<td>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.</td>
</tr>
<tr>
<td>PRO_E_HLEDEPTH</td>
<td>It is an option for different type drill depth, that is, of type ProHleStdDepType. <strong>Note:</strong> PRO_HLE_STD_VAR_DEPTH is not available for clearance hole (not for PRO_E_HLE_STAN_TYPE == PRO_HLE_CLEARANCE_TYPE).</td>
</tr>
<tr>
<td>PRO_E_HLE_HOLEDIAM</td>
<td>Stores Drill Diameter double value. See Hole Diameter.</td>
</tr>
<tr>
<td>PRO_E_HLE_DRILLANGLE</td>
<td>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).</td>
</tr>
<tr>
<td>PRO_E_HLE_ADD_THREAD</td>
<td>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.</td>
</tr>
<tr>
<td>PRO_E_HLE_THRD_DEPTH</td>
<td>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. <strong>Note:</strong> For part level, variable drill depth, and assembly level tapped threaded hole, the PRO_HLE_THRU_THREAD value for this element is invalid.</td>
</tr>
</tbody>
</table>
## Hole Elements Description

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_HLE_THRDEPTH</td>
<td>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.</td>
</tr>
<tr>
<td>PRO_E_HLE_ADD_CBORE</td>
<td>Option for Counter Bore. Type ProHleAddCboreFlag. For counter bore it’s value is PRO_HLE_ADD_CBORE. For the no counterbore option, set to PRO_HLE_NO_CBORE.</td>
</tr>
<tr>
<td>PRO_E_HLE_CBOREDEPTH</td>
<td>Stores counterbore depth. Double value. Available for counterbore option, when PRO_E_HLE_ADD_CBORE equals PRO_HLE_ADD_CBORE.</td>
</tr>
<tr>
<td>PRO_E_HLE_CBOREDIAM</td>
<td>Stores counterbore diameter. Double value. Available for counterbore option, when PRO_E_HLE_ADD_CBORE is PRO_HLE_ADD_CBORE.</td>
</tr>
<tr>
<td>PRO_E_HLE_ADD_CSINK</td>
<td>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.</td>
</tr>
<tr>
<td>PRO_E_HLE_CSINKANGLE</td>
<td>Stores counter sink angle. Double value. Available for countersink option, when PRO_E_HLE_ADD_CSINK equals PRO_HLE_ADD_CSINK.</td>
</tr>
<tr>
<td>PRO_E_HLE_CSINKDIAM</td>
<td>Stores countersink diameter. Double value. Available for countersink option, when PRO_E_HLE_ADD_CSINK PRO_HLE_ADD_CSINK.</td>
</tr>
<tr>
<td>PRO_E_HLE_DRILLDEPTH</td>
<td>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.</td>
</tr>
</tbody>
</table>
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

Hole Elements Description

<table>
<thead>
<tr>
<th>PRO_E_HLE_ADD_EXIT_CSINK</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_HLE_EXIT_CSINKANGLE</td>
<td>Stores exit countersink angle double value. Available for exit countersink option, that is, PRO_E_HLE_ADD_EXIT_CSINK == PRO_HLE_ADD_EXIT_CSINK.</td>
</tr>
<tr>
<td>PRO_E_HLE_EXIT_CSINKDIAM</td>
<td>Stores exit countersink diameter double value. Available for exit countersink option, that is, PRO_E_HLE_ADD_EXIT_CSINK == PRO_HLE_ADD_EXIT_CSINK.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_HLE_PRIM_REF</td>
<td>Primary selection, that is, planar surface or datum plane.</td>
</tr>
<tr>
<td>PRO_E_HLE_PL_TYPE</td>
<td>Set to PRO_HLE_PL_TYPE_LIN</td>
</tr>
<tr>
<td>PRO_E_HLE_DIM_REF1</td>
<td>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.</td>
</tr>
<tr>
<td>PRO_E_HLE_PLC_ALIGN_OPT1</td>
<td>- 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.</td>
</tr>
<tr>
<td>PRO_E_HLE_DIM_DIST1</td>
<td>Distance with regard to PRO_E_HLE_DIM_REF1.</td>
</tr>
<tr>
<td>PRO_E_HLE_DIM_REF2</td>
<td>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.</td>
</tr>
<tr>
<td>PRO_E_HLE_PLC_ALIGN_OPT2</td>
<td>- 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.</td>
</tr>
<tr>
<td>PRO_E_HLE_DIM_DIST2</td>
<td>Distance with regard to PRO_E_HLE_DIM_REF2.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_HLE_PRIM_REF</td>
<td>Primary Selection, planar surface/datum plane</td>
</tr>
<tr>
<td>PRO_E_HLE_PL_TYPE</td>
<td>set to PRO_HLE_PL_TYPE_RAD</td>
</tr>
<tr>
<td>PRO_E_HLE_AXIS</td>
<td>Axis for radial hole</td>
</tr>
<tr>
<td>PRO_E_HLE_DIM_RAD</td>
<td>Radial distance with regard to PRO_E_HLE_AXIS</td>
</tr>
<tr>
<td>PRO_E_HLE_REF_PLANE_1</td>
<td>Reference plane against which angular distance will be measured</td>
</tr>
<tr>
<td>PRO_E_HLE_REF_ANG_1</td>
<td>Angular distance with regard to PRO_E_HLE_REF_PLANE_1</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_HLE_PRIM_REF</td>
<td>Primary Selection, planar surface/datum plane</td>
</tr>
<tr>
<td>PRO_E_HLE_PL_TYPE</td>
<td>set to PRO_HLE_PL_TYPE_RAD_DIA_DIM</td>
</tr>
<tr>
<td>PRO_E_HLE_AXIS</td>
<td>Axis for radial hole</td>
</tr>
<tr>
<td>PRO_E_HLE_DIM_DIA</td>
<td>Diameter distance with regard to PRO_E_HLE_AXIS</td>
</tr>
<tr>
<td>PRO_E_HLE_REF_PLANE_1</td>
<td>Reference plane against which angular distance will be measured</td>
</tr>
<tr>
<td>PRO_E_HLE_REF_ANG_1</td>
<td>Angular distance with regard to PRO_E_HLE_REF_PLANE_1</td>
</tr>
</tbody>
</table>
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”.

<table>
<thead>
<tr>
<th>PRO_E_HLE_PRIM_REF</th>
<th>Primary Selection, planar surface/datum plane</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_HLE_PL_TYPE</td>
<td>is set to PRO_HLE_PL_TYPE_RAD_LIN_DIM</td>
</tr>
<tr>
<td>PRO_E_HLE_AXIS</td>
<td>Axis for radial hole</td>
</tr>
<tr>
<td>PRO_E_HLE_DIM_LIN</td>
<td>Linear distance with regard to PRO_E_HLE_AXIS</td>
</tr>
<tr>
<td>PRO_E_HLE_REF_PLANE_1</td>
<td>Reference plane against which angular distance will be measured</td>
</tr>
<tr>
<td>PRO_E_HLE_REF_ANG_1</td>
<td>Angular distance with regard to PRO_E_HLE_REF_PLANE_1</td>
</tr>
</tbody>
</table>

Radial Hole on Cone or Cylinder

This hole placement type requires the selection of a cone or cylinder for primary placement.

<table>
<thead>
<tr>
<th>PRO_E_HLE_PRIM_REF</th>
<th>Primary Selection, Cone or Cylinder</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_HLE_PL_TYPE</td>
<td>is set to PRO_HLE_PL_TYPE_RAD</td>
</tr>
<tr>
<td>PRO_E_HLE_REF_PLANE</td>
<td>Reference plane against which angular distance will be measured</td>
</tr>
<tr>
<td>PRO_E_HLE_REF_ANG</td>
<td>Angular distance with regard to PRO_E_HLE_REF_PLANE</td>
</tr>
<tr>
<td>PRO_E_HLE_NORM_PLA</td>
<td>Reference plane for linear measurement</td>
</tr>
<tr>
<td>PRO_E_HLE_NORM_OFFSET</td>
<td>Distance with regard to PRO_E_HLE_NORM_PLA</td>
</tr>
</tbody>
</table>

Coaxial Hole with Axis as Primary Reference

Coaxial hole placement requires an axis and a placement plane to complete the placement.

<table>
<thead>
<tr>
<th>PRO_E_HLE_PRIM_REF</th>
<th>Primary Selection, Axis</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_HLE_PL_TYPE</td>
<td>is set to PRO_HLE_PL_TYPE_COAX</td>
</tr>
<tr>
<td>PRO_E_HLE_PLCMNT_PLANE</td>
<td>Placement surface</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>PRO_E_HLE_PRIM_REF</th>
<th>Primary Selection, Plane, Cone or Cylinder</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_HLE_PL_TYPE</td>
<td>is set to PRO_HLE_PL_TYPE_COAX</td>
</tr>
<tr>
<td>PRO_E_HLE_AXIS</td>
<td>Axis</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>PRO_E_HLE_PRIM_REF</th>
<th>Primary Selection, on Surface Created Datum Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_HLE_PL_TYPE</td>
<td>is set to PRO_HLE_PL_TYPE_ON_PNT</td>
</tr>
</tbody>
</table>

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_SCRWSIZE, 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.
This chapter describes the element tree structure required to create patterns of features. The chapter on ‘Principles of Feature Creation’ is a necessary background for this topic. Read that chapter before this one.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>21 - 2</td>
</tr>
<tr>
<td>The Element Tree for Pattern Creation</td>
<td>21 - 2</td>
</tr>
<tr>
<td>Creating a Pattern</td>
<td>21 - 8</td>
</tr>
<tr>
<td>Obtaining the Element Tree for a Pattern</td>
<td>21 - 13</td>
</tr>
</tbody>
</table>
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 ‘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.
The following figure shows the element tree for patterns.
The element with the identifier PRO_E_PAT_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_PAT_TYPE element are as follows:

- PRO_PAT_REF_DRIVEN—Reference-driven pattern
- PRO_PAT_DIM_DRIVEN—Dimension-driven pattern
- PRO_PAT_TABLE_DRIVEN—Table-driven pattern

The element with the identifier PRO_E_PAT_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_PAT_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.

The following sections describe the three types of pattern in more detail.

**Reference-Driven Patterns**

A reference-driven pattern uses an existing pattern as a guide for the placement of the new pattern members. Consequently, if the pattern type is PRO_PAT_REF_DRIVEN, the element tree requires only that you specify the type of the reference pattern. The element with identifier PRO_E_PAT_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-Driven Patterns**

If the pattern type is PRO_PAT_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.
The elements with identifiers PRO_E_PAT_FIRST_DIR and PRO_E_PAT_SECOND_DIR contain information about the pattern dimensions.

These elements are array elements that contain as many PRO_E_PAT_DIR_DIM_COMPOUND elements as are required to complete the pattern. The following table describes the contents of the PRO_E_PAT_DIR_DIM_COMPOUND element.

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Element Name</th>
<th>Data Type</th>
<th>Valid Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_PAT_DIR_</td>
<td>Dimension</td>
<td>PRO_E_PAT_DIR_DIMENSION</td>
<td>Any selected dimension</td>
</tr>
<tr>
<td>DIMENSION</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRO_E_PAT_DIR_VAR_TYPE</td>
<td>Variation type</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>PRO_PAT_RELATION_DRIVEN, PRO_PAT_VALUE_DRIVEN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRO_E_PAT_DIR_VAR_VALUE</td>
<td>Variation value</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(increment)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRO_E_PAT_RELATION_EDIT</td>
<td>Relation</td>
<td>PRO_VALUE_TYPE_WSTRING</td>
<td></td>
</tr>
</tbody>
</table>

The element PRO_E_PAT_DIR_VAR_TYPE specifies whether the pattern increment is relation-driven or value-driven. If the increment is relation-driven, the element PRO_E_PAT_RELATION_EDIT contains an array of wide strings whose members are individual relations.

The elements PRO_E_PAT_FIRST_DIR_NUM_INST and PRO_E_PAT_SECOND_DIR_NUM_INST specify the number of instances in each of the pattern dimensions.

**Note:** If your pattern extends in only one direction, you must still 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-Driven Patterns**

If the pattern type is PRO_PAT_TABLE_DRIVEN, your element tree must contain the table-driven dimensions and table information (variation in dimensions for each instance).
Creating Patterns

The PRO_E_PAT_MULT_TABLE_DIMS element is an array that contains one table dimension (PRO_E_PAT_TABLE_DIM) element for each dimension to be varied in the tables. The value of each PRO_E_PAT_TABLE_DIM element is a ProSelection object for the corresponding dimension.

The PRO_E_PAT_TABLE_LIST element is an array element that contains all the tables that control the pattern. This element should contain one PRO_E_PAT_TABLE element for each table.

Each PRO_E_PAT_TABLE element contains the name of the table (PRO_E_PAT_TABLE_NAME) and instances of the table (PRO_E_PAT_TABLE_INSTANCES). The PRO_E_PAT_TABLE_INSTANCES element contains one PRO_E_PAT_TABLE_INSTANCE element for each instance (row) in the pattern table.

Each PRO_E_PAT_TABLE_INSTANCE element contains an index number (PRO_E_PAT_TABLE_INSTANCE_INDEX) element and a dimensions (PRO_E_PAT_TABLE_INSTANCE_DIMS) element. The PRO_E_PAT_TABLE_INSTANCE_DIMS element is an array element that must contain one dimension value (PRO_E_PAT_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_PAT_TABLE element.

<table>
<thead>
<tr>
<th>Element ID Values</th>
<th>Element Name</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_PAT_TABLE_NAME</td>
<td>Table name</td>
<td>PRO_VALUE_TYPE_WSTRING</td>
</tr>
<tr>
<td>PRO_E_PAT_TABLE_INSTANCES</td>
<td>Table instances</td>
<td>Array</td>
</tr>
<tr>
<td>PRO_E_PAT_TABLE_INSTANCE</td>
<td>Table instance</td>
<td>Compound</td>
</tr>
<tr>
<td>PRO_E_PAT_TABLE_INSTANCE_INDEX</td>
<td>Instance index</td>
<td>PRO_VALUE_TYPE_INT</td>
</tr>
<tr>
<td>PRO_E_PAT_TABLE_INSTANCE_DIMS</td>
<td>Dimension variations</td>
<td>Compound</td>
</tr>
</tbody>
</table>
The element PRO_E_PAT_TABLE_SET_ACTIVE sets the active table for the pattern. Valid values are 0 (for the first table) through \((\text{num\_tables} - 1)\), where \text{num\_tables} is the number of tables in the element tree.

**Fill Patterns**

Creation of Fill patterns via the element tree is not supported in Pro/ENGINEER Wildfire.

### Creating a Pattern

**Function introduced:**

- **ProPatternCreate()**

  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()**. (The function **ProFeaturePatternGet()** is described in the section Manipulating Patterns on page 12 - 10.)

<table>
<thead>
<tr>
<th>Element ID Values</th>
<th>Element Name</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_PAT_TABLE_INSTANCE_DIM_VALUE</td>
<td>Dimension value</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
</tr>
</tbody>
</table>
**Example 1: Creating a Pattern**

The following example code shows how to construct the element tree for a dimension-driven pattern with identical regeneration. The selected feature is patterned in one direction only.

```c
#include <ProToolkit.h>
#include <prodevelop.h>
#include <ProObjects.h>
#include <ProSelection.h>
#include <ProMdl.h>
#include <ProMessage.h>
#include <ProUtil.h>
#include <ProElement.h>
#include <ProMdl.h>
#include <ProPattern.h>
#include <prodevdim.h>

Application includes

Application global and external data

static ProName msg_file;

FUNCTION: UserPatternCreate()
PURPOSE: Create a pattern.

ProError UserPatternCreate()
{
  ProSelection *p_sel;  /* for selection */
  int n_sel;
  ProFeature leadfeat;  /* feature to be patterned */
  double increment;    /* data for pattern */
  int num_inst;
  int *dim_ids;        /* feature dimensions */
  int num_dims;
  ProSelection sel_dim; /* selected dimension */
  ProElement elem_tree, elem_pattype, elem_patregen;
  ProElement elem_patdir1, elem_dir1var, elem_dir1vtype;
  ProElement elem_dir1dim, elem_dir1inc, elem_dir1num;
  ProElement elem_patdir2, elem_dir2num;
```
```c
ProValue value;    /* values in the elements */
ProValueData value_data;    /* return status */
int err;    /* return status */
int i;
ProMdl model;    /* current model */

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);
}

Obtain the feature to pattern.

ProMessageDisplay (msg_file, "USER %0s",
    "Select a feature to pattern.");
err = ProSelect ("feature", 1, NULL, NULL, NULL, NULL, NULL, &p_sel,
    &n_sel);
if (err != PRO_TK_NO_ERROR)
{
    ProMessageDisplay (msg_file, "USER %0s",
        "Error or abort during feature selection.");
    return (err);
}

err = ProSelectionModelitemGet (p_sel[0],
    (ProModelitem*)&leadfeat);

Display the feature dimensions.

num_dims = prodb_get_feat_dim_ids ((Prohandle) model,
    leadfeat.id, &dim_ids);
for(i = 0; i < num_dims; i++)
{
    err = prodim_display_dimension ((Prohandle) model, NULL,
        dim_ids[i], NULL, NULL, NULL);
}

Obtain the dimension to be incremented.

ProMessageDisplay (msg_file, "USER %0s", "Select 1 dimension.");
err = ProSelect ("dimension", 1, NULL, NULL, NULL, NULL, &p_sel,
    &n_sel);
ProSelectionCopy (p_sel[0], &sel_dim);
```

ProTOOLKIT User's Guide
Obtain the increment and number of instances.

```c
create_pattern_element_tree_root();
add_pattern_type_element();
add_regeneration_method_element();
add_first_direction_element();
add_first_direction_dimension_variation_element();
add_first_direction_dimension_element();
```
err = ProElemtreeElementAdd (elem_dir1var, NULL, elem_dir1dim);
/* Add the first direction variation type. */
----------------------------------------------------------------
err = ProElementAlloc (PRO_E_PAT_DIR_VAR_TYPE, &elem_dir1vtype);
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_PAT_VALUE_DRIVEN;
err = ProValueAlloc (&value);
err = ProValueDataSet (value, &value_data);
err = ProElementValueSet (elem_dir1vtype, value);
err = ProElemtreeElementAdd (elem_dir1var, NULL, elem_dir1vtype);
/* Add the first direction variation (increment) element. */
----------------------------------------------------------------
err = ProElementAlloc (PRO_E_PAT_DIR_VAR_VALUE, &elem_dir1inc);
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = increment;
err = ProValueAlloc (&value);
err = ProValueDataSet (value, &value_data);
err = ProElementValueSet (elem_dir1inc, value);
err = ProElemtreeElementAdd (elem_dir1var, NULL, elem_dir1vtype);
/* Add the first direction number of instances element. */
----------------------------------------------------------------
err = ProElementAlloc (PRO_E_PAT_FIRST_DIR_NUM_INST, &elem_dir1num);
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = num_inst;
err = ProValueAlloc (&value);
err = ProValueDataSet (value, &value_data);
err = ProElementValueSet (elem_dir1num, value);
err = ProElemtreeElementAdd (elem_dir1num);
/* Add the second direction element to the tree. */
----------------------------------------------------------------
err = ProElementAlloc (PRO_E_PAT_SECOND_DIR, &elem_patdir2);
err = ProElemtreeElementAdd (elem_tree, NULL, elem_patdir2);
/* Add the second direction number of instances element. */
----------------------------------------------------------------
err = ProElementAlloc (PRO_E_PAT_SECOND_DIR_NUM_INST, &elem_dir2num);
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = 1;
err = ProValueAlloc (&value);
err = ProValueDataSet (value, &value_data);
err = ProElementValueSet (elem_dir2num, value);
err = ProElemtreeElementAdd (elem_dir2num);
/* Display the element tree. */
----------------------------------------------------------------
ProMessageDisplay (msg_file, "USER %0s", "Element tree is complete.");
err = ProUtilElementtreePrint (elem_tree, PRO_TEST_INFO_WINDOW, NULL);
/*-----------------------------------------------*/
Create the pattern.
="/-----------------------------------------------*/
err = ProPatternCreate (&leadfeat, PRO_FEAT_PATTERN, elem_tree);
return (err);
}

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 16 - 13) and Feature Element Paths (on page 16 - 11), such as the functions ProElement*Get(), ProElement*Visit(), and ProElementArrayGet().

Note: Inspection of Fill patterns is not supported via the element tree in Pro/ENGINEER Wildfire. The function ProPatternElemtreeCreate() returns PRO_TK_NOT_IMPLEMENTED for fill patterns.
Assembling Components

This chapter describes how to use the concepts of feature creation to assemble components into an assembly. Read the chapter ‘Principles of Feature Creation’ before this chapter.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembling Components by Functions</td>
<td>22 - 2</td>
</tr>
<tr>
<td>Assembling a Component Parametrically</td>
<td>22 - 3</td>
</tr>
<tr>
<td>Redefining Components Interactively</td>
<td>22 - 12</td>
</tr>
<tr>
<td>Assembling Components by Feature Creation</td>
<td>22 - 13</td>
</tr>
<tr>
<td>The Element Tree for a Component Assembly</td>
<td>22 - 14</td>
</tr>
</tbody>
</table>
Assembling Components by Functions

Functions introduced:

- `ProAsmcompCreateCopy()`
- `ProAsmcompAssemble()`
- `ProAsmcompPositionGet()`
- `ProAsmcompPositionSet()`
- `ProAsmcompConstraintsGet()`
- `ProAsmcompConstraintsSet()`

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.
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 ProAsmcompTypeGet() 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 surface.

• PRO_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—Use this option to select a reference on the component and assembly. The Automatic constraint type will select an appropriate constraint based on the references selected and their orientation. This is not typically used during programmatic feature creation and redefinition.

• 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.

ProAsmcompconstraintCompreferenceSet() 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**—The constraint should be ignored during regeneration.
- **PRO_ASM_CONSTR_ATTR_NONE**—No constraint attributes are specified. This is the default value.

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.

```c
#include <ProToolkit.h>
#include <ProAsmcomp.h>
#include <ProMessage.h>
#include <ProSolid.h>
#include <ProArray.h>
#include <TestError.h>
#define MSGFIL L##"msg_ugasmcomp.txt"

Prototypes

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 dtsiide;
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, &dtsiide);
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, &dtsiide);
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) {
    sprintf (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: 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)");

22 - 8 Pro/TOOLKIT User's Guide
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.

```c
#include <ProAsmcomp.h>
#include <ProMessage.h>
#include <ProSolid.h>
#include <ProArray.h>
#include <TestError.h>
#define MSGFIL L"msg_ugasmcomp.txt"

/*=====================================================================*
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;
  ProAsmcompasmpath 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);

/*-----------------------------------------------------------------*\nPrepare the constraints array
\*-----------------------------------------------------------------*/
ProArrayAlloc (0, sizeof (ProAsmcompconstraint), 1, (ProArray*)&constraints);
for (i = 0; i < 3; i++)
{
  /*-----------------------------------------------------------------*\n  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;
    }

  /*-----------------------------------------------------------------*\n  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;
    }

  /*-----------------------------------------------------------------*\n  For the assembly reference, initialize a component path.
  This is necessary even if the reference geometry is in the assembly.
  \*-----------------------------------------------------------------*/
  ProAsmcompPathInit (asm_model, c_id_table, 0, &comp_path);

  /*-----------------------------------------------------------------*\n  Allocate the references
  \*-----------------------------------------------------------------*/
  ProSelectionAlloc (&comp_path, &asm_datum, &asm_sel);
  ProSelectionAlloc (NULL, &comp_datum, &comp_sel);

  /*-----------------------------------------------------------------*\n  Allocate and fill the constraint.
  \*-----------------------------------------------------------------*/
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()`
- `ProAsmcompConstrRemove()`
- `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 `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 `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 Feature Creation**

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 `ProFeatureElemtreeCreate()`.
The Element Tree for a Component Assembly

The element tree for a component assembly is documented in the header file ProAsmcomp.h, and is shown in the following figure. The tree is very similar to that for creating datum features, as described in the chapter ‘Creating Datum Features’.

The element PRO_E_COMPONENT_MODEL identifies the part or assembly to be assembled as the new component. Its type is pointer, and it is set to the ProSolid handle for the component.
The element PRO_E_COMPONENT_INIT_POS defines the absolute position of the component in the absence of full 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.

The element PRO_E_COMPONENT_CONSTRAINTS is an array of elements of type PRO_E_COMPONENTCONSTRAINT, similar to datum plane feature element trees.

Each constraint element contains the following elements:

- **PRO_E_COMPONENT_CONSTR_TYPE**—Has values such as PRO_ASM_MATE, PRO_ASM_MATE_OFF, and PRO_ASM_ALIGN.
- **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.
- **PRO_E_COMPONENT_CONSTR_REF_OFFSET**—Gives the offset value, if the constraint type is an offset.

If the referenced geometry item in the component or assembly is a datum plane, the elements PRO_E_COMPONENT_COMP_ORIENT and PRO_E_COMPONENT_ASSM_ORIENT indicate which side of the plane is being referenced. A value of +1 indicates the yellow (default) side, and −1 indicates the red side.

**Example 3: Assembling a Part into an Assembly**

This example shows how to use the feature creation technique to assemble a part into an assembly using a single coordinate system constraint.

The function *UsrAsmByCsys()* is a command function that forms a simple demonstration harness. The function that performs the component assembly is called *UsrComponentAssembleCsys()*.

The following example code shows data structure for searching for the coordinate system datum by name.
typedef struct csysfind_data
{
  ProSolid solid;
  ProName name;
  ProGeomitem csys;
  ProBoolean found;
} CsysFind_data;

Action function for finding the coordinate system datums

ProError UsrCsysFindAction(
  ProCsys csys,
  ProError filter_status,
  ProAppData data)
{
  CsysFind_data *csys_data=(CsysFind_data*)data;
  int csys_id;
  ProGeomitem csys_geomitem;
  ProName name;

  Get the name of this coordinate system.

  ProCsysIdGet (csys, &csys_id);
  ProModelitemInit (csys_data->solid, csys_id, PRO_CSYS, &csys_geomitem);
  ProModelitemNameGet (&csys_geomitem, name);

  If it is the name you want, set the data and stop visiting.

  if (!ProUtilWstrCmp (name, csys_data->name))
  {
    csys_data->found = PRO_B_TRUE;
    memcpy (&csys_data->csys, &csys_geomitem, sizeof(ProGeomitem));
    return (PRO_TK_E_FOUND);
  }
  return (PRO_TK_NO_ERROR);
}

Find a coordinate system datum by name.

ProError UsrCsysFindbyname (
  ProSolid solid,
  char *csys_name,
  ProGeomitem *csys_geom)
{
  CsysFind_data data;

  Initialize the search data.

  data.solid = solid;

  Find the coordinate system datum.

  if (!ProUtilWstrCmp (csys_name, csys_data->name))
  {
    csys_data->found = PRO_B_TRUE;
    memcpy (&csys_data->csys, &csys_geomitem, sizeof(ProGeomitem));
    return (PRO_TK_E_FOUND);
  }
  return (PRO_TK_NO_ERROR);
}
ProStringToWstring (data.name, csys_name);
data.found = PRO_B_FALSE;
/*--------------------------------------------------------------------*/
Visit all the coordinate systems.
/*/--------------------------------------------------------------------*/
ProSolidCsysVisit (solid, UsrCsysFindAction, NULL, (ProAppData)&data);
/*/--------------------------------------------------------------------*/
If found, set the output argument.
/*/--------------------------------------------------------------------*/
if (data.found)
{
    memcpy (csys_geom, &data.csys, sizeof(ProGeomitem));
    return (PRO_TK_NO_ERROR);
}
return (PRO_TK_E_NOT_FOUND);

 /*--------------------------------------------------------------------*/
Assemble a component by coordinate system.
\--------------------------------------------------------------------*/
ProError UsrComponentAssembleCsys (
    ProSolid       assembly,
    char          *asm_csys_name,
    ProSolid       component,
    char          *comp_csys_name)
{
    ProError       status;
    ProGeomitem    asm_csys, comp_csys;
    ProElement     elem_root, elem_ftype, elem_model, elem_constrs,
                    elem_constr, elem_contype, elem_compref, elem_asmref;
    ProValue       value;
    ProValueData   value_data;
    ProModelitem   modelitem;
    ProErrorlist   errors;
    ProFeature     feature;
    ProFeatureCreateOptions  opts[1];
    int                      memb_id_tab[1];
    ProAsmcomppath           comppath;
    /*--------------------------------------------------------------------*/
    Find the named assembly coordinate system.
    /*--------------------------------------------------------------------*/
    status = UsrCsysFindbyname (assembly, asm_csys_name, &asm_csys);
    if (status != PRO_TK_NO_ERROR)
        return (PRO_TK_E_NOT_FOUND);
    /*--------------------------------------------------------------------*/
    Find the named component coordinate system.
    /*--------------------------------------------------------------------*/
    status = UsrCsysFindbyname (component, comp_csys_name,
                                &comp_csys);
    if (status != PRO_TK_NO_ERROR)
return (PRO_TK_E_NOT_FOUND);
/*-----------------------------------------------*/
    Allocate the element tree root.
/*-----------------------------------------------*/
    ProElementAlloc (PRO_E_FEATURE_TREE, &elem_root);
/*-----------------------------------------------*/
    Add the feature type element.
/*-----------------------------------------------*/
        ProElementAlloc (PRO_E_FEATURE_TYPE, &elem_ftype);
        value_data.type = PRO_VALUE_TYPE_INT;
        value_data.v.i = PRO_FEAT_COMPONENT;
        ProValueAlloc (&value);
        ProValueDataSet (value, &value_data);
        ProElementValueSet (elem_ftype, value);
        status = ProElemtreeElementAdd (elem_root, NULL, elem_ftype);
        printf("ProElemtreeElementAdd (elem_ftype) status = %d\n", status);
/*-----------------------------------------------*/
        Add the component model element.
/*-----------------------------------------------*/
        ProElementAlloc (PRO_E_COMPONENT_MODEL, &elem_model);
        value_data.type = PRO_VALUE_TYPE_POINTER;
        value_data.v.p = component;
        ProValueAlloc (&value);
        ProValueDataSet (value, &value_data);
        ProElementValueSet (elem_model, value);
        status = ProElemtreeElementAdd (elem_root, NULL, elem_model);
        printf("ProElemtreeElementAdd (elem_model) status = %d\n", status);
/*-----------------------------------------------*/
        Add the component constraints element.
/*-----------------------------------------------*/
        ProElementAlloc (PRO_E_COMPONENT_CONSTRAINTS, &elem_constrs);
        status = ProElemtreeElementAdd (elem_root, NULL, elem_constrs);
/*-----------------------------------------------*/
        Add a constraint element.
/*-----------------------------------------------*/
        ProElementAlloc (PRO_E_COMPONENT_CONSTRAINT, &elem_constr);
        status = ProElemtreeElementAdd (elem_constrs, NULL, elem_constr);
/*-----------------------------------------------*/
        Make the constraint type CSYS.
/*-----------------------------------------------*/
        ProElementAlloc (PRO_E_COMPONENT_CONSTRAINT_TYPE, &elem_contype);
        value_data.type = PRO_VALUE_TYPE_INT;
        value_data.v.i = PRO_ASM_CSYS;
        ProValueAlloc (&value);
        ProValueDataSet (value, &value_data);
        ProElementValueSet (elem_contype, value);
        ProElemtreeElementAdd (elem_constr, NULL, elem_contype);
/*-----------------------------------------------*/
        Add the component reference element.
Assembling Components

ProElementAlloc (PRO_E_COMPONENT_COMP_CONSTR_REF, &elem_compref);
value_data.type = PRO_VALUE_TYPE_SELECTION;
ProSelectionAlloc (NULL, &comp_csys, &sel);
value_data.v.r = sel;
ProValueAlloc (&value);
ProValueDataSet (value, &value_data);
ProElementValueSet (elem_compref, value);
ProElemtreeElementAdd (elem_constr, NULL, elem_compref);

ProElementAlloc (PRO_E_COMPONENT_ASSEM_CONSTR_REF, &elem_asmref);
value_data.type = PRO_VALUE_TYPE_SELECTION;
memb_id_tab[0] = -1;
ProAsmcomppathInit (assembly, memb_id_tab, 0, &comppath);
ProSelectionAlloc (&comppath, &asm_csys, &sel);
value_data.v.r = sel;
ProValueAlloc (&value);
ProValueDataSet (value, &value_data);
ProElementValueSet (elem_asmref, value);
ProElemtreeElementAdd (elem_constr, NULL, elem_asmref);

ProMdlToModelitem (assembly, &modelitem);
ProSelectionAlloc (NULL, &modelitem, &sel);

opts[0] = PRO_FEAT_CR_INCOMPLETE_FEAT;
status = ProFeatureCreate (sel, elem_root, opts, 1, &feature,
&errors);
if (status != PRO_TK_NO_ERROR)
ProUtilFeatErrsPrint (&errors);
return (status);

UsrAsmByCsys()
{
    ProSolid   assembly, part;
    ProName    name;
    ProMdlCurrentGet ((ProMdl*)&assembly);

    ProMessageDisplay (msgfil, "USER Component part : \\
|||p1");
if (ProMessageStringRead(PRO_NAME_SIZE, name) != PRO_TK_NO_ERROR)
    ProStringToWstring (name, "p1");
if (ProMdlRetrieve (name, PRO_MDL_PART, (ProMdl*)&part) !=
    PRO_TK_NO_ERROR)
{
    ProMessageDisplay (msgfil, "USER Failed to retrieve %0s.prt",
                        name);
    return (0);
}

/*--------------------------------------------------------------------*
Assemble the part by the assembly coordinate system "ASCO"
and the part coordinate system "CS0".
--------------------------------------------------------------------*/
status = UsrComponentAssembleCsys (assembly, "ACS0", part, "CS0");
if (status != PRO_TK_NO_ERROR)
    ProMessageDisplay (msgfil, "USER Failed to assemble component");
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.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>23 - 2</td>
</tr>
<tr>
<td>Creating Section Models</td>
<td>23 - 2</td>
</tr>
</tbody>
</table>
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.

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 ‘Creating 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 'Creating 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.

```c
ProSection    section;
ProName       wname;

ProSection2DAlloc (&section);
ProStringToWstring (wname, "demo");
ProSectionNameSet (section, wname);
```

To free a section allocated with `ProSection2DAlloc()`, you must use `ProSectionFree()`.
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.

```c
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.

### Adding Section Dimensions

Functions introduced:

- **ProSecdimCreate()**
- **ProSecdimDelete()**
- **ProSecdimDiameterSet()**
- **ProSecdimDiameterClear()**
- **ProSecdimDiameterInquire()**

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.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_TK_DIM_LINE</td>
<td>Length of a line</td>
</tr>
<tr>
<td>PRO_TK_DIM_LINE_POINT</td>
<td>Distance between a line and a vertex</td>
</tr>
<tr>
<td>PRO_TK_DIM_RAD</td>
<td>Radius of an arc or a circle</td>
</tr>
<tr>
<td>PRO_TK_DIM_DIA</td>
<td>Diameter of an arc or a circle</td>
</tr>
<tr>
<td>PRO_TK_DIM_LINE_LINE</td>
<td>Distance between two lines</td>
</tr>
<tr>
<td>PRO_TK_DIM_PNT_PNT</td>
<td>Distance between two points</td>
</tr>
<tr>
<td>PRO_TK_DIM_PNT_PNT_HORIZ</td>
<td>Distance between two points (X coordinates)</td>
</tr>
<tr>
<td>PRO_TK_DIM_PNT_PNT_VERT</td>
<td>Distance between two points (Y coordinates)</td>
</tr>
<tr>
<td>PRO_TK_DIM_AOC_AOC_TAN_HORIZ</td>
<td>Horizontal distance between two arcs or circles</td>
</tr>
</tbody>
</table>
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 that 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.

```c
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.

```c
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.
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.

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.

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.

Section Constraints

Functions introduced:

• ProSectionConstraintsIdsGet()
• ProSectionConstraintsGet()
• ProSectionConstraintDeny()

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.

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:

<table>
<thead>
<tr>
<th>Constraint Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_CONSTRAINTSAME_POINT</td>
<td>Make the points coincident.</td>
</tr>
<tr>
<td>PRO_CONSTRAINT_HORIZONTAL_ENT</td>
<td>Make the entity horizontal.</td>
</tr>
<tr>
<td>PRO_CONSTRAINT_VERTICAL_ENT</td>
<td>Make the entity vertical.</td>
</tr>
<tr>
<td>PRO_CONSTRAINT_PNT_ON_ENT</td>
<td>Place the point on the entity.</td>
</tr>
<tr>
<td>PRO_CONSTRAINT_TANGENT_ENTS</td>
<td>Make the entities tangent.</td>
</tr>
<tr>
<td>PRO_CONSTRAINT_ORTHOG_ENTS</td>
<td>Make the entities perpendicular.</td>
</tr>
<tr>
<td>PRO_CONSTRAINT_EQUAL_RADII</td>
<td>Make the arcs or circles of equal radius.</td>
</tr>
<tr>
<td>PRO_CONSTRAINT_PARALLEL_ENTS</td>
<td>Make the entities parallel.</td>
</tr>
<tr>
<td>PRO_CONSTRAINT_EQUAL_SEGMENTS</td>
<td>Make the segments of equal length.</td>
</tr>
<tr>
<td>PRO_CONSTRAINT_COLLINEAR_LINES</td>
<td>Make lines co-linear.</td>
</tr>
<tr>
<td>PRO_CONSTRAINT_90_ARC</td>
<td>Make the arcs 90 degrees.</td>
</tr>
<tr>
<td>PRO_CONSTRAINT_180_ARC</td>
<td>Make the arcs 180 degrees.</td>
</tr>
<tr>
<td>PRO_CONSTRAINT_HORIZONTAL_ARC</td>
<td>Make the arcs horizontal.</td>
</tr>
<tr>
<td>PRO_CONSTRAINT_VERTICAL_ARC</td>
<td>Make the arcs vertical.</td>
</tr>
<tr>
<td>PRO_CONSTRAINT_SYMMETRY</td>
<td>Impose symmetry.</td>
</tr>
<tr>
<td>PRO_CONSTRAINTSAME_COORD</td>
<td>Assume the endpoints and centers of arcs to have the same coordinates.</td>
</tr>
</tbody>
</table>
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.

### 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.

```c
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 1: Creating a Section Model**

The following example code illustrates how to use all the functions described in this chapter to create a section model.

```c
UserSectionCreateExample();
ProDemoSectCreate();
```

```c
 FILE : UgSectModelCreate.c
 PURPOSE : Feature Creation

/*====================================================================*
 FILE    : UgSectModelCreate.c
 PURPOSE : Feature Creation
 \*====================================================================*/

#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 "TestError.h"

/*---------------------- Function Prototypes -------------------------*/
ProError UserSectionCreateExample();
ProError ProDemoSectCreate();

typedef char ProUtilCname[PRO_NAME_SIZE];

/*------------------------- Global Data -----------------------------*/

/*====================================================================*
 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];
/*-----------------------------------------------------------*\nCheck 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);
/*-----------------------------------------------------------*\nAllocate the handle for the 2-D section.
\*-----------------------------------------------------------*/
if (alloc == PRO_B_TRUE)
   { status = ProSection2DAlloc (&section);
   } else
   { section = *p_section;
   }
/*-----------------------------------------------------------*\nSet the name of the section.
\*-----------------------------------------------------------*/
pro_str_to_wstr (wname, name);
status = ProSectionNameSet (section, wname);
/*-----------------------------------------------------------*\nSet 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.

```c
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).

```c
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).

```c
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.

```c
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.

```c
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.

```c
status = ProSecerrorAlloc (&errors);
```

Solve the section.

```c
solve_status = ProSectionSolve (section, &errors);
```

If the solve failed, report error messages and exit.

```c
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);
        printf ("Error %d message : %s
", e, msg);
    }
    return (-1);
}
```

Save the section.

```c
status = ProMdlSave (section);
```

Return the section handle

```c
*p_section = section;
return (0);
```

FUNCTION: UserSectionCreateExample()

PURPOSE: Invokes the function to create the Section model

```c
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 );
Creating 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.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>24 - 2</td>
</tr>
<tr>
<td>Creating Features Containing Sections</td>
<td>24 - 3</td>
</tr>
<tr>
<td>Creating Features with 2D Sections</td>
<td>24 - 5</td>
</tr>
<tr>
<td>Creating Features with 3D Sections</td>
<td>24 - 5</td>
</tr>
<tr>
<td>Reference Entities and Use Edge</td>
<td>24 - 6</td>
</tr>
<tr>
<td>Reusing Existing Sketches</td>
<td>24 - 8</td>
</tr>
</tbody>
</table>
Overview

The chapter ‘Principles of Feature Creation’ explains how to create a simple feature using the feature element tree, and the chapter on ‘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 24-1: Element Tree for Sketched Features

```
PRO_E_STD_SECTION  (PRO_E_STD_TRAJ)
    PRO_E_STD_SEC_SETUP_PLANE
        PRO_E_STD_SEC_PLANE
        PRO_E_STD_SEC_PLANE_VIEW_DIR
        PRO_E_STD_SEC_PLANE_ORIENT_DIR
        PRO_E_STD_SEC_PLANE_ORIENT_REF
    PRO_E_SKETCHER
```

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. The element PRO_E_SKETCHER is of type POINTER, and its value is the object ProSection, introduced in the chapter on Sections.
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.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Header</th>
<th>Element containing Subtree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extrude</td>
<td>ProExtrude.h</td>
<td>PRO_E_STDSECTION</td>
</tr>
<tr>
<td>Revolve</td>
<td>ProRevolve.h</td>
<td>PRO_E_STDSECTION</td>
</tr>
<tr>
<td>Rib</td>
<td>ProRib.h</td>
<td>PRO_E_STDSECTION</td>
</tr>
<tr>
<td>Hole</td>
<td>ProHole.h</td>
<td>PRO_E_SKETCHER (2D)</td>
</tr>
<tr>
<td>Fill (Flat datum surface)</td>
<td>ProFlatSrf.h</td>
<td>PRO_E_STDSECTION</td>
</tr>
<tr>
<td>Sketched datum curve</td>
<td>ProDtmCrv.h</td>
<td>PRO_E_STDSECTION</td>
</tr>
<tr>
<td>Sketched datum point</td>
<td>ProDtmPnt.h</td>
<td>PRO_E_STDSECTION</td>
</tr>
<tr>
<td>Simple (constant) sweep</td>
<td>ProSweep.h</td>
<td>PRO_E_SWEEP_SPINE PRO_E_SWEEP SECTION (2D)</td>
</tr>
</tbody>
</table>

Creating Features Containing Sections

The chapter ‘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 ‘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 using **ProFeatureElemValueGet()** or **ProFeatureElemtreeCreate()**.

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 are features whose 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.

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 Owning Model

Functions 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()

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 skinner 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 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.
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.

```c
/*====================================================================*
 FILE    : UgSktExtrusionCreate.c
 */====================================================================*/

#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 wchar_t msgfile[PRO_NAME_SIZE];

/*---------------------- Function Prototypes ------------------------*/
int ProDemoBaseExtrudeProtrCreate();

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            errs;
    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;
    ProValue                value;
    double                  width;
    double                  height;
    double                  bite_radius;
    double                  bite_height;
    char                    name[PRO_NAME_SIZE];
    ProBoolean              alloc;
    ProError    err;
    ProSelection *     sketch_selection;
    ProSelection *     selection_array;
    int     n_select;

    /*Allocate the element tree.*/
    /*Allocate the element tree.*/
    err = ProElementAlloc (PRO_E_FEATURE_TREE, &(elem.tree));

    /*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;
err = ProTestFeatElemAdd (&elem);

/*---------------------------------------------------------------*\
 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;
err = ProTestFeatElemAdd (&elem);

/*---------------------------------------------------------------*\
 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;
err = ProTestFeatElemAdd (&elem);

/*---------------------------------------------------------------*\
 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;
err = ProTestFeatElemAdd (&elem);

/*---------------------------------------------------------------*\
 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;
err = ProTestFeatElemAdd (&elem);

/*---------------------------------------------------------------*\
 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;
err = ProTestFeatElemAdd (&elem);

/*---------------------------------------------------------------*/
Add the section depth elements to the element tree.

```c
/*-----------------------------------------------*/
    elem.parent_element = elem.tree;
    elem.elem_id = PRO_E_STD_EXT_DEPTH;
    elem.value_data.type = -1;
    elem.value_data.v.i = -1;
    err = ProTestFeatElemAdd (&elem);

    elem.elem_id = PRO_E_EXT_DEPTH_FROM;
    elem.value_data.type = -1;
    elem.value_data.v.i = -1;
    err = ProTestFeatElemAdd (&elem);
    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;
    err = 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;
    err = ProTestFeatElemAdd (&elem);

    err = ProMdlCurrentGet (&model);
    if ( err != PRO_TK_NO_ERROR ) return ( err );
    err = ProMdlToModelitem( model, &model_item );
    err = ProSelectionAlloc (p_comp_path, &model_item,
                        &model_sel);

    printf ( "Please select a feature to copy section from \n");
    err = ProSelect ( "feature", -1, NULL, NULL, NULL,
                     &sketch_selection, &n_select );
    if ( n_select <= 0 ) return -1;
    err = ProArrayAlloc ( n_select, sizeof (ProSelection),
                        1, ( ProArray *)&selection_array );
    err = ProSelectionCopy ( sketch_selection[0], &selection_array[0] );

    /*-----------------------------------------------*/
    Create the protrusion using the section of the selected feature
    /*-----------------------------------------------*/

    opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEM;
    err = ProFeatureSketchedCreate ( model_sel, elem.tree,
                                    opts, 1, selection_array,
                                    &feature, &errs );
```
Free up the allocated memory.

```c
err = ProElementFree (&elem.tree);
return (0);
}
```

Example 2: Creating a Sketched Datum Curve

The following code fragment shows the creation of a sketched datum curve using the conventional approach.

```c
FILE    : UgSketchedCurveCreate.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"
```
#include "ProDtmCrv.h"
#define C_PRINT(a) printf ( "%s\n", a);

static ProFileName message_file;

/*---------------------- 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" );

/*---------------------------------------------------------------*
*---------------------------------------------------------------*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TREE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TREE, &pro_e_feature_tree );

/*---------------------------------------------------------------*
*---------------------------------------------------------------*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TYPE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TYPE, &pro_e_feature_type );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_FEAT_CURVE; /* 949 */
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_feature_type, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_feature_type );

/*---------------------------------------------------------------*
*---------------------------------------------------------------*/
C_PRINT( " *** Processing Element PRO_E_CURVE_TYPE *** " );
status = ProElementAlloc ( PRO_E_CURVE_TYPE, &pro_e_curve_type );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_CURVE_TYPE_SKETCHED; /* 0 */
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_curve_type, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_curve_type );

/*---------------------------------------------------------------*
*---------------------------------------------------------------*/
C_PRINT( " *** Processing Element PRO_E_STD_SECTION *** " );
status = ProElementAlloc ( PRO_E_STD_SECTION, &pro_e_std_sec_section );
value_data.type = PRO_VALUE_TYPE_RECTANGULAR;
value_data.v.d = PRO_TRUE;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_sec_section, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_std_sec_section );
status = ProElementAlloc (PRO_E_STD_SECTION, &pro_e_std_section);
status = ProElementAdd (pro_e_feature_tree, NULL, pro_e_std_section);

/*---------------------------------------------------------------*
Not required to populate PRO_E_STD_SEC_METHOD
="/---------------------------------------------------------------*/

/*

status = ProElementAlloc (PRO_E_STD_SEC_METHOD, &pro_e_std_sec_method);
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_SEC_SKETCH ** 25;
status = ProValueAlloc (&value);
status = ProValueDataSet (value, &value_data);
status = ProElementValueSet (pro_e_std_sec_method, value);
status = ProElementAdd (pro_e_std_section, NULL, pro_e_std_sec_method);
*
/*---------------------------------------------------------------*
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);
status = ProElementAdd (pro_e_std_section, NULL, pro_e_std_sec_setup_plane);

sketch_refs = (ProSelection *) calloc (2, sizeof (ProSelection));

="/---------------------------------------------------------------*/

C_PRINT(" *** Processing Element PRO_E_STD_SEC_PLANE *** ");
status = ProMessageDisplay (message_file, "Select Surface for sketch placement");
printf("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);
if (n_select <= 0) return -1;
else
status = ProSelectionCopy( p_select[0], &sketch_refs[0]);
}

status = ProElementAlloc ( PRO_E_STD_SEC_PLANE, &pro_e_std_sec_plane );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_sec_plane, value );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane );

/*---------------------------------------------------------------*
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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_SEC_VIEW_DIR_SIDE_ONE; /* 1 */
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_sec_plane_view_dir, value );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup Plane, NULL,
pro_e_std_sec_plane_view_dir );

/*---------------------------------------------------------------*
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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_SEC_ORIENT_DIR_UP; /* 1 */
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_sec_plane_orient_dir, value );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup Plane, NULL,
pro_e_std_sec_plane_orient_dir );

/*---------------------------------------------------------------*
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");
printf ( "Please select datum,surface,slidface,qltface_ID_5 type of ModelItem\n");
status = ProSelect ( "datum,surface,slidface,qltface", 1, NULL,
NULL, NULL, NULL, &p_select, &n_select );
if ( n_select <= 0 ) return -1;
else
{
  status = ProSelectionCopy( p_select[0], &sketch_refs[1]);
}
status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_ORIENT_REF,
&pro_e_std_sec_plane_orient_ref );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_sec_plane_orient_ref, value );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_orient_ref );

/*---------------------------------------------------------------*
 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);
/* Using the element tree from created feature */
status = ProFeatureElemtreeCreate ( &feature, &created_elemtree );

/*---------------------------------------------------------------*
 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);
status = ProElementValueGet ( sketch_element, &value);

status = ProValueDataGet (value, &value_data);

section = (ProSection)value_data.v.p;

/*---------------------------------------------------------------*
  Creating a 3-D section
  
  status = UserSectionBuild ( ( ProSection )(value_data.v.p),
    sketch_refs );
*/

/*---------------------------------------------------------------------*/
  Redefining the feature to make it complete.
/*---------------------------------------------------------------------*/
  opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMS;
  status = ProSelectionAsmcomppathGet (model_sel, &comp_path);

  status = ProFeatureRedefine (&comp_path, &feature, created_elmtree,
    opts, 1, &errors);

/*---------------------------------------------------------------*/
  Free up the allocated memory.
/*---------------------------------------------------------------*/
  status = ProElementFree (&created_elmtree );

  status = ProElementFree (&pro_e_feature_tree );

  return (status);
}
Example 3: Creating Sweep Protrusion Feature

The following code shows how to create a Sweep Protrusion Feature for sketched features using the conventional approach. The procedure is as follows:

- Call function `ProFeatureCreate()` to create an incomplete feature.
- Extract the Spine section handle.
- Add a Spine section (Trajectory).
- Call the function `ProFeatureRedefine()`.
- Extract the Sweep section handle to build the Sweep section.
- Call the function `ProFeatureRedefine()` to complete the feature.

You are prompted to select the sketching and orientation plane. Select two orthogonal edges first for dimensioning the spine (trajectory - PRO_E_SWEEP_SPINE) section and then for the sweep cross section (PRO_E_SWEEP_SECTION).

```c
FILE    : UgSweepCreate.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"
#include "ProSweep.h"

static ProFileName message_file;
static ProSelection * references1;
static ProSelection * references2;

ERROR_CHECK( a, "UgSweepCreate.c", status );

static ProError status;
```
#define C_PRINT(a) printf ( "%s\n", a);

/******************************************************** Function Prototypes **********************************/
ProError UserSweepCreate_local( ProElement );
ProError UserSweepSectionAdd ( ProSection );
ProError UserSweepSpineAdd ( ProSection , ProSelection * );
ProError UserSecerrorPrint ( ProWSecerror *section_errors );

/******************************************************** External Data **********************************/

/******************************************************** Global Data **********************************/

FUNCTION : UgBaseSweepProtrCreate
PURPOSE : Demonstrates the creation of the Simple Sweep Protrusion

ProError UgBaseSweepProtrCreate()
{
    ProElement sketch_element;
    ProElement created_eletmtree;
    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;
    ProValuevalue;
    ProName wide_string;
    ProError status;
    ProValueData value_data;
    ProSelection * p_select;
    int n_select;
    ProBoolean is_interactive = PRO_B_TRUE;
    ProSelection *sketch_refs;
    ProStringToWstring ( message_file, "utilities.txt" );
    references1 = ( ProSelection *) calloc ( 2, sizeof ( ProSelection ) );
references2 = ( 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 );

Populating element PRO_E_FEATURE_TYPE
--------------------------------------*
C_PRINT( " *** Processing Element PRO_E_FEATURE_TYPE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TYPE, &pro_e_feature_type );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = 917;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_feature_type, value );
status = ProElmtreeElementAdd ( pro_e_feature_tree, NULL, pro_e_feature_type );

Populating element PRO_E_FEATURE_FORM
--------------------------------------*
C_PRINT( " *** Processing Element PRO_E_FEATURE_FORM *** " );
status = ProElementAlloc ( PRO_E_FEATURE_FORM, &pro_e_feature_form );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = 4;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_feature_form, value );
status = ProElmtreeElementAdd ( pro_e_feature_tree, NULL, pro_e_feature_form );

Populating element PRO_E_SWEEP_SPINE
-------------------------------------*
C_PRINT( " *** Processing Element PRO_E_SWEEP_SPINE *** " );
status = ProElementAlloc ( PRO_E_SWEEP_SPINE, &pro_e_sweep_spine );
status = ProElmtreeElementAdd ( pro_e_feature_tree, NULL, pro_e_sweep_spine );

Populating element PRO_E_SWEEP_SPINE
-------------------------------------*
C_PRINT( " *** Processing Element PRO_E_SWEEP_SPINE *** " );
status = ProElementAlloc ( PRO_E_SWEEP_SPINE, &pro_e_sweep_spine );
status = ProElmtreeElementAdd ( pro_e_sweep_spine, NULL, pro_e_sweep_spine );

Populating element PRO_E_STD_SEC_SETUP
-> 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 );
status = ProElmtreeElementAdd ( pro_e_std_sec_setup, NULL, pro_e_std_sec_setup );
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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = 25;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_sec_method, value );
status = ProElementtreeElementAdd ( pro_e_std_sec_setup, NULL,
pro_e_std_sec_method );
{
    /* 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 );
    status = ProElementtreeElementAdd ( pro_e_std_sec_setup, NULL,
    pro_e_std_section_plane );
    C_PRINT(" *** Processing Element PRO_E_STD_SEC_PLANE *** ");
    printf( "Please select datum,surface,sldface,qltface type of Modelitem\n");
    status = ProMessageDisplay ( message_file, "Select Surface for sketch placement");
    status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL, NULL, NULL, NULL, &p_select, &n_select );
    if ( n_select <= 0 ) return -1;
    else
    {
        status = ProSelectionCopy ( p_select[0], &references1[0] );
    }

    /*---------------------------------------------------------------*/

    Populating element PRO_E_STD_SEC_SETUP
-> PRO_E_STD_SEC_PLANE
    /*---------------------------------------------------------------*/
    C_PRINT(" *** Processing Element PRO_E_STD_SEC_PLANE *** ");
    status = ProElementAlloc ( PRO_E_STD_SEC_PLANE, &pro_e_std_sec_plane );
    value_data.type = PRO_VALUE_TYPE_SELECTION;
    value_data.v.r = p_select[0];
    status = ProValueAlloc ( &value );
    status = ProValueDataSet ( value, &value_data );

    ProTOOLKIT User's Guide
status = ProElementValueSet ( pro_e_std_sec_plane, value );
status = ProElementTreeAdd ( pro_e_std_section_plane, NULL, pro_e_std_sec_plane );
{
    /*---------------------------------------------------------------------*/
    // Populating element PRO_E_STD_SEC_SETUP
    /*---------------------------------------------------------------------*/
    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 );
    value_data.type = PRO_VALUE_TYPE_INT;
    value_data.v.i = 1;
    status = ProValueAlloc ( &value );
    status = ProValueDataSet ( value, &value_data );
    status = ProElementValueSet ( pro_e_std_sec_plane_view_dir, value );
    status = ProElementTreeAdd ( pro_e_std_section_plane, NULL, pro_e_std_sec_plane_view_dir );

    /*---------------------------------------------------------------------*/
    // Populating element PRO_E_STD_SEC_SETUP
    /*---------------------------------------------------------------------*/
    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 );
    value_data.type = PRO_VALUE_TYPE_INT;
    value_data.v.i = 1;
    status = ProValueAlloc ( &value );
    status = ProValueDataSet ( value, &value_data );
    status = ProElementValueSet ( pro_e_std_sec_plane_orient_dir, value );
    status = ProElementTreeAdd ( pro_e_std_section_plane, NULL, pro_e_std_sec_plane_orient_dir );

    /*---------------------------------------------------------------------*/
    // Populating element PRO_E_STD_SEC_SETUP
    /*---------------------------------------------------------------------*/
    C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_ORIENT_REF *** " );
    printf ( "Please select datum,surface,sldface,qltface type of Modelitem\n" );
    status = ProMessageDisplay ( message_file, "Select Surface for sketch orientation" );
    status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL, NULL, NULL, NULL, &p_select, &n_select );
    if ( n_select <= 0 ) return -1;
else
{
    status = ProSelectionCopy ( p_select[0], &references1[1]);
}

    status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_ORIENT_REF,
&pro_e_std_sec_plane_orient_ref );
    value_data.type = PRO_VALUE_TYPE_SELECTION;
    value_data.v.r = p_select[0];
    status = ProValueAlloc ( &value );
    status = ProValueDataSet ( value, &value_data );
    status = ProElementValueSet ( pro_e_std_sec_plane_orient_ref, value );
    status = ProElemtreeElementAdd ( pro_e_std_section_plane, NULL,
pro_e_std_sec_plane_orient_ref );
}

/*---------------------------------------------------------------*\
   Populating element PRO_E_SWEEP_SECTION

   ---------------------------------------------------------------*/
C_PRINT( " *** Processing Element PRO_E_SWEEP_SECTION *** " );
    status = ProElementAlloc ( PRO_E_SWEEP_SECTION, &pro_e_sweep_section );
    status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_sweep_section );

C_PRINT( " *** Processing Element PRO_E_STD_SEC.Setup *** " );
    status = ProElementAlloc ( PRO_E_STD_SEC_SETUP, &pro_e_std_sec_setup );
    status = ProElemtreeElementAdd ( pro_e_sweep_section, NULL,
pro_e_std_sec_setup );
    status = UserSweepCreate_local ( pro_e_feature_tree );

return status;
}

ProError  UserSweepCreate_local ( ProElement elem_tree )
{
   ProMdl model_current;
   ProModelitem model_item;
   ProSelection model_selection;
   ProSelection *sketch_selection;
   int n_select;
   ProFeature created_feature;
   ProErrorlistfeat_errors;
   ProElementcreated_elemtree;
   ProElementcreated_elemtree_after1st;
   ProBoolean code_first = PRO_B_TRUE;
   ProBoolean code_second = PRO_B_TRUE;
   ProBoolean is_incomplete = PRO_B_TRUE;
   ProValue value;
   ProValueData value_data;
ProFeatureCreateOptions create_options[] =
{ PRO_FEAT_CR_INCOMPLETE_FEAT};
ProFeatureCreateOptions redefine_options[] =
{ PRO_FEAT_CR_DEFINE_MISS_ELEMS};

status = ProMdlCurrentGet( &model_current );
status = ProMdlToModelitem( model_current, &model_item );
status = ProSelectionAlloc( NULL, &model_item, &model_selection );
status = ProFeatureCreate ( model_selection, elem_tree,
create_options, 1, &created_feature, &feat_errors );

if ( status != PRO_TK_NO_ERROR )
return status;

status = ProFeatureElemtreeCreate ( &created_feature,
&created_elemtree);

if ( code_first == PRO_B_TRUE )
{
ProElempath element_path;
ProElempathItem element_path_item[2];
ProElement sketch_element;

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 );
status = ProElempathDataSet ( element_path, element_path_item, 2 );
status = ProElemtreeElementGet ( created_elemtree,
    element_path, &sketch_element );
status = ProElementValueGet ( sketch_element, &value );
status = ProValueDataGet ( value, &value_data );

/* Adding Spine 1st section */
status = UserSweepSpineAdd ( ( ProSection ) (value_data.v.p),
( ProSelection * )references1);

status = ProFeatureRedefine ( NULL, &created_feature,
created_elemtree, redefine_options, 1, &feat_errors );
status = ProFeatureElemtreeCreate ( &created_feature,
&created_elemtree_after1st);
if ( code_second == PRO_B_TRUE )
{
    ProElempath element_path;
    ProElempathItem element_path_item[2];
    ProElement sketch_element;
    ProBoolean is_2d;

    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 = ProElempathAlloc ( &element_path );
    status = ProElempathDataSet ( element_path, element_path_item, 2 );
    status = ProElemtreeElementGet ( created_elemtree_after1st,
                             element_path, &sketch_element );
    status = ProElementValueGet ( sketch_element, &value );
    status = ProValueDataGet ( value, &value_data );
    /* Adding C/S  2nd section */
    status = UserSweepSectionAdd ( value_data.v.p );
}

status = ProFeatureRedefine ( NULL, &created_feature,
                             created_elemtree_after1st,
                             redefine_options,
                             1, &feat_errors );
status = ProElementFree ( &created_elemtree_after1st);

status = ProFeatureElemtreeCreate ( &created_feature,
                                    &created_elemtree_after1st);

status = ProFeatureIsIncomplete( &created_feature, &is_incomplete );
/* Freeing memory */
status = ProElementFree ( &created_elemtree);

status = ProElementFree ( &created_elemtree_after1st);

return status;
}
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 ‘Principles of Feature Creation’ and ‘Creating Sketched Features’ before referring to this chapter.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Element Tree for Extruded Features</td>
<td>25 - 2</td>
</tr>
<tr>
<td>The Element Tree for Revolved Features</td>
<td>25 - 58</td>
</tr>
<tr>
<td>The Element Tree for First Features</td>
<td>25 - 90</td>
</tr>
</tbody>
</table>
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

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.
The following figure shows the element tree for extruded features.

Figure 25-1: 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_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_STD_EXT_DEPTH
    - PRO_E_EXT_DEPTH_FROM
      - PRO_E_EXT_DEPTH_FROM_TYPE
      - PRO_E_EXT_DEPTH_FROM_REF
      - PRO_E_EXT_DEPTH_FROM_VALUE
    - PRO_E_EXT_DEPTH_TO
      - PRO_E_EXT_DEPTH_TO_TYPE
      - PRO_E_EXT_DEPTH_TO_REF
      - PRO_E_EXT_DEPTH_TO_VALUE

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:

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_FEATURE_TYPE</td>
<td>Feature type, not required for creation: PRO_FEAT_PROTRUSION, PRO_FEAT_CUT, PRO_FEAT_DATUM_SURFN</td>
</tr>
<tr>
<td>PRO_E_FEATURE_FORM</td>
<td>Mandatory = PRO_EXTRUDE</td>
</tr>
<tr>
<td>PRO_E_EXT_SURF_CUT_SOLID_TYPE</td>
<td>Mandatory Of type ProExtFeatType = PRO_EXT_FEAT_TYPE_SOLID for Solid feature type, = PRO_EXT_FEAT_TYPE_SURFACE for Surface feature type</td>
</tr>
<tr>
<td>PRO_E_FEAT_FORM_IS_THIN</td>
<td>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</td>
</tr>
<tr>
<td>PRO_E_REMOVE_MATERIAL</td>
<td>Material Removal Of type ProExtRemMaterial = PRO_EXT_MATERIAL_ADD for a Protruded feature, = PRO_EXT_MATERIAL_REMOVE for a Cut feature</td>
</tr>
<tr>
<td>PRO_E_STD_SECTION</td>
<td>Standard section elements</td>
</tr>
<tr>
<td>PRO_E_STD_DIRECTION*</td>
<td>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</td>
</tr>
<tr>
<td>PRO_E_STD_MATRLSIDE*</td>
<td>Direction of material affected with respect to the sketch. Required for all cuts, all thin features, and for solid protrusions with open sections.</td>
</tr>
<tr>
<td>PRO_E_STD_EXT_DEPTH</td>
<td>Compound Element</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_EXT_DEPTH_TO</td>
<td>Compound Element</td>
</tr>
<tr>
<td>PRO_E_EXT_DEPTH_TO_TYPE</td>
<td>Mandatory Of type ProExtDepthToType</td>
</tr>
<tr>
<td>PRO_E_EXT_DEPTH_TO_VALUE</td>
<td>Depends on PRO_E_EXT_DEPTH_TO_TYPE. Of type PRO_VALUE_TYPE_DOUBLE</td>
</tr>
<tr>
<td>PRO_E_EXT_DEPTH_TO_REF</td>
<td>Depends on PRO_E_EXT_DEPTH_TO_TYPE. Of type listed in the Depth Type table that follows.</td>
</tr>
<tr>
<td>PRO_E_EXT_DEPTH_FROM</td>
<td>Compound Element</td>
</tr>
<tr>
<td>PRO_E_EXT_DEPTH_FROM_TYPE</td>
<td>Mandatory Of type ProExtDepthFromType</td>
</tr>
<tr>
<td>PRO_E_EXT_DEPTH_FROM_VALUE</td>
<td>Depends on PRO_E_EXT_DEPTH_FROM_TYPE. Of type PRO_VALUE_TYPE_DOUBLE</td>
</tr>
<tr>
<td>PRO_E_EXT_DEPTH_FROM_REF</td>
<td>Depends on PRO_E_EXT_DEPTH_FROM_TYPE. Of type listed in the Depth Type table that follows.</td>
</tr>
<tr>
<td>PRO_E_STD_FEATURE_NAME</td>
<td>Default given by application depending on the feature type. Can be modified by the user.</td>
</tr>
</tbody>
</table>
The following table lists the Depth Types for extruded features along with possible valid references:

<table>
<thead>
<tr>
<th>Depth Type</th>
<th>Valid Reference Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_EXT_DEPTH_TO_REF</td>
<td>PRO_SURFACE, PRO_AXIS, PRO_EDGE, PRO_CURVE, PRO_POINT, PRO_EDGE_START, PRO_EDGE_END, PRO_CRV_START and PRO_CRV_END.</td>
</tr>
<tr>
<td>PRO_EXT_DEPTH_FROM_REF</td>
<td>PRO_SURFACE, PRO_AXIS, PRO_EDGE, PRO_CURVE, PRO_POINT, PRO_EDGE_START, PRO_EDGE_END, PRO_CRV_START and PRO_CRV_END.</td>
</tr>
</tbody>
</table>

The following table lists the elements needed to create extruded features, in addition to those already discussed:

<table>
<thead>
<tr>
<th>Feature Type</th>
<th>Element ID</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solid</td>
<td>No Additional Elements Required</td>
<td></td>
</tr>
<tr>
<td>Thin</td>
<td>PRO_E_THICKNESS</td>
<td>Mandatory &gt;= 0.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Of type PRO_VALUE_TYPE_DOUBLE</td>
</tr>
<tr>
<td>Solid Cut</td>
<td>PRO_E_STD_MATRLSIDE</td>
<td>Mandatory</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Of type ProExtMatlSide</td>
</tr>
<tr>
<td>Thin Cut</td>
<td>PRO_E_STD_MATRLSIDE</td>
<td>Mandatory</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Of type ProExtMatlSide</td>
</tr>
<tr>
<td></td>
<td>PRO_E_THICKNESS</td>
<td>Mandatory &gt;= 0.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Of type PRO_VALUE_TYPE_DOUBLE</td>
</tr>
<tr>
<td>Surface</td>
<td>PRO_E_SRF_END_ATTRIBUTES</td>
<td>Mandatory ProExtSurfEndAttr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>It must be assigned at the same time or after the section is fully completed.</td>
</tr>
</tbody>
</table>
## 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_apps/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

<table>
<thead>
<tr>
<th>Feature Type</th>
<th>Element ID</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Surface Trim</td>
<td>PRO_E_STD_MATRLSIDE</td>
<td>Mandatory Of type ProExtMatrlSide</td>
</tr>
<tr>
<td></td>
<td>PRO_E_TRIM_QUILT</td>
<td>Mandatory Of type Quilt</td>
</tr>
<tr>
<td></td>
<td>PRO_E_TRIM_QLT_SIDE</td>
<td>Mandatory Of type ProExtTrimQltSide</td>
</tr>
<tr>
<td></td>
<td>PRO_E_SRF_END_ATTRIBUTES</td>
<td>Mandatory Of type ProExtSurfEndAttr</td>
</tr>
<tr>
<td>Thin Surface Trim</td>
<td>PRO_E_STD_MATRLSIDE</td>
<td>Mandatory Of type ProExtMatrlSide</td>
</tr>
<tr>
<td></td>
<td>PRO_E_THICKNESS</td>
<td>Mandatory &gt;= 0.0 Of type PRO_VALUE_TYPE_DOUBLE</td>
</tr>
<tr>
<td></td>
<td>PRO_E_TRIM_QUILT</td>
<td>Mandatory Of type Quilt</td>
</tr>
<tr>
<td></td>
<td>PRO_E_TRIM_QLT_SIDE</td>
<td>Mandatory Of type ProExtTrimQltSide if PRO_E_STD_MATRLSIDE is “both”. Must be assigned at the same time as PRO_E_STD_MATRLSIDE.</td>
</tr>
<tr>
<td></td>
<td>PRO_E_SRF_END_ATTRIBUTES</td>
<td>Mandatory Of type ProExtSurfEndAttr</td>
</tr>
</tbody>
</table>
Conventional Approach

Example 1: Creating an Extruded Feature

The example shows how to create an 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:

```c
#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;

ERROR_CHECK( a, "UgSktExtrusionCreate.c", status );

#define C_PRINT(a) printf ( "%s\n", a);

/*===============================================================================*
FUNCTION : UserSktExtrusionProtrusion
PURPOSE  : Demonstrates the creation of the extruded protrusion base feature.
*===============================================================================*/
ProError UserSktExtrusionProtrusion()
{
    ProErrorlist errors;
    ProMdl model;
    ProModelitem model_item;
    ProSelection model_sel;
    ProFeature feature;
```
Creating Extruded and Revolved Features

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_elmtree;

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 = fopen ( "ug_sketched_curve.log", "w" );

/*-----------------------------------------------*/
Popping root element PRO_E_FEATURE_TREE
/*-----------------------------------------------*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TREE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TREE, &pro_e_feature_tree );
Populating element PRO_E_FEATURE_FORM

C_PRINT( " *** Processing Element PRO_E_FEATURE_FORM *** " );
status = ProElementAlloc ( PRO_E_FEATURE_FORM, &pro_e_feature_form );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXTRUDE;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_feature_form, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL, pro_e_feature_form );

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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_FEAT_TYPE_SOLID;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_ext_surf_cut_solid_type, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL, pro_e_ext_surf_cut_solid_type );

Populating element PRO_E_REMOVE_MATERIAL

C_PRINT( " *** Processing Element PRO_E_REMOVE_MATERIAL *** " );
status = ProElementAlloc ( PRO_E_REMOVE_MATERIAL, &pro_e_remove_material );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_MATERIAL_ADD;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_remove_material, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL, pro_e_remove_material );

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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_FEAT_FORM_NO_THIN;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_feat_form_is_thin, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL, pro_e_feat_form_is_thin );

/*---------------------------------------------------------------*
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 );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL, pro_e_std_ext_depth );

/*---------------------------------------------------------------*
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 );
status = ProElemtreeElementAdd ( pro_e_std_ext_depth, NULL, pro_e_ext_depth_from );

/*---------------------------------------------------------------*
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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_DEPTH_FROM_BLIND;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_ext_depth_from_type, value );
status = ProElemtreeElementAdd ( pro_e_ext_depth_from, NULL, pro_e_ext_depth_from_type );

/*---------------------------------------------------------------*
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 );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 120.000000;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_ext_depth_from_value, value );
status = ProElemtreeElementAdd ( pro_e_ext_depth_from, NULL,
pro_e_ext_depth_from_value );

/*---------------------------------------------------------------*/
Populating element PRO_E_STD_EXTDEPTH

--PRO_E_EXTDEPTH_TO
/*---------------------------------------------------------------*/
C_PRINT(" *** Processing Element PRO_E_EXTDEPTH_TO *** ");
status = ProElementAlloc ( PRO_E_EXTDEPTH_TO, &pro_e_ext_depth_to );
status = ProElemtreeElementAdd ( pro_e_std_ext_depth, NULL,
pro_e_ext_depth_to );

/*---------------------------------------------------------------*/
Populating element PRO_E_STD_EXTDEPTH

--PRO_E_EXTDEPTH_TO
--PRO_E_EXTDEPTH_TO_TYPE
/*---------------------------------------------------------------*/
C_PRINT(" *** Processing Element PRO_E_EXTDEPTH_TO_TYPE *** ");
status = ProElementAlloc ( PRO_E_EXTDEPTH_TO_TYPE,
&pro_e_ext_depth_to_type );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_DEPTH_SYMMETRIC;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_ext_depth_to_type, value );
status = ProElemtreeElementAdd ( pro_e_ext_depth_to, NULL,
pro_e_ext_depth_to_type );

/*---------------------------------------------------------------*/
Populating element PRO_E_STD_SECTION

/*---------------------------------------------------------------*/
C_PRINT(" *** 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 );

/*---------------------------------------------------------------*/
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 );
status = ProElemtreeElementAdd ( pro_e_std_section, NULL,
pro_e_std_sec_setup_plane );
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");
printf("Please select datum,surface,sldface,qltface_ID_5 type of Modelitem
");
status = ProSelect("datum,surface,sldface,qltface", 1, NULL, NULL, NULL, NULL, &p_select, &n_select);
if (n_select <= 0) return -1;
else{
    status = ProSelectionCopy(p_select[0], &sketch_refs[0]);
}
status = ProElementAlloc (PRO_E_STD_SEC_PLANE, &pro_e_std_sec_plane);
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc (&value);
status = ProValueDataSet (value, &value_data);
status = ProElementValueSet (pro_e_std_sec_plane, value);
status = ProElemtreeElementAdd (pro_e_std_sec_setup_plane, NULL, pro_e_std_sec_plane);

/*---------------------------------------------------------------*/
// Populating element PRO_E_STD_SEC_SETUP_PLANE
-> 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);
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_SEC_VIEW_DIR_SIDE_ONE; /* 1 */
status = ProValueAlloc (&value);
status = ProValueDataSet (value, &value_data);
status = ProElementValueSet (pro_e_std_sec_plane_view_dir, value);
status = ProElemtreeElementAdd (pro_e_std_sec_setup_plane, NULL, pro_e_std_sec_plane_view_dir);

/*---------------------------------------------------------------*/
// Populating element PRO_E_STD_SEC_SETUP_PLANE
-> 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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_SEC_ORIENT_DIR_UP; /* 1 */
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_sec_plane_orient_dir, value );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_orient_dir );

/*---------------------------------------------------------------*
Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
"---------------------------------------------------------------*/
C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_ORIENT_REF *** " );
status = ProMessageDisplay ( message_file,
"Select Surface for sketch orientation";
printf ( "Select datum,surface,sldface,qltface_ID_5 type of
Modelitem\n");
status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL,
NULL, NULL, &p_select, &n_select );
if ( n_select <= 0 ) return -1;
else
{
    status = ProSelectionCopy( p_select[0], &sketch_refs[1]);
}
status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_ORIENT_REF,
&pro_e_std_sec_plane_orient_ref );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_sec_plane_orient_ref, value );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_orient_ref );

/*---------------------------------------------------------------*
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);

/* Using the element tree from created feature */
status = ProFeatureElemtreeCreate ( &feature, &created_elemtree );

/*---------------------------------------------------------------*/
/* 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);
status = ProElementValueGet ( sketch_element, &value);

status = 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);
status = ProElementValueGet ( pro_e_std_matrlside, &value );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_MATERIAL_SIDE_ONE;
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_matrlside, value );

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);

status = ProElementValueGet ( pro_e_std_direction, &value );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_CR_IN_SIDE_TWO;
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_direction, value );

ProElempathFree (&path);

/*---------------------------------------------------------------*\
   Redefining the feature to make it complete.
\*------------------------------------------------------------------*/
 opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMS;
status = ProSelectionAsmcompPathGet (model_sel, &comp_path);
status = ProFeatureRedefine (&comp_path, &feature, created_elemtree,
 opts, 1, &errors);

/*---------------------------------------------------------------*\
   Free up the allocated memory.
\*------------------------------------------------------------------*/
status = ProElementFree (&created_elemtree );
status = ProElementFree (&pro_e_feature_tree );

free(sketch_refs);

return (status);
} #undef C_PRINT
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.

Creating Extruded and Revolved Features 25 - 17
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_elmtree;

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 = fopen ( "ug_sketched_curve.log", "w" );

status = ProMdlCurrentGet (&model);
if ( status != PRO_TK_NO_ERROR ) return ( status );

/*----------------------------------------*/
   Populating root element PRO_E_FEATURE_TREE
Creating Extruded and Revolved Features

Creating Extruded and Revolved Features

```c
C_PRINT( " *** Processing Element PRO_E_FEATURE_TREE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TREE, &pro_e_feature_tree );

C_PRINT( " *** Processing Element PRO_E_FEATURE_FORM *** ");
status = ProElementAlloc ( PRO_E_FEATURE_FORM, &pro_e_feature_form );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXTRUDE;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_feature_form, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                  pro_e_feature_form );

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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_FEAT_TYPE_SOLID;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_ext_surf_cut_solid_type, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                 pro_e_ext_surf_cut_solid_type );

C_PRINT( " *** Processing Element PRO_E_REMOVE_MATERIAL *** ");
status = ProElementAlloc ( PRO_E_REMOVE_MATERIAL, &pro_e_remove_material );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_MATERIAL_REMOVE;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_remove_material, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                 pro_e_remove_material );
```

Creating Extruded and Revolved Features
```c
C_PRINT( " *** Processing Element PRO_E_FEAT_FORM_IS_THIN *** " );
status = ProElementAlloc ( PRO_E_FEAT_FORM_IS_THIN,
    &pro_e_feat_form_is_thin );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_FEAT_FORM_NO_THIN;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_feat_form_is_thin, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
    pro_e_feat_form_is_thin );

/*---------------------------------------------------------------*/
Populating compound element PRO_E_STD_EXT_DEPTH
="/**** Processing Element PRO_E_STD_EXT_DEPTH *** " */
status = ProElementAlloc ( PRO_E_STD_EXT_DEPTH, &pro_e_std_ext_depth );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
    pro_e_std_ext_depth );

;/*-----------------------------*/
Populating element PRO_E_STD_EXT_DEPTH
-> PRO_E_EXT_DEPTH_FROM
="/**** Processing Element PRO_E_EXT_DEPTH_FROM *** " */
status = ProElementAlloc ( PRO_E_EXT_DEPTH_FROM, &pro_e_ext_depth_from );
status = ProElemtreeElementAdd ( pro_e_std_ext_depth, NULL,
    pro_e_ext_depth_from );

;/*-----------------------------*/
Populating element PRO_E_STD_EXT_DEPTH
-> PRO_E_EXT_DEPTH_FROM
--> PRO_E_EXT_DEPTH_FROM_TYPE
="/**** Processing Element PRO_E_EXT_DEPTH_FROM_TYPE *** " */
status = ProElementAlloc ( PRO_E_EXT_DEPTH_FROM_TYPE, &pro_e_ext_depth_from_type );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_DEPTH_FROM_ALL;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_ext_depth_from_type, value );
status = ProElemtreeElementAdd ( pro_e_ext_depth_from, NULL,
    pro_e_ext_depth_from_type );

;/*-----------------------------*/
Populating element PRO_E_STD_EXT_DEPTH
-> PRO_E_EXT_DEPTH_TO
="/**** Processing Element PRO_E_EXT_DEPTH_TO *** " */
```
Creating Extruded and Revolved Features

C_PRINT( " *** Processing Element PRO_E_EXT_DEPTH_TO *** " );
status = ProElementAlloc ( PRO_E_EXT_DEPTH_TO, &pro_e_ext_depth_to );
status = ProElemtreeElementAdd ( pro_e_std_ext_depth, NULL,
pro_e_ext_depth_to );

/*----------------------------------------*/
-> 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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_DEPTH_TO_ALL;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_ext_depth_to_type, value );
status = ProElemtreeElementAdd ( pro_e_ext_depth_to, NULL,
pro_e_ext_depth_to_type );

/*----------------------------------------*/
-> Populating element PRO_E_STD_SECTION
/*----------------------------------------*/
C_PRINT( " *** 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 );

/*----------------------------------------*/
-> 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 );
status = ProElemtreeElementAdd ( pro_e_std_section, NULL,
pro_e_std_sec_setup_plane );
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");
printf ( "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);
if ( n_select <= 0 ) return -1;
else
{
    status = ProSelectionCopy( p_select[0], &sketch_refs[0]);
}
status = ProElementAlloc ( PRO_E_STD_SEC_PLANE, &pro_e_std_sec_plane );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_sec_plane, value );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane );
/*---------------------------------------------------------------*/
Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
-> PRO_E_STD_SEC_PLANE_VIEW_DIR
"** 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 );
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 );
status = ProElementValueSet ( pro_e_std_sec_plane_view_dir, value );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_view_dir );
/*---------------------------------------------------------------*/
Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
-> PRO_E_STD_SEC_PLANE_ORIENT_DIR
"** 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 );
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 );
status = ProElementValueSet ( pro_e_std_sec_plane_orient_dir, value );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
Creating Extruded and Revolved Features

pro_e_std_sec_plane_orient_dir);

Population element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANES
-> 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");
printf ( "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);
if ( n_select <= 0 ) return -1;
else
    status = ProSelectionCopy( p_select[0], &sketch_refs[1]);

status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_ORIENT_REF,
    &pro_e_std_sec_plane_orient_ref );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_sec_plane_orient_ref, value );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
    pro_e_std_sec_plane_orient_ref );

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);

/* Using the element tree from created feature */
status = ProFeatureElemtreeCreate ( &feature, &created_elemtree );

/* 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 = ProElmtreeElementGet ( created_elemtree, path, &sketch_element);
status = ProElementValueGet ( sketch_element, &value);

status = ProValueDataGet (value, &value_data);
section = (ProSection)value_data.v.p;
/*-----------------------------------------------*/
        Creating a 3-D section
/*-----------------------------------------------*/
status = UserSectionBuild ( (ProSection)value_data.v.p, sketch_refs);
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 = ProElmtreeElementGet ( created_elemtree, path, &pro_e_std_matrlside);
status = ProElementValueGet ( pro_e_std_matrlside, &value );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_MATERIAL_SIDE_TWO;
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_matrlside, value );
ProElempathFree (&path);
/*-----------------------------------------------*/
        Redefining the feature to make it complete.
/*-----------------------------------------------*/
    opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMS;
status = ProSelectionAsmcomppathGet (model_sel, &comp_path);

status = ProFeatureRedefine (&comp_path, &feature, 
created_elementree, opts, 1, &errors);

/*---------------------------------------------------------------*\
  Free up the allocated memory.
*/
status = ProElementFree (&created_elementree );

status = ProElementFree (&pro_e_feature_tree );

free(sketch_refs);

return (status);
}
#undef C_PRINT

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;

ERROR_CHECK(a, "UgSktExtrusionCreate.c", status);
#define C_PRINT(a) printf ( "%s
", a);

FUNCTION : UserSktExtrusionThin()
PURPOSE : Demonstrates the creation of the extruded thin cut.

```c
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 = fopen ( "ug_sketched_curve.log", "w" );

status = ProMdlCurrentGet ( &model );
if ( status != PRO_TK_NO_ERROR ) return ( status );

/*---------------------------------------------------------------*\n| 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 );

/*---------------------------------------------------------------*\n| Populating element PRO_E_FEATURE_FORM                          |
\*---------------------------------------------------------------*/
C_PRINT(" *** Processing Element PRO_E_FEATURE_FORM *** ");
status = ProElementAlloc ( PRO_E_FEATURE_FORM, &pro_e_feature_form );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXTRUDE;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_feature_form, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_feature_form );

/*---------------------------------------------------------------*\n| 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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_FEAT_TYPE_SOLID;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_ext_surf_cut_solid_type, value );
status = ProElemtreeElementAdd (pro_e_feature_tree, NULL, pro_e_ext_surf_cut_solid_type);

/*---------------------------------------------------------------*
| Populating element PRO_E_REMOVE_MATERIAL                      |
/*---------------------------------------------------------------*/
C_PRINT( " *** Processing Element PRO_E_REMOVE_MATERIAL *** " );
status = ProElementAlloc (PRO_E_REMOVE_MATERIAL, &pro_e_remove_material);
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_MATERIAL_REMOVE;
status = ProValueAlloc (&value);
status = ProValueDataSet (value, &value_data);
status = ProElementValueSet (pro_e_remove_material, value);
status = ProElemtreeElementAdd (pro_e_feature_tree, NULL, pro_e_remove_material);

/*---------------------------------------------------------------*
| 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);
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_FEAT_FORM_THIN;
status = ProValueAlloc (&value);
status = ProValueDataSet (value, &value_data);
status = ProElementValueSet (pro_e_feat_form_is_thin, value);
status = ProElemtreeElementAdd (pro_e_feature_tree, NULL, pro_e_feat_form_is_thin);

/*---------------------------------------------------------------*
| Populating element PRO_E_THICKNESS (thin only)                |
/*---------------------------------------------------------------*/
C_PRINT( " *** Processing Element PRO_E_THICKNESS *** " );
status = ProElementAlloc (PRO_E_THICKNESS, &pro_e_thickness);
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 0.50;
status = ProValueAlloc (&value);
status = ProValueDataSet (value, &value_data);
status = ProElementValueSet (pro_e_thickness, value);
status = ProElemtreeElementAdd (pro_e_feature_tree, NULL, pro_e_thickness);

/*---------------------------------------------------------------*
| Populating compound element PRO_E_STD_EXT_DEPTH              |
/*---------------------------------------------------------------*/
C_PRINT( " *** Processing Element PRO_E_STD_EXT_DEPTH *** " );
Creating Extruded and Revolved Features

```c
status = ProElementAlloc ( PRO_E_STD_EXT_DEPTH, &pro_e_std_ext_depth );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_std_ext_depth );

/* ------------------------------- */
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 );
status = ProElemtreeElementAdd ( pro_e_std_ext_depth, NULL,
pro_e_ext_depth_from );

/* ------------------------------- */
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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_DEPTH_FROM_REF;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElemtreeElementAdd ( pro_e_ext_depth_from, NULL,
pro_e_ext_depth_from_type );

/* ------------------------------- */
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 );
value_data.type = PRO_VALUE_TYPE_SELECTION;
UserUtilItemSelect ("surface,sldface,qltface,datum,axis,edge,curve,edge_end,curve_end",
"USER Select reference.", &from_ref_id, &from_ref_type);
status = ProModelitemInit ( model, from_ref_id, from_ref_type,
&from_ref_item);
```
status = ProSelectionAlloc (NULL, &from_ref_item, &value_data.v.r);
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_ext_depth_from_ref, value );
status = ProElemtreeElementAdd ( pro_e_ext_depth_from, NULL, pro_e_ext_depth_from_ref );

/*---------------------------------------------------------------*
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 );
status = ProElemtreeElementAdd ( pro_e_std_ext_depth, NULL, pro_e_ext_depth_to );
/*---------------------------------------------------------------*
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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_DEPTH_TO_REF;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_ext_depth_to_type, value );
status = ProElemtreeElementAdd ( pro_e_ext_depth_to, NULL, pro_e_ext_depth_to_type );

/*---------------------------------------------------------------*
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 );
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, &to_ref_type);
status = ProModelitemInit (model, to_ref_id, to_ref_type, &to_ref_item);

status = ProSelectionAlloc (NULL, &to_ref_item, &value_data.v.r);

status = ProValueAlloc (&value);
status = ProValueDataSet (value, &value_data);
status = ProElementValueSet (pro_e_ext_depth_to_ref, value);
status = ProElemtreeElementAdd (pro_e_ext_depth_to, NULL, pro_e_ext_depth_to_ref);

/*---------------------------------------------------------------*
* Populating 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);

/*---------------------------------------------------------------*
* Populating element PRO_E_STD_SECTION
--> PRO_E_STD_SEC_SETUP_PLANE
="/-----------------------------------------------*/

status = ProElementAlloc (PRO_E_STD_SEC_SETUP_PLANE, &pro_e_std_sec_setup_plane);
status = ProElemtreeElementAdd (pro_e_std_section, NULL, pro_e_std_sec_setup_plane);

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 = ProMessageDisplay (message_file, "Select Surface for sketch placement");
printf("Please select datum,surface,sldface,qltface_ID_5 type of Modelitem\n");
status = ProSelect ("datum,surface,sldface,qltface", 1, NULL, NULL, &p_select, &n_select);
if (n_select <= 0) return -1;
else {
    status = ProSelectionCopy(p_select[0], &sketch_refs[0]);
}
status = ProElementAlloc (PRO_E_STD_SEC_PLANE, &pro_e_std_sec_plane);
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_sec_plane, value );
status = ProElementTreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane );

/*---------------------------------------------*/
  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 );
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 );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_sec_plane_view_dir, value );
status = ProElementTreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_view_dir );

/*---------------------------------------------*/
  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 );
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 );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_sec_plane_orient_dir, value );
status = ProElementTreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_orient_dir );

/*---------------------------------------------*/
  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";
printf ( "Select datum,surface,sldface,qltface_ID_5 type of Modelitem\n");
status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL, NULL, NULL, &p_select, &n_select );
if ( n_select <= 0 ) return -1;
else
{
    status = ProSelectionCopy( p_select[0], &sketch_refs[1]);
}

status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_ORIENT_REF,
&pro_e_std_sec_plane_orient_ref );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_sec_plane_orient_ref, value );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_orient_ref );
/*------------------------------------------------------------------------*/
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);

/* Using the element tree from created feature */
status = ProFeatureElemtreeCreate ( &feature, &created_elemtree );
/*------------------------------------------------------------------------*/
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 );

status = ProElementValueGet ( sketch_element, &value );
status = ProValueDataGet (value, &value_data);
[section = (ProSection)value_data.v.p;

/*---------------------------------------------------------------*\
Creating a 3-D section
\*------------------------------------------------------------------*/

status = UserSectionBuild (( ProSection )(value_data.v.p),
    sketch_refs);

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);

status = ProElementValueGet ( pro_e_std_matrlside, &value );

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_MATERIAL_BOTH_SIDES;
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_matrlside, value );

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);

status = ProElementValueGet ( pro_e_std_direction, &value );

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_CR_IN_SIDE_ONE;
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_direction, value );

ProElempathFree (&path);
/*---------------------------------------------------------------*
 Redefining the feature to make it complete.
\*---------------------------------------------------------------*/
 opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMES;
 status = ProSelectionAsmcomppathGet (model_sel, &comp_path);

 status = ProFeatureRedefine (&comp_path, &feature, created_elemtree,
 opts, 1, &errors);

/*---------------------------------------------------------------*
 Free up the allocated memory.
\*---------------------------------------------------------------*/
 status = ProElementFree (&created_elemtree );
 status = ProElementFree (&pro_e_feature_tree );
 free(sketch_refs);
 return (status);
}
#endif C_PRINT

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.

รู(95,727,948,740)
#include "ProExtrude.h"

static ProFileName message_file;

ERROR_CHECK( a, "UgSktExtrusionCreate.c", status );

#define C_PRINT(a) printf ( "%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_elementree;

    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;
Creating Extruded and Revolved Features

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 = fopen ( "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 );

/*---------------------------------------------------------------*
| Populating element PRO_E_FEATURE_FORM
/*---------------------------------------------------------------*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_FORM *** " );
status = ProElementAlloc ( PRO_E_FEATURE_FORM, &pro_e_feature_form );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXTRUDE;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_feature_form, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL, pro_e_feature_form );

/*---------------------------------------------------------------*
| 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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_FEAT_TYPE_SURFACE;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_ext_surf_cut_solid_type, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL, pro_e_ext_surf_cut_solid_type );
Populating element PRO_E_REMOVE_MATERIAL

C_PRINT( " *** Processing Element PRO_E_REMOVE_MATERIAL *** " );
status = ProElementAlloc ( PRO_E_REMOVE_MATERIAL, &pro_e_remove_material);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_MATERIAL_ADD;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_remove_material, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_remove_material );

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 );

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_FEAT_FORM_NO_THIN;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_feat_form_is_thin, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_feat_form_is_thin );

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 );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_std_ext_depth );

Populating element PRO_E_STD_EXT_DEPTH

C_PRINT( " *** Processing Element PRO_E_EXT_DEPTH_FROM *** " );
status = ProElementAlloc ( PRO_E_EXT_DEPTH_FROM,
&pro_e_ext_depth_from );
status = ProElemtreeElementAdd ( pro_e_std_ext_depth, NULL,
pro_e_ext_depth_from );

Populating 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 );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_std_ext_depth );
Creating Extruded and Revolved Features

-> 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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_DEPTH_FROM_NONE;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_ext_depth_from_type, value );
status = ProElemtreeElementAdd ( pro_e_ext_depth_from, NULL,
pro_e_ext_depth_from_type );

/*---------------------------------------------------------------*/

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 );
status = ProElemtreeElementAdd ( pro_e_std_ext_depth, NULL,
pro_e_ext_depth_to );

/*---------------------------------------------------------------*/

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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_DEPTH_TO_BLIND;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_ext_depth_to_type, value );
status = ProElemtreeElementAdd ( pro_e_ext_depth_to, NULL,
pro_e_ext_depth_to_type );

/*---------------------------------------------------------------*/

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 );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 75.0;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_ext_depth_to_value, value );
status = ProElemtreeElementAdd ( pro_e_ext_depth_to, NULL,
pro_e_ext_depth_to_value );

/*---------------------------------------------------------------*/
// Populating 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 );

/*---------------------------------------------------------------*/
// Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
/*---------------------------------------------------------------*/
status = ProElementAlloc ( PRO_E_STD_SEC_SETUP_PLANE, NULL,
pro_e_std_sec_setup_plane );
status = ProElemtreeElementAdd ( pro_e_std_section, NULL,
pro_e_std_sec_setup_plane );
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_SETUP_PLANE *** ");
status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL, NULL,
NULL, &p_select, &n_select );
if ( n_select <= 0 ) return -1;
else
{
status = ProSelectionCopy( p_select[0], &sketch_refs[0]);
}
status = ProElementAlloc ( PRO_E_STD_SEC_PLANE, &pro_e_std_sec_plane );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_sec_plane, value );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane );
Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
  -> PRO_E_STD_SEC_PLANE_VIEW_DIR

Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE
  -> PRO_E_STD_SEC_PLANE_ORIENT_DIR

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_VIEW_DIR *** " );
status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_VIEW_DIR,
  &pro_e_std_sec_plan_view_dir );
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 );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_sec_plan_view_dir, value );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
  pro_e_std_sec_plan_view_dir );

C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE_CHAIN_DIR *** " );
status = ProElementAlloc ( PRO_E_STD_SEC_PLAN_CHAIN_DIR,
  &pro_e_std_sec_plan_chain_dir );
value_data.type = PRO_VALUE_TYPE_CSTRING;
value_data.v.s = PRO_CHAIN_DIR_UP /* PRO_CHAIN_DIR_DOWN */;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_sec_plan_chain_dir, value );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
  pro_e_std_sec_plan_chain_dir );

C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLAN_CHAIN_REF *** " );
status = ProMessageDisplay ( message_file,
  "Select Surface for sketch orientation");
printf ( "Select datum,surface,sldface,qltface_ID_5 type of
  Modelitem\n");
status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL,
  NULL, NULL, &p_select, &n_select );
if ( n_select <= 0 ) return -1;
else
status = ProSelectionCopy( p_select[0], &sketch_refs[1]);
}

status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_ORIENT_REF,
&pro_e_std_sec_plane_orient_ref );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_sec_plane_orient_ref, value );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_orient_ref );

/*---------------------------------------------------------------*
| 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);

/* Using the element tree from created feature */
status = ProFeatureElemtreeCreate ( &feature, &created_elemtree );

/*---------------------------------------------------------------*
| 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);

status = ProElementValueGet ( sketch_element, &value);

status = ProValueGetData (value, &value_data);
section = (ProSection)value_data.v.p;

/*---------------------------------------------------------------*
| Creating a 3-D section                                         |
/*---------------------------------------------------------------*/

status = UserSectionBuild ( ( ProSection )(value_data.v.p),
sketch_refs );
Creating Extruded and Revolved Features

ProElempathFree (&path);

/*-----------------------------------------------*/
// Processing 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);

status = ProElementValueGet ( pro_e_std_direction, &value );

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 );
status = ProElementValueSet ( pro_e_std_direction, value );
ProElempathFree (&path);

/*-----------------------------------------------*/
// Processing 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);

status = ProElementValueGet ( pro_e_srf_end_attributes, &value );

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_SURF_ENDATTR_CAPPED;

status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_srf_end_attributes, value );
ProElempathFree (&path);

/*-----------------------------------------------*/
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.

```c
#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 "ProExtrude.h"

#include "ProExtrude.h"

static ProFileName message_file;

ERROR_CHECK( a, "UgSktExtrusionCreate.c", status );
#define C_PRINT(a) printf ( "%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 = fopen ( "ug_sketched_curve.log", "w" );

status = ProMd1CurrentGet (&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_PRINT( " *** Processing Element PRO_E_FEATURE_FORM *** " );
status = ProElementAlloc ( PRO_E_FEATURE_FORM, &pro_e_feature_form );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXTRUDE;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_feature_form, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_feature_form );

/*---------------------------------------------------------------*
| 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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_FEAT_TYPE_SURFACE;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_ext_surf_cut_solid_type, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_ext_surf_cut_solid_type );

/*---------------------------------------------------------------*
| Populating element PRO_E_REMOVE_MATERIAL                         |
/*---------------------------------------------------------------*/
C_PRINT( " *** Processing Element PRO_E_REMOVE_MATERIAL *** " );
status = ProElementAlloc ( PRO_E_REMOVE_MATERIAL, &pro_e_remove_material );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_MATERIAL_REMOVE;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_remove_material, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_remove_material );

/*---------------------------------------------------------------*
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 );
value_data.type = PRO_VALUE_TYPE_SELECTION;
UserUtilItemSelect ("dtmqlt", "USER Select quilt.", &quilt_id,
&quilt_type);
status = ProModelitemInit ( model, quilt_id, quilt_type, &quilt_item);
status = ProSelectionAlloc (NULL, &quilt_item, &value_data.v.r);
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_trim_quilt, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_trim_quilt );

/*---------------------------------------------------------------*
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 );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_std_ext_depth );

/*---------------------------------------------------------------*
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 );
status = ProElemtreeElementAdd ( pro_e_std_ext_depth, NULL,
pro_e_ext_depth_from );

/*---------------------------------------------------------------*/
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_TYPE,
&pro_e_ext_depth_from_type);
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_DEPTH_FROM_NONE;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_ext_depth_from_type, value );
status = ProElemtreeElementAdd ( pro_e_ext_depth_from, NULL,
pro_e_ext_depth_from_type );

/*---------------------------------------------------------------*
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 );
status = ProElemtreeElementAdd ( pro_e_std_ext_depth, NULL,
pro_e_ext_depth_to );

/*---------------------------------------------------------------*
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);
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_DEPTH_TO_BLIND;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_ext_depth_to_type, value );
status = ProElemtreeElementAdd ( pro_e_ext_depth_to, NULL,
pro_e_ext_depth_to_type );

/*---------------------------------------------------------------*
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);
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 40.0;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data);
status = ProElementValueSet ( pro_e_ext_depth_to_value, value );
status = ProElemTreeElementAdd ( pro_e_ext_depth_to, NULL, pro_e_ext_depth_to_value);
/*---------------------------------------------------------------*
Populating element PRO_E_STD_SECTION
="/------------------------------------------------------------------*/
C_PRINT ( " *** 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 );
/*---------------------------------------------------------------*
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 );
status = ProElemTreeElementAdd ( pro_e_std_section, NULL, pro_e_std_sec_setup_plane );
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");
printf ( "Please select datum,surface,sldface,qltface_ID_5 type of Modelitem\n" );
status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL, NULL, NULL, &p_select, &n_select );
if ( n_select <= 0 ) return -1;
else
{    status = ProSelectionCopy( p_select[0], &sketch_refs[0]);
}
status = ProElementAlloc ( PRO_E_STD_SEC_PLANE, &pro_e_std_sec_plan );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_sec_plan, value );
status = ProElemTreeElementAdd ( pro_e_std_sec_setup_plan, NULL, pro_e_std_sec_plan );
Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE

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 );
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 );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_sec_plane_view_dir, value );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_view_dir );

Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE

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 );
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 );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_sec_plane_orient_dir, value );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_orient_dir );

Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANE

C_PRINT(" *** Processing Element PRO_E_STD_SEC_PLANE_ORIENT_REF *** ");
status = ProMessageDisplay ( message_file,
"Select Surface for sketch orientation" );
printf ( "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 );
if ( n_select <= 0 ) return -1;
else
Creating Extruded and Revolved Features

```c
{
    status = ProSelectionCopy( p_select[0], &sketch_refs[1]);
}

status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_ORIENT_REF,
    &pro_e_std_sec_plane_orient_ref );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_sec_plane_orient_ref, value );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
    pro_e_std_sec_plane_orient_ref );

/*-----------------------------------------------*
 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);

/* Using the element tree from created feature */
status = ProFeatureElemtreeCreate ( &feature, &created_elemtree );

/*-----------------------------------------------*
 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);

 status = ProElementValueGet ( sketch_element, &value);

 status = ProValueGetData (value, &value_data);
 section = (ProSection)value_data.v.p;

/*-----------------------------------------------*
 Creating a 3-D section
-----------------------------------------------*/
```
status = UserSectionBuild((ProSection)(value_data.v.p),
    sketch.refs);

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);
status = ProElementValueGet(pro_e_std_matrlside, &value);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_MATERIAL_BOTH_SIDES;
status = ProValueDataSet(value, &value_data);
status = ProElementValueSet(pro_e_std_matrlside, value);

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);
status = ProElementValueGet(pro_e_std_direction, &value);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_CR_IN_SIDE_TWO;
status = ProValueDataSet(value, &value_data);
status = ProElementValueSet(pro_e_std_direction, value);

ProElempathFree(&path);

="/***********************************************************************************/

25 - 52 Pro/TOOLKIT User's Guide
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 = ProElemtreeElementGet (created_elemtree, path,
 &pro_e_trim_qlt_side);
status = ProElementValueGet (pro_e_trim_qlt_side, &value);

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_TRIMQLT_SIDE_ONE;
status = ProValueDataSet (value, &value_data);
status = ProElementValueSet (pro_e_trim_qlt_side, value); 

ProElempathFree (&path);
/*---------------------------------------------------------------*/
Redefining the feature to make it complete.
/*---------------------------------------------------------------*/
opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMS;
status = ProSelectionAsmcomppathGet (model_sel, &comp_path);

status = ProFeatureRedefine (&comp_path, &feature, created_elemtree,
 opts, 1, &errors);
/*---------------------------------------------------------------*/
Free up the allocated memory.
/*---------------------------------------------------------------*/
status = ProElementFree (&created_elemtree);
free(sketch_refs);
return (status);
}
#undef C_PRINT

**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:
#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;

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;
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));
/*---------------------------------------------------------------*/
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);
/*---------------------------------------------------------------*/
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);
/*---------------------------------------------------------------*/
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);
/*---------------------------------------------------------------*/
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 = ProcTestFeatElemAdd (&elem);

/*---------------------------------------------------------------*
 \* 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 = ProcTestFeatElemAdd (&elem);

/*---------------------------------------------------------------*
 \* 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 = ProcTestFeatElemAdd (&elem);

/*---------------------------------------------------------------*
 \* 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 = ProcTestFeatElemAdd (&elem);

elem.elem_id = PRO_E_EXT_DEPTH_FROM;
elem.value_data.type = -1;
elem.value_data.v.i = -1;
status = ProcTestFeatElemAdd (&elem);
pARENT_ELEM = elem.parent_element;
vl.elem_id = PRO_E_EXT_DEPTH_FROM_TYPE;
vl.value_data.type = PRO_VALUE_TYPE_INT;
vl.value_data.v.i = PRO_EXT_DEPTH_FROM_BLIND;
status = ProcTestFeatElemAdd (&vl);
pARENT_ELEM = parent_elem;
vl.elem_id = PRO_E_EXT_DEPTH_FROM_VALUE;
vl.value.data.type = PRO_VALUE_TYPE_DOUBLE;
vl.value_data.v.d = 50.0;
status = ProcTestFeatElemAdd (&vl);

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);
status = ProMessageDisplay ( message_file, "Select a Sketched Datum Curve Feature");
printf ( "Please Select a Sketched Datum Curve Feature 
");
status = ProSelect ( "feature", 1, NULL, NULL, NULL, NULL, &sketch_selection, &n_select );
if ( n_select <= 0 ) return -1;
status = ProArrayAlloc ( n_select, sizeof (ProSelection), 1, ( ProArray *)&selection_array );
status = ProSelectionCopy ( sketch_selection[0], &selection_array[0] );
/*---------------------------------------------------------------*
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 );
/*---------------------------------------------------------------*
Free up the allocated memory.
/*---------------------------------------------------------------*/
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);
    status = ProValueDataSet (value, &elem->value_data);
  }
  status = ProElementAlloc (elem->elem_id, &element);
if (elem->value_data.type != -1)
{
    status = ProElementValueSet (element, value);
}
status = ProElemtreeElementAdd (elem->parent_element, NULL, element);
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.
The following figure shows the element tree for revolved features.

Figure 25-2: 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_MTRLSIDE (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_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

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:

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_FEATURE_TYPE</td>
<td>Feature type:</td>
</tr>
<tr>
<td></td>
<td>PRO_FEAT_PROTRUSION</td>
</tr>
<tr>
<td></td>
<td>PRO_FEAT_CUT</td>
</tr>
<tr>
<td></td>
<td>PRO_FEAT_DATUM_SURF</td>
</tr>
<tr>
<td></td>
<td>Not required for creation.</td>
</tr>
<tr>
<td>PRO_E_FEATURE_FORM</td>
<td>Mandatory= PRO_REVOLVE</td>
</tr>
<tr>
<td>PRO_E_EXT_SURF_CUT_SOLID_TYPE</td>
<td>Mandatory</td>
</tr>
<tr>
<td></td>
<td>Of type ProRevFeatType</td>
</tr>
<tr>
<td></td>
<td>= PRO_REV_FEAT&gt;Type SOLID</td>
</tr>
<tr>
<td></td>
<td>for Solid feature type</td>
</tr>
<tr>
<td></td>
<td>= PRO_REV_FEAT_TYPE_SURFACE</td>
</tr>
<tr>
<td></td>
<td>for Surface feature type</td>
</tr>
<tr>
<td>PRO_E_FEAT_FORM_IS_THIN</td>
<td>Feature Form</td>
</tr>
<tr>
<td></td>
<td>Of Type ProRevFeatForm</td>
</tr>
<tr>
<td></td>
<td>= PRO_REV_FEAT_FORM_NO_THIN</td>
</tr>
<tr>
<td></td>
<td>for a Solid feature</td>
</tr>
<tr>
<td></td>
<td>= PRO_REV_FEAT_FORM_THIN</td>
</tr>
<tr>
<td></td>
<td>for a Thin feature</td>
</tr>
<tr>
<td>PRO_E_REMOVE_MATERIAL</td>
<td>Material Removal</td>
</tr>
<tr>
<td></td>
<td>Of type ProRevRemMaterial</td>
</tr>
<tr>
<td></td>
<td>= PRO_REV_MATERIAL_ADD</td>
</tr>
<tr>
<td></td>
<td>for a Protruded feature</td>
</tr>
<tr>
<td></td>
<td>= PRO_REV_MATERIAL_REMOVE</td>
</tr>
<tr>
<td></td>
<td>for a Cut feature</td>
</tr>
<tr>
<td>PRO_E_STD_SECTION</td>
<td>Standard section elements.</td>
</tr>
<tr>
<td>PRO_E_STD_DIRECTION*</td>
<td>Direction of creation.</td>
</tr>
<tr>
<td></td>
<td>Of type ProRevDirection</td>
</tr>
<tr>
<td></td>
<td>= PRO_REV_CR_IN_SIDE_ONE</td>
</tr>
<tr>
<td></td>
<td>for angle in side one</td>
</tr>
<tr>
<td></td>
<td>= PRO_REV_CR_IN_SIDE_TWO</td>
</tr>
<tr>
<td></td>
<td>for angle in side two</td>
</tr>
<tr>
<td>PRO_E_STD_MATRLSIDE*</td>
<td>Direction of material affected with respect to the sketch. It is</td>
</tr>
<tr>
<td></td>
<td>required for all cuts, all thin features, and for solid</td>
</tr>
<tr>
<td></td>
<td>protrusions with open sections.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_REV_ANGLE</td>
<td>Compound Element</td>
</tr>
<tr>
<td>PRO_E_REV_ANGLE_TO</td>
<td>Compound Element</td>
</tr>
<tr>
<td>PRO_E_REV_ANGLE_TO_TYPE</td>
<td>Mandatory Of type ProRevAngleToType</td>
</tr>
<tr>
<td>PRO_E_REV_ANGLE_TO_VAL</td>
<td>Depends on PRO_E_REV_ANGLE_TO_TYPE Of type PRO_VALUE_TYPE_DOUBLE (in degrees)</td>
</tr>
<tr>
<td>PRO_E_REV_ANGLE_TO_REF</td>
<td>Depends on PRO_E_REV_ANGLE_TO_TYPE Of type listed in the Angle Type table that follows.</td>
</tr>
<tr>
<td>PRO_E_REV_ANGLE_FROM</td>
<td>Compound Element</td>
</tr>
<tr>
<td>PRO_E_REV_ANGLE_FROM_TYPE</td>
<td>Mandatory Of type ProRevAngleFromType</td>
</tr>
<tr>
<td>PRO_E_REV_ANGLE_FROM_VAL</td>
<td>Depends on PRO_E_REV_ANGLE_FROM_TYPE Of type PRO_VALUE_TYPE_DOUBLE (in degrees)</td>
</tr>
<tr>
<td>PRO_E_REV_ANGLE_FROM_REF</td>
<td>Depends on PRO_E_REV_ANGLE_FROM_TYPE Of type listed in the Angle Type table that follows.</td>
</tr>
<tr>
<td>PRO_E_STD_FEATURE_NAME</td>
<td>Default given by application depending on the feature type. Can be modified by the user.</td>
</tr>
</tbody>
</table>
The following table lists the angle types for revolved features along with possible valid references:

<table>
<thead>
<tr>
<th>Angle Type</th>
<th>Valid Reference Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_REV_ANGLE_TO_REF</td>
<td>PRO_POINT, PRO_EDGE_START, PRO_EDGE_END, PRO_CRV_START, PRO_CRV_END, PRO_SURFACE (Plane).</td>
</tr>
<tr>
<td>PRO_REV_ANGLE_FROM_REF</td>
<td>PRO_POINT, PRO_EDGE_START, PRO_EDGE_END, PRO_CRV_START, PRO_CRV_END, PRO_SURFACE (Plane).</td>
</tr>
</tbody>
</table>

The following table lists the elements needed to create revolved features, in addition to those already discussed:

<table>
<thead>
<tr>
<th>Feature Type</th>
<th>Element ID</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solid</td>
<td>No Additional Elements Required</td>
<td></td>
</tr>
<tr>
<td>Thin</td>
<td>PRO_E_THICKNESS</td>
<td>Mandatory ( \geq 0.0 ) of type PRO_VALUE_TYPE_DOUBLE</td>
</tr>
<tr>
<td>Solid Cut</td>
<td>PRO_E_STD_MATRLSIDE</td>
<td>Mandatory of type ProRevMatlSide</td>
</tr>
<tr>
<td>Thin Cut</td>
<td>PRO_E_STD_MATRLSIDE</td>
<td>Mandatory of type ProRevMatlSide</td>
</tr>
<tr>
<td></td>
<td>PRO_E_THICKNESS</td>
<td>Mandatory ( \geq 0.0 ) of type PRO_VALUE_TYPE_DOUBLE</td>
</tr>
<tr>
<td>Surface</td>
<td>PRO_E_SRF_END_ATTRIBUTES</td>
<td>Mandatory of type ProRevSurfEndAttr Must be assigned at the same time or after the section is fully completed.</td>
</tr>
</tbody>
</table>
Creating Extruded and 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

<table>
<thead>
<tr>
<th>Feature Type</th>
<th>Element ID</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Surface Trim</td>
<td>PRO_E_STD_MATRLSIDE</td>
<td>Mandatory Of type ProRevMatlSide</td>
</tr>
<tr>
<td></td>
<td>PRO_E_TRIM_QUILT</td>
<td>Mandatory Of type Quilt</td>
</tr>
<tr>
<td></td>
<td>PRO_E_TRIM(QLT)_SIDE</td>
<td>Mandatory Of type ProRevTrimQltSide if PRO_E_STD_MATRLSIDE is “both”. Must be assigned at the same time as PRO_E_STD_MATRLSIDE.</td>
</tr>
<tr>
<td></td>
<td>PRO_E_SRF_END_ATTRIBUTES</td>
<td>Mandatory Of type ProRevSurfEndAttr</td>
</tr>
<tr>
<td>Thin Surface Trim</td>
<td>PRO_E_STD_MATRLSIDE</td>
<td>Mandatory Of type ProRevMatlSide</td>
</tr>
<tr>
<td></td>
<td>PRO_E_THICKNESS</td>
<td>Mandatory ( \geq 0.0 ) Of type PRO_VALUE_TYPE_DOUBLE</td>
</tr>
<tr>
<td></td>
<td>PRO_E_TRIM_QUILT</td>
<td>Mandatory Of type Quilt</td>
</tr>
<tr>
<td></td>
<td>PRO_E_TRIM(QLT)_SIDE</td>
<td>Mandatory Of type ProRevTrimQltSide</td>
</tr>
<tr>
<td></td>
<td>PRO_E_SRF_END_ATTRIBUTES</td>
<td>Mandatory Of type ProRevSurfEndAttr</td>
</tr>
</tbody>
</table>

**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.

```c
FILE    : UgSktRevolveProtrusion.c
PURPOSE : Creates a revolved protrusion

FILE    : UgSktRevolveProtrusion.c
PURPOSE : Creates a revolved protrusion

#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;

ERROR_CHECK( a, "UgSktRevolveCreate.c", status );

#define C_PRINT(a) printf ( "%s\n", a);

/*---------------------- 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;

ERROR_CHECK( a, "UgSktRevolveCreate.c", status );

#define C_PRINT(a) printf ( "%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_element;

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 = fopen ( "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 );

/*-----------------------------------------------*
   Populating element PRO_E_FEATURE_FORM
-----------------------------------------------*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_FORM *** " );
status = ProElementAlloc ( PRO_E_FEATURE_FORM, &pro_e_feature_form );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REVOLVE;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_feature_form, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL, pro_e_feature_form );

/*---------------------------------------------------------------*/
| 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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_FEAT_TYPE SOLID;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_ext_surf_cut_solid_type, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL, pro_e_ext_surf_cut_solid_type );

/*---------------------------------------------------------------*/
| Populating element PRO_E_REMOVE_MATERIAL
/*---------------------------------------------------------------*/
C_PRINT( " *** Processing Element PRO_E_REMOVE_MATERIAL *** " );
status = ProElementAlloc ( PRO_E_REMOVE_MATERIAL, &pro_e_remove_material );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_MATERIAL_ADD;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_remove_material, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL, pro_e_remove_material );

/*---------------------------------------------------------------*/
| 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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_FEAT_FORM NO_THIN;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
Creating Extruded and Revolved Features

Populating compound element PRO_E_STD_REV_ANGLE

Populating element PRO_E_REV_ANGLE
-> PRO_E_REV_ANGLE_FROM

Populating element PRO_E_REV_ANGLE
-> PRO_E_REV_ANGLE_FROM_TYPE

Populating element PRO_E_REV_ANGLE
-> PRO_E_REV_ANGLE_FROM_VAL

status = ProElementValueSet ( pro_e_feat_form_is_thin, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                 pro_e_feat_form_is_thin );

status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                 pro_e_feat_form_is_thin );

status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                 pro_e_feat_form_is_thin );

status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                 pro_e_feat_form_is_thin );

status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                 pro_e_feat_form_is_thin );

status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                 pro_e_feat_form_is_thin );

status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                 pro_e_feat_form_is_thin );

status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                 pro_e_feat_form_is_thin );

status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                 pro_e_feat_form_is_thin );

status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                 pro_e_feat_form_is_thin );

status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                 pro_e_feat_form_is_thin );

status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                 pro_e_feat_form_is_thin );

value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_ANG_FROM_ANGLE;
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_rev_angle_from_type, value );
status = ProElemtreeElementAdd ( pro_e_rev_angle_from, NULL,
                                 pro_e_rev_angle_from );

value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 120.000000;
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_rev_angle_from_val, value );
status = ProElementAlloc ( PRO_E_REV_ANGLE_TO, &pro_e_rev_angle_to);
status = ProElemtreeElementAdd ( pro_e_rev_angle, NULL,
pro_e_rev_angle_to);
/*---------------------------------------------------------------*/
// Populating element PRO_E_REV_ANGLE
-> PRO_E_REV_ANGLE_TO
/*---------------------------------------------------------------*/
C_PRINT( " *** Processing Element PRO_E_REV_ANGLE *** " );
status = ProElementAlloc ( PRO_E_REV_ANGLE_TO, &pro_e_rev_angle_to );
status = ProElemtreeElementAdd ( pro_e_rev_angle, NULL,
pro_e_rev_angle_to );

/*---------------------------------------------------------------*/
// Populating element PRO_E_REV_ANGLE
-> 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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_ANG_SYMMETRIC;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_rev_angle_to_type, value );
status = ProElemtreeElementAdd ( pro_e_rev_angle_to, NULL,
pro_e_rev_angle_to_type );

/*---------------------------------------------------------------*/
// Populating element PRO_E_STD_SECTION
/*---------------------------------------------------------------*/
C_PRINT( " *** 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 );

/*---------------------------------------------------------------*/
// Populating element 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 );
status = ProElemtreeElementAdd ( pro_e_std_section, NULL,
pro_e_std_sec_setup_plane );

sketch_refs = ( ProSelection *) calloc ( 2, sizeof ( ProSelection ));
/*---------------------------------------------------------------*/
Creating Extruded and Revolved Features

```

Populating element PRO_E_STD_SECTION
  -> PRO_E_STD_SEC_SETUP_PLANE
     -> PRO_E_STD_SEC_PLANE

URRENCY------------------------------------------------------------------------*

C_PRINT( " *** Processing Element PRO_E_STD_SEC_PLANE *** " );
status = ProMessageDisplay ( message_file, "Select Surface for sketch placement");
printf ( "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 );
if ( n_select <= 0 ) return -1;
else
{
  status = ProSelectionCopy( p_select[0], &sketch_refs[0] );
}
status = ProElementAlloc ( PRO_E_STD_SEC_PLANE, &pro_e_std_sec_plane );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_sec_plane, value );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL, pro_e_std_sec_plane );

/*---------------------------------------------------------------*/
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 );
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 );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_sec_plane_view_dir, value );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL, pro_e_std_sec_plane_view_dir );

/*---------------------------------------------------------------*/
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, NULL, pro_e_std_sec_plane_orient_dir );
```

Creating Extruded and Revolved Features 25 - 69
&pro_e_std_sec_plane_orient_dir);
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);
status = ProValueDataSet (value, &value_data);
status = ProElementValueSet (pro_e_std_sec_plane_orient_dir, value);
status = ProElemtreeElementAdd (pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_orient_dir);

/*---------------------------------------------------------------*
| 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");
printf ("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);
if (n_select <= 0) return -1;
else {
    status = ProSelectionCopy(p_select[0], &sketch_refs[1]);
}
status = ProElementAlloc (PRO_E_STD_SEC_PLANE_ORIENT_REF,
&pro_e_std_sec_plane_orient_ref);
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc (&value);
status = ProValueDataSet (value, &value_data);
status = ProElementValueSet (pro_e_std_sec_plane_orient_ref, value);
status = ProElemtreeElementAdd (pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_orient_ref);

/*---------------------------------------------------------------*
| 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);

/* Using the element tree from created feature */
status = ProFeatureElemtreeCreate (&feature, &created_elemtree);

/*---------------------------------------------------------------*
  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);

status = ProElementValueGet ( sketch_element, &value);

status = ProValueDataGet (value, &value_data);
section = (ProSection)value_data.v.p;

/*---------------------------------------------------------------*
  Creating a 3-D section
/*---------------------------------------------------------------*/
  status = UserSectionBuild (( ProSection )(value_data.v.p), sketch_refs);

ProElempathFree (&path);

/*---------------------------------------------------------------*
  Redefining the feature to make it complete.
/*---------------------------------------------------------------*/
  opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMS;
status = ProSelectionAsmcomppathGet (model_sel, &comp_path);

status = ProFeatureRedefine (&comp_path, &feature, created_elemtree, opts, 1, &errors);

/*---------------------------------------------------------------*
  Free up the allocated memory.
/*---------------------------------------------------------------*/
  status = ProElementFree (&created_elemtree);
status = ProElementFree (&pro_e_feature_tree);
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.

```c
FILE : UserSktRevolveThinCut.c
PURPOSE : Creates a revolved thin cut.
/*====================================================================*
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;

ERROR_CHECK( a, "UgSktRevolveCreate.c", status );

#define C_PRINT(a) printf ( "%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;
    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 = fopen ( "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 );

/*---------------------------------------------------------------*
| Populating element PRO_E_FEATURE_FORM                            |
/*---------------------------------------------------------------*/
C_PRINT( " *** Processing Element PRO_E_FEAT_FORM *** " );
status = ProElementAlloc ( PRO_E_FEAT_FORM, &pro_e_feature_form );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REVOLVE;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_feature_form, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL, pro_e_feature_form );

/*-----------------------------------------------*/
  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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_FEAT_TYPE_SOLID;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_ext_surf_cut_solid_type, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL, pro_e_ext_surf_cut_solid_type );

/*-----------------------------------------------*/
  Populating element PRO_E_REMOVE_MATERIAL
/*-----------------------------------------------*/
C_PRINT( " *** Processing Element PRO_E_REMOVE_MATERIAL *** " );
status = ProElementAlloc ( PRO_E_REMOVE_MATERIAL, &pro_e_remove_material );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_MATERIAL_REMOVE;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_remove_material, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL, pro_e_remove_material );

/*-----------------------------------------------*/
  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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_FEAT_FORM_THIN;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_feat_form_is_thin, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL, pro_e_feat_form_is_thin );

/*---------------------------------------------------------------*
| Populating element PRO_E_THICKNESS (thin only)               |
/*---------------------------------------------------------------*/

C_PRINT( " *** Processing Element PRO_E_THICKNESS *** " );
status = ProElementAlloc ( PRO_E_THICKNESS, &pro_e_thickness );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 1.00;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_thickness, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL, pro_e_thickness );

/*---------------------------------------------------------------*
| 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 );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL, pro_e_rev_angle );

/*---------------------------------------------------------------*
| 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 );
status = ProElemtreeElementAdd ( pro_e_rev_angle, NULL, pro_e_rev_angle_from );

/*---------------------------------------------------------------*
| 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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_ANG_FROM_ANGLE;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_rev_angle_from_type, value );
status = ProElemtreeElementAdd ( pro_e_rev_angle_from_type, NULL, pro_e_rev_angle_from_type );
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_val );
   value_data.type = PRO_VALUE_TYPE_DOUBLE;
   value_data.v.d = 60.000000;
   status = ProValueAlloc ( &value );
   status = ProValueDataSet ( value, &value_data );
   status = ProElementValueSet ( pro_e_rev_angle_from_val, value );
   status = ProElemtreeElementAdd ( pro_e_rev_angle_from, NULL, pro_e_rev_angle_from_val );
   \------------------------------------------------------------------------*
   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 );
   status = ProElemtreeElementAdd ( pro_e_rev_angle, NULL, pro_e_rev_angle_to );
   \------------------------------------------------------------------------*
   Populating element PRO_E_REV_ANGLE
-> 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 );
   value_data.type = PRO_VALUE_TYPE_INT;
   value_data.v.i = PRO_REV_ANG_TO_ANGLE;
   status = ProValueAlloc ( &value );
   status = ProValueDataSet ( value, &value_data );
   status = ProElementValueSet ( pro_e_rev_angle_to_type, value );
   status = ProElemtreeElementAdd ( pro_e_rev_angle_to, NULL, pro_e_rev_angle_to_type );
   \------------------------------------------------------------------------*
   Populating element PRO_E_REV_ANGLE
-> 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 );
   value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 135.000;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_rev_angle_to_val, value );
status = ProElementSet ( pro_e_rev_angle_to, NULL, pro_e_rev_angle_to_val );

/*---------------------------------------------------------------*/
Populating element PRO_E_STD_SECTION
/*---------------------------------------------------------------*/
C_PRINT( " *** Processing Element PRO_E_STD_SECTION *** " );
status = ProElementAlloc ( PRO_E_STD_SECTION, &pro_e_std_section );
status = ProElementAdd ( pro_e_std_section, NULL, pro_e_std_section );

/*---------------------------------------------------------------*/
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 );
status = ProElementAdd ( pro_e_std_section, NULL, pro_e_std_sec_setup_plane );

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");
printf ( "Please select datum,surface,sldface,qltface_ID_5 type of Modelitem\n");
status = ProSelect ( "datum,surface,sldface,qltface", 1, NULL, NULL, NULL, &p_select, &n_select );
if ( n_select <= 0 ) return -1;
else
{
    status = ProSelectionCopy ( p_select[0], &sketch_refs[0] );
}
status = ProElementAlloc ( PRO_E_STD_SEC_PLANE, &pro_e_std_sec_plane );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet (pro_e_std_sec_plane, value);
status = ProElemtreeElementAdd (pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane);

/*-----------------------------------------------*/
// 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);
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);
status = ProValueDataSet (value, &value_data);
status = ProElementValueSet (pro_e_std_sec_plane_view_dir, value);
status = ProElemtreeElementAdd (pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_view_dir);

/*-----------------------------------------------*/
// 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);
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);
status = ProValueDataSet (value, &value_data);
status = ProElementValueSet (pro_e_std_sec_plane_orient_dir, value);
status = ProElemtreeElementAdd (pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_orient_dir);

/*-----------------------------------------------*/
// 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");
printf("Select datum,surface,sldface,qltface,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);
if ( n_select <= 0 ) return -1;
else
{
    status = ProSelectionCopy( p_select[0], &sketch_refs[1]);
}

status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_ORIENT_REF,
    &pro_e_std_sec_plane_orient_ref );
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_sec_plane_orient_ref, value );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
    pro_e_std_sec_plane_orient_ref );

/*---------------------------------------------------------------*
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);

/* Using the element tree from created feature */
status = ProFeatureElemtreeCreate ( &feature, &created_elemtree );

/*---------------------------------------------------------------*
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);

status = ProElementValueGet ( sketch_element, &value);
status = ProValueDataGet (value, &value_data);
section = (ProSection)value_data.v.p;

/*---------------------------------------------------------------*/
Creating a 3-D section

/*---------------------------------------------------------------*/
status = UserSectionBuild (( ProSection )(value_data.v.p),
sketch_refs );
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);
status = ProElementValueGet ( pro_e_std_matrlside, &value );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_MATERIAL_SIDE_ONE;
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_matrlside, value );
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);
status = ProElementValueGet ( pro_e_std_direction, &value );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_CR_IN_SIDE_TWO;
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_direction, value );
ProElempathFree (&path);

/*---------------------------------------------------------------*
Redefining the feature to make it complete.
*---------------------------------------------------------------*/
opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMS;
status = ProSelectionAsmcomppathGet (model_sel, &comp_path);

status = ProFeatureRedefine (&comp_path, &feature, created_elemtree,
opts, 1, &errors);

/*---------------------------------------------------------------*
Free up the allocated memory.
*---------------------------------------------------------------*/
status = ProElementFree (&created_elemtree );
status = ProElementFree (&pro_e_feature_tree );
free(sketch_refs);
return (status);
}
#endif C_PRINT

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.

/*====================================================================*
FILE    : UgSktRevolveSurface.c
PURPOSE : Includes all possible element assignments for revolved features.
*====================================================================*/

#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;

ERROR_CHECK( a, "UgSktRevolveCreate.c", status );

#define C_PRINT(a) printf ( "%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_elmtree;

    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;
Creating Extruded and Revolved Features

```c
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 = fopen ( "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 );

/**************************************************************************/
Populating element PRO_E_FEATURE_FORM
**************************************************************************/
C_PRINT( " *** Processing Element PRO_E_FEATURE_FORM *** " );
status = ProElementAlloc ( PRO_E_FEATURE_FORM, &pro_e_feature_form );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REVOLVE;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_feature_form, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL, pro_e_feature_form );

/**************************************************************************/
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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_FEAT_TYPE_SURFACE;
```
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_ext_surf_cut_solid_type, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_ext_surf_cut_solid_type );

/***************************************************************'
 Populating element PRO_E_REMOVE_MATERIAL
="/***************************************************************'

status = ProElementAlloc ( PRO_E_REMOVE_MATERIAL, &pro_e_remove_material );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_MATERIAL_ADD;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_remove_material, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_remove_material );

/***************************************************************'
 Populating element PRO_E_FEAT_FORM_IS_THIN
="/***************************************************************'

status = ProElementAlloc ( PRO_E_FEAT_FORM_IS_THIN,
&pro_e_feat_form_is_thin );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_FEAT_FORM_NO_THIN;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_feat_form_is_thin, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_feat_form_is_thin );

/***************************************************************'
 Populating compound element PRO_E_STD_REV_ANGLE
="/***************************************************************'

status = ProElementAlloc ( PRO_E_REV_ANGLE, &pro_e_rev_angle );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_rev_angle );

/***************************************************************'
 Populating element PRO_E_REV_ANGLE
="/***************************************************************'

status = ProElementAlloc ( PRO_E_REV_ANGLE_FROM, &pro_e_rev_angle_from );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_rev_angle_from );
status = ProElemtreeElementAdd ( pro_e_rev_angle, NULL, pro_e_rev_angle_from );

/**********************************************************************\n  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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_ANG_FROM_NONE;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_rev_angle_from_type, value );
status = ProElemtreeElementAdd ( pro_e_rev_angle_from, NULL, pro_e_rev_angle_from_type );

/**********************************************************************\n  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 );
status = ProElemtreeElementAdd ( pro_e_rev_angle, NULL, pro_e_rev_angle_to );

/**********************************************************************\n  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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_ANG_TO_REF;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_rev_angle_to_type, value );
status = ProElemtreeElementAdd ( pro_e_rev_angle_to, NULL, pro_e_rev_angle_to_type );

/**********************************************************************\n  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 );
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, &to_ref_type) != PRO_TK_NO_ERROR)
return PRO_TK_GENERAL_ERROR;
status = ProModelitemInit ( model, to_ref_id, to_ref_type,
&to_ref_item);

status = ProSelectionAlloc (NULL, &to_ref_item, &value_data.v.r);
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_rev_angle_to_ref, value );
status = ProElemtreeElementAdd ( pro_e_rev_angle_to, NULL,
pro_e_rev_angle_to_ref );

/*---------------------------------------------------------------*\
... Populating element PRO_E_STD_SECTION
\*---------------------------------------------------------------*/
C_PRINT( " *** 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 );

/*---------------------------------------------------------------*\
... Populating element PRO_E_STD_SECTION
\*---------------------------------------------------------------*/
C_PRINT( " *** Processing Element PRO_E_STD_SEC_SETUP_PLANE *** " );
status = ProElementAlloc ( PRO_E_STD_SEC_SETUP_PLANE,
&pro_e_std_sec_setup_plane );
status = ProElemtreeElementAdd ( pro_e_std_section, NULL,
pro_e_std_sec_setup_plane );
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_PLANE *** " );
status = ProMessageDisplay ( message_file, "Select Surface for sketch
placement");
printf ( "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 );  
if ( n_select <= 0 ) return -1;  
else  
{  
status = ProSelectionCopy ( p_select[0], &sketch_refs[0]);  
}  
status = ProElementAlloc ( PRO_E_STD_SEC_PLANE, &pro_e_std_sec_plane );  
value_data.type = PRO_VALUE_TYPE_SELECTION;  
value_data.v.r = p_select[0];  
status = ProValueAlloc ( &value );  
status = ProValueDataSet ( value, &value_data );  
status = ProElementValueSet ( pro_e_std_sec_plane, value );  
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL, pro_e_std_sec_plane );  
/*---------------------------------------------------------------*/  
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 );  
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 );  
status = ProValueDataSet ( value, &value_data );  
status = ProElementValueSet ( pro_e_std_sec_plane_view_dir, value );  
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL, pro_e_std_sec_plane_view_dir );  
/*---------------------------------------------------------------*/  
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 );  
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 );  
status = ProValueDataSet ( value, &value_data );  
status = ProElementValueSet ( pro_e_std_sec_plane_orient_dir, value );  
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL, pro_e_std_sec_plane_orient_dir );
Pro_e_std_sec_plane_orient_dir);

/*-----------------------------------------------*/
  Populating element PRO_E_STD_SECTION
-> PRO_E_STD_SEC_SETUP_PLANES
  -> 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");
printf("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);
if (n_select <= 0) return -1;
else
{
  status = ProSelectionCopy ( p_select[0], &sketch_refs[1]);
}
status = ProElementAlloc ( PRO_E_STD_SEC_PLANE_ORIENT_REF,
&pro_e_std_sec_plane_orient_ref);
value_data.type = PRO_VALUE_TYPE_SELECTION;
value_data.v.r = p_select[0];
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_sec_plane_orient_ref, value );
status = ProElemtreeElementAdd ( pro_e_std_sec_setup_plane, NULL,
pro_e_std_sec_plane_orient_ref );

/*-----------------------------------------------*/
  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);
/* Using the element tree from created feature */
status = ProFeatureElemtreeCreate ( &feature, &created_elemtree );

/*-----------------------------------------------*/
  Getting the initialized section element from the database.
/*-----------------------------------------------*/
/* path to PRO_E_SKETCHER element */
path_items[0].type = PRO_ELEM_PATH_ITEM_TYP_ID;
path_items[0].path_item.elem_id = PRO_E_STD_SECTION;
Creating Extruded and Revolved Features

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);

status = ProElementValueGet ( sketch_element, &value);
status = ProValueDataGet (value, &value_data);
section = (ProSection)value_data.v.p;

/*---------------------------------------------------------------*
Creating a 3-D section

status = UserSectionBuild (( ProSection )(value_data.v.p),
  sketch_refs);
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_itemelem_id = PRO_E_STD_DIRECTION;
status = ProElempathAlloc (&path);
status = ProElempathDataSet (path, path_items, 1);
status = ProElemtreeElementGet ( created_elemtree, path, &pro_e_std_direction);

status = ProElementValueGet ( pro_e_std_direction, &value );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_CR_IN_SIDE_ONE;
status = ProValueDataSet (value, &value_data);
status = ProElementValueSet ( pro_e_std_direction, value );
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);

status = ProElementValueGet(pro_e_srf_end_attributes, &value);
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_SURF_END_ATTR_OPEN;
status = ProValueDataSet(value, &value_data);
status = ProElementValueSet(pro_e_srf_end_attributes, value);
ProElempathFree(&path);

*---------------------------------------------------------------*
Redefining the feature to make it complete.
\*---------------------------------------------------------------*/
opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMMS;
status = ProSelectionAsmcomppathGet(model_sel, &comp_path);

status = ProFeatureRedefine(&comp_path, &feature, created_elemtree, opts, 1, &errors);

*---------------------------------------------------------------*
Free up the allocated memory.
\*---------------------------------------------------------------*/
status = ProElementFree(&created_elemtree);

status = ProElementFree(&pro_e_feature_tree);
free(sketch_refs);
return(status);
}

#undef C_PRINT

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):

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_FEATURE_TYPE</td>
<td>Feature type: PRO_FEAT_FIRST</td>
</tr>
<tr>
<td>PRO_E_FEATURE_FORM</td>
<td>PRO_EXTRUDE / PRO_REVOLVE</td>
</tr>
<tr>
<td>PRO_E_FEAT_FORM_IS_THIN</td>
<td>Feature Form Of Type ProRevFeatForm = PRO_REV_FEAT_FORM_NO_THIN for a Solid feature = PRO_REV_FEAT_FORM_THIN for a Thin feature</td>
</tr>
<tr>
<td>PRO_E_SKETCHER</td>
<td>Sketcher pointer. Used because the standard section requires selected references not available in an empty model.</td>
</tr>
<tr>
<td>PRO_E_STD_MATRLSIDE*</td>
<td>Mandatory if thin Of type ProExtMatlSide (Extrude) Of type ProRevMatlSide (Revolve)</td>
</tr>
<tr>
<td>PRO_E_THICKNESS</td>
<td>Mandatory ( \geq 0.0 ) if thin Of type PRO_VALUE_TYPE_DOUBLE</td>
</tr>
<tr>
<td>PRO_E_EXT_DEPTH_FROM</td>
<td>Compound Element (Extrude only)</td>
</tr>
<tr>
<td>PRO_E_EXT_DEPTH_FROM_VAL</td>
<td>Depth dimension (of type PRO_VALUE_TYPE_DOUBLE) (Extrude only)</td>
</tr>
<tr>
<td>PRO_E_REV_ANGLE_FROM</td>
<td>Compound Element (Revolve only)</td>
</tr>
<tr>
<td>PRO_E_REV_ANGLE_FROM_VAL</td>
<td>Angular dimension (of type PRO_VALUE_TYPE_DOUBLE) (Revolve only)</td>
</tr>
<tr>
<td>PRO_E_STD_FEATURE_NAME</td>
<td>Default given by application depending on the feature type. Can be modified by the user.</td>
</tr>
</tbody>
</table>

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)
/*====================================================================*/

#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;

ERROR_CHECK( a, "UgSktFirstFeatureCreate.c", status );

#define C_PRINT(a) printf ( "%s
", 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_element;

    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_sketcher;
ProError status;
ProValueData value_data;
ProSelection * p_select;
int n_select;

ProStringToWstring ( message_file, "utilities.txt" );

log_file = fopen ( "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 );

/*-------------------------------------------------------------------------------*/
// Populating element PRO_E_FEATURE_TYPE
="/----------------------------------------*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TYPE *** ");
status = ProElementAlloc ( PRO_E_FEATURE_TYPE, &pro_e_feature_type );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_FEAT_FIRST_FEAT;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_feature_type, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                 pro_e_feature_type );

/*-------------------------------------------------------------------------------*/
// Populating element PRO_E_FEATURE_FORM
="/----------------------------------------*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_FORM *** ");
status = ProElementAlloc ( PRO_E_FEATURE_FORM, &pro_e_feature_form );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXTRUDE;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_feature_form, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                 pro_e_feature_form );

/*-------------------------------------------------------------------------------*/
// 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 );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
                                 pro_e_std_ext_depth );
Creating Extruded and Revolved Features

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 );
status = ProElemtreeElementAdd ( pro_e_std_ext_depth, NULL,
 pro_e_ext_depth_from );

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 );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 120.000000;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_ext_depth_from_value, value );
status = ProElemtreeElementAdd ( pro_e_ext_depth_from, NULL,
 pro_e_ext_depth_from_value );

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);
/* Using the element tree from created feature */
status = ProFeatureElemtreeCreate ( &feature, &created_elemtree );

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_element, path, &sketch_element);

status = ProElementValueGet (sketch_element, &value);

status = ProValueDataGet (value, &value_data);

/*---------------------------------------------------------------*
| Creating a 3-D section                                       |
/*---------------------------------------------------------------*/

status = UserSectionFirstFeatureBuild ((ProSection*)value_data.v.p);

/*---------------------------------------------------------------*
| Allocating and assigning new ProValue to sketcher element     |
| :: Wildfire requirement                                       |
/*---------------------------------------------------------------*/

status = ProValueAlloc (&new_value);

status = ProValueDataSet (new_value, &value_data);

status = ProElementValueSet (sketch_element, new_value);

ProElempathFree (&path);

/*---------------------------------------------------------------*
| Redefining the feature to make it complete.                   |
/*---------------------------------------------------------------*/

opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMS;

status = ProSelectionAsmcomppathGet (model_sel, &comp_path);

status = ProFeatureRedefine (&comp_path, &feature, created_element,
   opts, 1, &errors);

/*---------------------------------------------------------------*
| Free up the allocated memory.                                |
/*---------------------------------------------------------------*/

status = ProElementFree (&created_element);

status = ProElementFree (&pro_e_feature_tree);

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 );
status = ProSecerrorAlloc ( &section_errors2 );
status = ProSecerrorAlloc ( &section_errors3 );
status = ProSection2DAlloc ( section_here );

printf("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] ) ;

line.type = PRO_2D_LINE;
line.end1[0] = 100;
line.end1[1] = 0;
line.end2[0] = 100;
line.end2[1] = 0;

status = ProSectionEntityAdd ( *section_here, (Pro2dEntdef*)&line, &line_id[1] ) ;

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] ) ;
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 \rightarrow PRO_E_STD_SECTION \rightarrow PRO_E_SKETCHER

   For Pro/ENGINEER Wildfire release:
   
   PRO_E_FEATURE_TREE \rightarrow 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).

\*====================================================================*/

/*---------------------- 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;

ERROR_CHECK( a, "UgSktFirstFeatureRevCreate.c", status );


#define C_PRINT(a) printf ( "%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 = fopen ( "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 );

/*---------------------------------------------------------------*/
Populating element PRO_E_FEATURE_TYPE
/*---------------------------------------------------------------*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_TYPE *** " );
status = ProElementAlloc ( PRO_E_FEATURE_TYPE, &pro_e_feature_type );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_FEAT_FIRST_FEAT;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_feature_type, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL, pro_e_feature_type );

/*---------------------------------------------------------------*/
Populating element PRO_E_FEATURE_FORM
/*---------------------------------------------------------------*/
C_PRINT( " *** Processing Element PRO_E_FEATURE_FORM *** " );
status = ProElementAlloc ( PRO_E_FEATURE_FORM, &pro_e_feature_form );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REVOLVE;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_feature_form, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL, pro_e_feature_form );

/*---------------------------------------------------------------*/
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 );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_FEAT_FORM_THIN; /* 128 */
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_feat_form_is_thin, value );
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL, pro_e_feat_form_is_thin );

/*---------------------------------------------------------------*/
Populating compound element PRO_E_REV_ANGLE

```c
C_PRINT( " *** Processing Element PRO_E_REV_ANGLE *** " );
status = ProElementAlloc ( PRO_E_REV_ANGLE, &pro_e_rev_angle);
status = ProElemtreeElementAdd ( pro_e_feature_tree, NULL,
pro_e_rev_angle );

C_PRINT( " *** Processing Element PRO_E_REV_ANGLE *** " );
status = ProElementAlloc ( PRO_E_REV_ANGLE_FROM, &pro_e_rev_angle_from );
status = ProElemtreeElementAdd ( pro_e_rev_angle, NULL,
pro_e_rev_angle_from );

C_PRINT( " *** Processing Element PRO_E_REV_ANGLE_FROM *** " );
status = ProElementAlloc ( PRO_E_REV_ANGLE_FROM_TYPE, &pro_e_rev_angle_from_type );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_REV_ANG_FROM_ANGLE;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_rev_angle_from_type, value );
status = ProElemtreeElementAdd ( pro_e_rev_angle_from, NULL,
pro_e_rev_angle_from_type );

C_PRINT( " *** Processing Element PRO_E_REV_ANGLE_FROM *** " );
status = ProElementAlloc ( PRO_E_REV_ANGLE_FROM_VAL, &pro_e_rev_angle_from_val );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 120.000000;
status = ProValueAlloc ( &value );
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_rev_angle_from_val, value );
```
status = ProElemtreeElementAdd ( pro_e_rev_angle_from, NULL, 
pro_e_rev_angle_from_val );

/*---------------------------------------------------------------*\
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);

/* Using the element tree from created feature */
status = ProFeatureElemtreeCreate ( &feature, &created_elemtree );

/*---------------------------------------------------------------*\
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);

status = ProElementValueGet ( sketch_element, &value);
status = ProValueDataGet (value, &value_data);

/*---------------------------------------------------------------*\
Creating a 3-D section
\*---------------------------------------------------------------*/

status = UserSectionFirstFeatureRevBuild (( ProSection * )
(&value_data.v.p) );

/*---------------------------------------------------------------*\
Allocating and assigning new ProValue to sketcher element
:: Wildfire requirement
\*---------------------------------------------------------------*/

status = ProValueAlloc ( &new_value );
status = ProValueDataSet ( new_value, &value_data);
status = ProElementValueSet ( sketch_element, new_value);
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)
;
status = ProElementValueGet ( pro_e_std_matrlside, &value );
value_data.type = PRO_VALUE_TYPE_INT;
value_data.v.i = PRO_EXT_MATERIAL_SIDE_ONE;
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_std_matrlside, value );
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_thick);
status = ProElementValueGet ( pro_e_thick, &value );
value_data.type = PRO_VALUE_TYPE_DOUBLE;
value_data.v.d = 10.0;
status = ProValueDataSet ( value, &value_data );
status = ProElementValueSet ( pro_e_thickness, value );

ProElempathFree (&path);

/*---------------------------------------------------------------*/
Redefining the feature to make it complete.
/*---------------------------------------------------------------*/
opts[0] = PRO_FEAT_CR_DEFINE_MISS_ELEMS;
status = ProSelectionAsmcomppathGet (model_sel, &comp_path);

status = ProFeatureRedefine (&comp_path, &feature, created_elemtree,
 opts, 1, &errors);

/*---------------------------------------------------------------*/
Free up the allocated memory.
/*---------------------------------------------------------------*/
status = ProElementFree (&created_elemtree );

status = ProElementFree (&pro_e_feature_tree );

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 );
   status = ProSecerrorAlloc ( &section_errors2 );
   status = ProSecerrorAlloc ( &section_errors3 );
   status = ProSection2DAlloc ( section_here );
   printf("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 ) ;

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] ) ;

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] ) ;

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] ) ;

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] ) ;

status = ProSectionEpsilonSet( *section_here, 0.1 );

status = ProSectionAutodim ( *section_here, &section_errors1 );

if ( status != PRO_TK_NO_ERROR )
status = ProSectionRegenerate ( *section_here, &section_errors3 );

if ( status != PRO_TK_NO_ERROR )
{
    status = ProSectionRegenerate ( *section_here, &section_errors3 );
    printf("The status for ProSectionRegenerate is = %d \n", status );
    printf("Number of errors in ProSectionRegenerate = %d \n", n_sec );
}

return ( status );

#undef C_PRINT
Creating Swept Features

This chapter describes the basic principles of creating a swept feature. The chapters ‘Principles of Feature Creation’ and ‘Creating Sketched Features’ provide the necessary background for this topic. Read those chapters before this one.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sweeps in Pro/ENGINEER Wildfire</td>
<td>26 - 3</td>
</tr>
<tr>
<td>Creating a Swept Feature</td>
<td>26 - 5</td>
</tr>
</tbody>
</table>
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 26-1: Element Tree for Sweep Feature

```
PRO_E_FEATURE_TREE
  PRO_E_FEATURE_TYPE
  PRO_E_FEATURE_FORM
  PRO_E_SWEEP_SPINE
  PRO_E_SWEEP_SECTION
```

The following table describes the elements in the element tree for sweeps.

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_FEATURE_TYPE</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>Feature type</td>
</tr>
<tr>
<td>PRO_E_FEATURE_FORM</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>Feature form (PRO_SWEEP)</td>
</tr>
<tr>
<td>PRO_E_SWEEP_SPINE</td>
<td>Compound</td>
<td>Trajectory (like (PRO_E_STD_SECTION))</td>
</tr>
<tr>
<td>PRO_E_SWEEP_SECTION</td>
<td>Compound</td>
<td>Section (like PRO_E_STD_SECTION)</td>
</tr>
</tbody>
</table>

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 26-2: 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 `ProFeatureElemtreeCreate()` 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 `ProFeatureElemtreeCreate()` 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.

```c
/*-----------------------------------------------*/
Pro/TOOLKIT includes
/*-----------------------------------------------*/
#include <ProToolkit.h>
#include <ProMd1.h>
#include <ProElement.h>
#include <ProFeatType.h>
#include <ProFeatForm.h>
#include <ProSweep.h>
#include <ProFeature.h>
#include <ProSection.h>
#include <ProSecdim.h>
```
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:  Create a sweep protrusion as the first feature.

int UgBaseSweepProtrCreate (ProPart part)
{
    ProError err;
    UgElemdata elem_data;
    ProFeature feature;
    ProErrorlist ftcr_errs;
    ProElement elem, elem_tree;
    ProSelection part_sel;
    ProModelitem part_mdlitem;
    ProFeatureCreateOptions ftcropts[] = {PRO_FEAT_CR_INCOMPLETE_FEAT};
    ProElempathItem sect_path[] = {{PRO_ELEM_PATH_ITEM_TYPE_ID,
                                    PRO_S_WEE EP_SECTION}};
    ProElempathItem spine_skt_path[] = {
        {PRO_ELEM_PATH_ITEM_TYPE_ID,
         PRO_E_S WEEP_SPINE},
        {PRO_ELEM_PATH_ITEM_TYPE_ID,
         PRO_E_S KETCHER}};
    ProElempathItem sect_skt_path[] = {
        {PRO_ELEM_PATH_ITEM_TYPE_ID,
         PRO_E_S WEEP_SECTION},
        {PRO_ELEM_PATH_ITEM_TYPE_ID,
         PRO_E_S KETCHER}};

    /*-----------------------------*/

ProValueData

spine_skt_val, sect_skt_val;

if (part == NULL)
{
    err = ProMdlCurrentGet ((Pro)&part);
}
/*
 Initialize the element tree.

 err = ProElementAlloc (PRO_E_FEATURE_TREE, &elem_tree);
*/
Add the feature type element to the tree.

elem_data.parent_element = elem_tree;
elem_dataelem_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 the 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 the 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 the 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 an incomplete feature to initialize the spine sketch handle.

err = ProMdlToModelitem ((ProMdl)part, &part_mdlitem);
err = ProSelectionAlloc (NULL, &part_mdlitem, &part_sel);
err = ProFeatureCreate (part_sel, elem_tree, ftcropts, 1, &feature,
&ftcr_errs);
  err = ProElementFree (&elem_tree);
/*-----------------------------*/
Retrieve the initialized element tree from the Pro/ENGINEER
database.
/*------------------------------*/
  err = ProFeatureElemtreeCreate (&feature, &elem_tree);
/*-----------------------------*/
Get the spine sketch handle from the element tree.
/*------------------------------*/
  err = UgFeatElemValueGet (elem_tree, spine_skt_path, 2,
&spine_skt_val);
/*-----------------------------*/
Create the sweep spine.
/*------------------------------*/
  err = UgSweepSpineSktCreate ((ProSection)spine_skt_val.v.p);
/*-----------------------------*/
Remove the sweep section element tree to avoid uninitialized
data.
/*------------------------------*/
  err = UgFeatElemRemove (elem_tree, sect_path, 1);
/*-----------------------------*/
Add the section method for the sweep cross 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 the feature as incomplete so you can add the spine and
initialize the section sketch.
/*------------------------------*/
  err = ProFeatureRedefine (NULL, &feature, elem_tree, ftcropts, 1,
&ftcr_errs);
  err = ProElementFree (&elem_tree);
/*-----------------------------*/
Retrieve the initialized element tree from the Pro/ENGINEER
database.
/*------------------------------*/
  err = ProFeatureElemtreeCreate (&feature, &elem_tree);
/*-----------------------------*/
Get the section sketch handle from the element tree.
/*------------------------------*/
  err = UgFeatElemValueGet (elem_tree, sect_skt_path, 2,
&sect_skt_val);
/*-----------------------------*/
Create the sweep sketch.
/*------------------------------*/
  err = UgSweepSectSktCreate ((ProSection)sect_skt_val.v.p);
/*--------------------------------------------------------------------*
Redefine the feature to create the complete feature.
ftcropts[0] = PRO_FEAT_CR_NO_OPTS;
err = ProFeatureRedefine (NULL, &feature, elem_tree, ftcropts, 1, &ftcr_errs);
err = ProElementFree (&elem_tree);
return (err);
} /*====================================================================*
FUNCTION: UgFeatElemAdd
PURPOSE:  Add 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);

        err = ProValueDataSet (value, &elem->value_data);
    }
    err = ProElementAlloc (elem->elem_id, &element);

    if (elem->value_data.type != -1)
    {
        err = ProElementValueSet (element, value);
    }
    err = ProElemtreeElementAdd (elem->parent_element, NULL, element);

    *added_elem = element;
    return (err);
}

FUNCTION: UgFeatElemAddSecMethod
PURPOSE:  Add the necessary elements below the PRO_E_STD_SECTION
          element to specify the section 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: Get 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);

err = ProElempathDataSet (elem_path, path, size);

err = ProElemtreeElementGet(tree, elem_path, &elem);

err = ProElementValueGet(elem, &value);

err = ProValueDataGet (value, elem_val);

return (err);

/*====================================================================*
FUNCTION: UgFeatElemRemove
PURPOSE: Remove 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);
err = ProElempathDataSet (elem_path, path, size);
err = ProElemtreeElementGet (tree, elem_path, &elem);
err = ProElemtreeElementRemove (tree, elem_path, &elem);
err = ProElementFree (&elem);
return (err);
}
/*====================================================================*
FUNCTION: UgSweepSpineSktCreate
PURPOSE:  Create 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);
    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);
    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);
    err = UgSectionRegenerate (section);
    return (err);
}
FUNCTION: UgSweepSectSktCreate

PURPOSE: Create a section sketch for a constant section sweep.

ProError UgSweepSectSktCreate (ProSection section)
{
    ProError                     err;
    Pro2dCircledef              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);
    printf ("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);

    err = ProSecdimCreate (section, &circle_id, pnt_types, 1,
                           PRO_TK_DIM_DIA, place_pnt, &dim_id);

    err = UgSectionRegenerate (section);
    return (err);
}

FUNCTION: UgSectionRegenerate

PURPOSE: Regenerate a section.

ProError UgSectionRegenerate (ProSection section)
{
    ProWSecerror     sec_errs;
    ProError         err;

    err = ProSecerrorAlloc (&sec_errs);
    err = ProSectionSolve (section,&sec_errs);
    err = ProSecerrorFree (&sec_errs);
    err = ProSecerrorAlloc (&sec_errs);
    err = ProSectionRegenerate (section, &sec_errs);
    err = ProSecerrorAlloc (&sec_errs);
    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 : UgSweepCreate.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) printf ("Status for %s is = %d\n", a, status );
    ERROR_CHECK(a, "UgSweepCreate.c", status);
#define C_PRINT(a) printf ("%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
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;
ProElempath element_path;
ProElempathItem 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 = ProElementtreeElementAdd ( pro_e_feature_tree, NULL, pro_e_feature_type );
C_LOG( " ProElementtreeElementAdd" );

/*---------------------------------------------------------------*
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 = ProElementtreeElementAdd ( pro_e_feature_tree, NULL, pro_e_feature_form );
C_LOG( " ProElementtreeElementAdd" );

/*---------------------------------------------------------------*
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 = ProElementtreeElementAdd ( pro_e_feature_tree, NULL, pro_e_sweep_spine );
C_LOG( " ProElementtreeElementAdd" );

/*---------------------------------------------------------------*
Populating element 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 = ProElementtreeElementAdd ( pro_e_sweep_spine, NULL, pro_e_sweep_spine );
C_LOG( " ProElementtreeElementAdd" );

Creating Swept Features

Creating Swept Features
NULL,
    pro_e_std_sec_setup);
C_LOG(" ProElemtreeElementAdd" );
/
*---------------------------------------------------------------*/
Populating element PRO_E_STD_SEC_SETUP
  -> PRO_E_STD_SEC_METHOD
STRUCTION ELEMENT
Populating element PRO_E_STD_SEC_SETUP
  -> PRO_E_STD_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
STRUCTURE ELEMENT
Populating element PRO_E_STD_SEC_SETUP
  -> PRO_E_STD_SEC_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 = ProElementTreeElementAdd ( pro_e_std_section_plane, NULL,
pro_e_std_sec_plane_orient_dir );
C_LOG( " ProElementTreeElementAdd" );

/*---------------------------------------------------------------*\
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 = ProElementTreeElementAdd ( pro_e_std_section_plane,
NULL, pro_e_std_sec_plane_orient_ref );
C_LOG( " ProElementTreeElementAdd" );

/*---------------------------------------------------------------*\
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 = ProFeatureElemtreeCreate ( &created_feature,
                                   &created_elemtree);
C_LOG( " ProFeatureElemtreeCreate " );

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 = ProFeatureElemtreeCreate ( &created_feature,
&created_elemtree_after1st);
C_LOG( " ProFeatureElemtreeCreate " );

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 = ProElempathAlloc ( &element_path );
C_LOG( " ProElempathAlloc " );

status = ProElempathDataSet ( element_path, element_path_item, 2 );
C_LOG( " ProElempathDataSet " );

status = ProElemtreeElementGet ( created_elemtree_after1st,
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 section sketch and finish the feature.
/*--------------------------------------------------------------*/
status = UserSweepSectionAdd ( value_data.v.p );
C_LOG( " PTTestSection2DAdd " );

status = ProFeatureRedefine ( NULL, &created_feature,
created_element_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 = ProElementFree ( &created_element);
C_LOG( " ProElementFree " );

status = ProElementFree ( &created_element_after1st);
C_LOG( " ProElementFree " );

free (references1);

return status;
}

#undef C_LOG
#undef C_PRINT
This chapter describes the Pro/TOOLKIT functions that enable you to create foreign datum surfaces and curves.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foreign Datum Surfaces</td>
<td>27 - 2</td>
</tr>
<tr>
<td>Foreign Datum Curves</td>
<td>27 - 3</td>
</tr>
</tbody>
</table>
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 Datum > Surface.
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 TWEAK menu. 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 ‘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.
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:

\[
\begin{align*}
X &= f(t) \\
Y &= g(t) \\
Z &= h(t)
\end{align*}
\]
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:

\[
\begin{align*}
X &= R \cos(2\pi t); \\
Y &= R \sin(2\pi t); \\
Z &= 0;
\end{align*}
\]

In these equations, \( \pi = 3.14159 \).

Pro/TOOLKIT provides the prototype for the evaluation function. The syntax is as follows:

define 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.

```c
/*==================================*/
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];
```
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);
    return;
}

Create the selection object for the current model to use in
ProFeatureCreate().

Create the selection object for the current model to use in
ProFeatureCreate().

Create the selection object for the current model to use in
ProFeatureCreate().

Create the selection object for the current model to use in
ProFeatureCreate().

Create the selection object for the current model to use in
ProFeatureCreate().

Create the selection object for the current model to use in
ProFeatureCreate().

Create the selection object for the current model to use in
ProFeatureCreate().

Create the selection object for the current model to use in
ProFeatureCreate().

Create the selection object for the current model to use in
ProFeatureCreate().

Create the selection object for the current model to use in
ProFeatureCreate().

Create the selection object for the current model to use in
ProFeatureCreate().
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);
This chapter describes the basic principles of creating a tweak surface replacement feature. The chapter ‘Principles of Feature Creation’ is a necessary background for this topic. Read that chapter before this one.

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.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Feature Element Tree</td>
<td>28 - 2</td>
</tr>
</tbody>
</table>
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 28-1: Element Tree for Surface Replacement

```
+------------------+
| PRO_E_FEATURE_TREE |
|                  |
| --PRO_E_FEATURE_TYPE|
|                  |
| --PRO_E_STD_REPLACED_SURF|
|                  |
| --PRO_E_STD_REPLACEMENT_SURF|
|                  |
| --PRO_E_STD_KEEP_QUILT|
```

The following table describes the elements in the element tree for the surface replacement feature.

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_FEATURE_TYPE</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>PRO_FEAT_REPLACE_SURF</td>
</tr>
<tr>
<td>PRO_E_STD_REPLACED_SURF</td>
<td>PRO_VALUE_TYPE_SELECTION</td>
<td>The surface to be removed</td>
</tr>
<tr>
<td>PRO_E_STD_REPLACEMENT_SURF</td>
<td>PRO_VALUE_TYPE_SELECTION</td>
<td>The replacement surface</td>
</tr>
<tr>
<td>PRO_E_STD_KEEP_QUILT</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>Specifies whether to keep the quilt</td>
</tr>
</tbody>
</table>

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.
29

External Analyses and Analysis Features

This chapter describes the functions that enable you to create analysis and analysis feature objects in a Pro/ENGINEER solid.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>29 - 2</td>
</tr>
<tr>
<td>Interactive Creation of External Analysis</td>
<td>29 - 3</td>
</tr>
<tr>
<td>Interactive Creation of External Analysis Feature</td>
<td>29 - 3</td>
</tr>
<tr>
<td>Storage of External Analysis Feature in Pro/ENGINEER</td>
<td>29 - 4</td>
</tr>
<tr>
<td>Registering an External Analysis with Pro/ENGINEER</td>
<td>29 - 5</td>
</tr>
<tr>
<td>Analysis Callbacks</td>
<td>29 - 5</td>
</tr>
<tr>
<td>Pro/TOOLKIT Analysis Information</td>
<td>29 - 8</td>
</tr>
<tr>
<td>Results Data</td>
<td>29 - 8</td>
</tr>
<tr>
<td>Analysis Attributes</td>
<td>29 - 29</td>
</tr>
<tr>
<td>Visiting Saved External Analyses</td>
<td>29 - 29</td>
</tr>
<tr>
<td>Visiting External Analyses Features</td>
<td>29 - 30</td>
</tr>
<tr>
<td>Using the Model without Pro/TOOLKIT</td>
<td>29 - 30</td>
</tr>
</tbody>
</table>
Overview

The Pro/ENGINEER module Behavioral Modeler 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 external; this stresses the fact that the computations could be performed by a separate application, either a general calculation application such as Excel, or a specialist third-party application dedicated to a specific design task.

Pro/TOOLKIT users should practice using standard Pro/ENGINEER analyses and analysis features before studying Pro/TOOLKIT external analyses.

The functions and data structures specific to external analysis features are declared in the header file ProAnalysis.h.
Interactive Creation of External Analysis

If a Pro/TOOLKIT application registers an External Analysis type, the Analysis menu on the Pro/ENGINEER toolbar contains an extra button labeled External Analysis. Click this button to see the External Analysis dialog.

The selector at the top of the dialog 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. 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.

Interactive Creation of External Analysis Feature

On the Pro/ENGINEER top menu bar, select the command Insert > Model Datum > Analysis. This displays the Pro/ENGINEER Analysis dialog, 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 external analysis, there will be an additional button labeled External Analysis in the Type section.

Set the name of the analysis first, set the type to External Analysis, and continue to the next step, which is called “Definition.” The External Analysis dialog appears. This behaves exactly as for an external analysis, until you click the Close button. It then returns to the Analysis dialog, 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 external analysis defines no parameters or no datums.

Storage of External Analysis Feature in Pro/ENGINEER

An external 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 external analysis features that is not stored as either geometry references or dimensions.
Registering an External Analysis with Pro/ENGINEER

Functions introduced:

- **ProAnalysisTypeRegister()**

  The function `ProAnalysisTypeRegister()` registers an external 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 used in creating an analysis feature will include the button **External Analysis**. Refer to the sections 'Interactive Creation of External Analysis' and 'Interactive Creation of External Analysis Feature' for more information on creating these objects.

Analysis Callbacks

When registering an **External 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 an external analysis or analysis feature:

<table>
<thead>
<tr>
<th>Callback</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>infoalloc</td>
<td>Allocate memory for the Pro/TOOLKIT application information about the external analysis.</td>
</tr>
<tr>
<td>ui</td>
<td>Pro/TOOLKIT prompts the user for inputs that define the analysis—for example, select a surface datum point on which to position a csys.</td>
</tr>
<tr>
<td>compcheck</td>
<td>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.</td>
</tr>
<tr>
<td>compute</td>
<td>Perform the analysis computation and store the results in memory.</td>
</tr>
<tr>
<td>display</td>
<td>Display graphics showing the computation result.</td>
</tr>
<tr>
<td>output</td>
<td>Pass a set of text lines to Pro/ENGINEER for display in the Analysis dialog to show the result of the computation.</td>
</tr>
<tr>
<td>infocopy</td>
<td>Copy the application information from an existing analysis to a new one. Call infocopy during creation because of the way in which Pro/ENGINEER handles feature creation.</td>
</tr>
<tr>
<td>dims</td>
<td>Pro/TOOLKIT gives Pro/ENGINEER a list of double values needed to calculate the geometry. Pro/ENGINEER stores these as model dimensions.</td>
</tr>
<tr>
<td>result</td>
<td>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.</td>
</tr>
</tbody>
</table>
When the Pro/ENGINEER user saves the analysis to the solid:

<table>
<thead>
<tr>
<th>Callback</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>savecheck</td>
<td>Tell Pro/ENGINEER whether the description of the analysis can be saved.</td>
</tr>
<tr>
<td>infosave</td>
<td>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.</td>
</tr>
</tbody>
</table>

When the Pro/ENGINEER user retrieves a solid containing analyses:

<table>
<thead>
<tr>
<th>Callback</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inforetrieve</td>
<td>Pro/ENGINEER provides an array of ProSelection 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.)</td>
</tr>
</tbody>
</table>

When the Pro/ENGINEER user leaves the Analysis dialog without saving the new analysis, or erases a solid containing an external analysis:

<table>
<thead>
<tr>
<th>Callback</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>infofree</td>
<td>The Pro/TOOLKIT application frees the memory used by its internal description of the analysis.</td>
</tr>
</tbody>
</table>

Many of the callbacks will be called during other commands in Pro/ENGINEER whenever the external 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. The description appears alongside the parameter in the Analysis dialog. 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
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 an external 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.

Example 1: Offset Coordinate System Datum

The following example code shows a Pro/TOOLKIT application that defines an external 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.

```c
#include "ProToolkit.h"
#include "prodevelop.h"
#include "ProFeature.h"
#include "ProAnalysis.h"
#include "ProArray.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 "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;

/*====================================================================*
FUNCTION : UsrSurfcsysUiAction()
PURPOSE : Callback for UI action for surface csys
/*====================================================================*/
ProError UsrSurfcsysUiAction(
    ProAnalysis analysis)
{
    Surfcsysdata_t *data;
    ProError status;
    ProSelection *sel;
    int nSel;
    ProModelitem modelitem;
    ProPoint p;
    ProDtmpntType ptype;
    ProError UsrYesnoGet();

    /* 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,&nSel)
            != PRO_TK_NO_ERROR || nSel < 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)
            { /* 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. */
    ProError UsrYesnoGet();
}
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];
ProGraphicsLineDraw(pos);

/*-------------------------------------------------------*/
* Draw a line for the Y vector                           *
*-------------------------------------------------------*/
ProGraphicsPenPosition(data->csysdata.origin);
pos[0]=data->csysdata.origin[0]+data->csysdata.y_vector[0];
ProGraphicsLineDraw(pos);

/*-------------------------------------------------------*/
* Draw a line for the Z vector                           *
*-------------------------------------------------------*/
ProGraphicsPenPosition(data->csysdata.origin);
pos[0]=data->csysdata.origin[0]+data->csysdata.z_vector[0];
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(analyses, (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];
ProGraphicsLineDraw(pos);

    /*-------------------------------------------------------*/
    * Draw a line for the Y vector                              
    /*-------------------------------------------------------*/
    ProGraphicsPenPosition(data->csysdata.origin);
pos[0]=data->csysdata.origin[0]+data->csysdata.y_vector[0];
ProGraphicsLineDraw(pos);

    /*-------------------------------------------------------*/
    * Draw a line for the Z vector                              
    /*-------------------------------------------------------*/
    ProGraphicsPenPosition(data->csysdata.origin);
pos[0]=data->csysdata.origin[0]+data->csysdata.z_vector[0];
ProGraphicsLineDraw(pos);

    return(PRO_TK_NO_ERROR);
}
/*----------------------------------------*/
\* Write the origin of the csys
\*----------------------------------------*/
  sprintf(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
\*----------------------------------------*/
  sprintf(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
\*----------------------------------------*/
  sprintf(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
\*----------------------------------------*/
  sprintf(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);
}

/*====================================================================*\nFUNCTION : UsrSurfcsysSavecheckAction()
PURPOSE  : Callback to tell Pro/E whether the analysis application
data can be saved
/*====================================================================*/
ProError UsrSurfcsysSavecheckAction(
  ProAnalysis analysis)
```c
{    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);

Pro/TOOLKIT User's Guide
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 user_initialize()
{
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);
}

void user_terminate()
{
}
FUNCTION : UsrPointPtypeGet()
PURPOSE : Utility to get the placement type of a datum point

```c
/*====================================================================*
FUNCTION : UsrPointPtypeGet()
PURPOSE  : Utility to get the placement type of a datum point
====================================================================*/
int UsrPointPtypeGet(
    ProSolid owner,
    ProPoint p,
    ProDtmpntType *ptype)
{
    ProElempathItem items[4];
    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_DTMPNT_PNTS;
    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_DTMPNT_TYPE;
    ProElempathAlloc(&epath);
    ProElempathDataSet(epath, items, 3);
    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.

```c
int UsrPointSrfGet(
    ProSelection point,
    ProSurface *surface)
{
    ProElempathItem items[4];
    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_DTMPNT_PNTS;
    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_DTMPNT_PLACE_SURF;
    ProElempathAlloc(&epath);
    ProElempathDataSet(epath, items, 3);
    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

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?

FooBar

/*---------------------------------------------*/
FUNCTION : UsrYesnoGet()
PURPOSE : Prompt for a yes/no answer

WHAT HAPPENS WITH FOREIGN LANGUAGES?
#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);
    sprintf(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);

    sprintf(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

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;
    ProStringToWstring(sname, slotname);
    if(ProExtdataSlotCreate(class, sname, &slot) !=
        PROEXTDATA_TK_NO_ERROR)
    {
        slot.p_class = class;
        ProStringToWstring(slot.slot_name, slotname);
    }
    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 an external 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 external 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 external analysis to save time when making other changes to the model.

Visiting Saved External Analyses

Functions introduced:

- ProSolidAnalysisVisit()
- ProAnalysisNameGet()

The function ProSolidAnalysisVisit() visits a saved external analysis in a part or assembly. It does not visit standard saved analyses, nor external analyses in features.

The function ProAnalysisNameGet() provides the name under which an external analysis is saved.
Visiting External 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 external analyses will

retain their old values, and external analysis features will have

their geometry frozen.

If a Pro/TOOLKIT application is terminated during a

Pro/ENGINEER session, any external analysis types it registered

will be automatically deregistered, and any analyses that use those
types will be frozen.
This chapter describes the Pro/TOOLKIT functions for manufacturing operations. Familiarity with Pro/NC functions simplifies the use of these manufacturing functions.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manufacturing Models</td>
<td>30 - 2</td>
</tr>
<tr>
<td>Creating a Manufacturing Model</td>
<td>30 - 3</td>
</tr>
<tr>
<td>Analyzing a Manufacturing Model</td>
<td>30 - 3</td>
</tr>
<tr>
<td>Creating Manufacturing Objects</td>
<td>30 - 9</td>
</tr>
<tr>
<td>Analyzing Manufacturing Features</td>
<td>30 - 38</td>
</tr>
</tbody>
</table>
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.
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 ‘Assemblies’ 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.getAssem**—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.

```c
/*===============================================================*/
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 ‘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 ‘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:

```c
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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Turn</th>
<th>Turn Groove</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOSE_RADIUS</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>TOOL_WIDTH</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>SIDE_WIDTH</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>LENGTH</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>SIDE_ANGLE</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>END_ANGLE</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>GAUGE_X_LENGTH</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>GAUGE_Z_LENGTH</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>TOOL_MATERIAL</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>HOLDER_TYPE</td>
<td>•</td>
<td></td>
</tr>
</tbody>
</table>
The following table lists the parameters required to be defined for milling tools.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Mill</th>
<th>Side Mill</th>
<th>Thread Mill</th>
<th>Groove</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUTTER_DIAM</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>CORNER_RADIUS</td>
<td>•</td>
<td>•</td>
<td></td>
<td>•</td>
</tr>
<tr>
<td>CUTTER_WIDTH</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHANK_DIAM</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LENGTH</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>INSERT_LENGTH</td>
<td></td>
<td></td>
<td></td>
<td>•</td>
</tr>
<tr>
<td>END_OFFSET</td>
<td></td>
<td></td>
<td></td>
<td>•</td>
</tr>
<tr>
<td>SIDE_ANGLE</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GAUGE_X_LENGTH</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GAUGE_Z_LENGTH</td>
<td>•</td>
<td>•</td>
<td></td>
<td>•</td>
</tr>
<tr>
<td>NUM_OF_TEETH</td>
<td>•</td>
<td>•</td>
<td></td>
<td>•</td>
</tr>
<tr>
<td>TOOL_MATERIAL</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following table lists the parameters required to be defined for auxiliary and contouring tools.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Auxiliary</th>
<th>Contouring</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUTTER_DIAM</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>LENGTH</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>
The following table lists the parameters required to be defined for holemaking tools.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Drill</th>
<th>Csink</th>
<th>Tap</th>
<th>Ream</th>
<th>Center</th>
<th>Bore</th>
<th>Back</th>
<th>Spot</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUTTER_DIAM</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>POINT_DIAMETER</td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DRILL_DIAMETER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BODY_DIAMETER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LENGTH</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>CHAMFER_LENGTH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DRILL_LENGTH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSERT_LENGTH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TIP_OFFSET</td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GAUGE_OFFSET</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUTTING_OFFSET</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIDE_ANGLE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>POINT_ANGLE</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CSINK_ANGLE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GAUGE_X_LENGTH</td>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>GAUGE_Z_LENGTH</td>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>TOOL_MATERIAL</td>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

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 \texttt{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 \texttt{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 \texttt{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).

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>The name used to identify the tool</td>
<td>R</td>
</tr>
<tr>
<td>Type</td>
<td>Mill, drill, and so on</td>
<td>R</td>
</tr>
<tr>
<td>Parameters (for parameter-driven tools only)</td>
<td>Tool parameters</td>
<td>R</td>
</tr>
<tr>
<td>Model (for solid tools only)</td>
<td>Model that represents the tool</td>
<td>R</td>
</tr>
</tbody>
</table>

### Example 2: Creating a Tool from a Solid Model

The following example code shows how to create a tool from solid model.

```c
/*===========================================================*/
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.

```c
/*===============================================================*/
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);
    }
}
```
```c
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 30-2: 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.

```c
/*==================================================================*
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 = ProElmtreeElementAdd (mfg_param_elem, NULL,
                                             name_elem);
        }
    }
}`
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 = ProElemtreeElementAdd (mfg_param_elem, NULL, value_elem);
}

/*---------------------------------------------*/
Add the parent to the tree.
"*---------------------------------------------*/
if (status == PRO_TK_NO_ERROR)
{
    status = ProElemtreeElementAdd (*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.
```c
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;

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).

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell type</td>
<td>Mill, mill/turn, and so on.</td>
<td>R</td>
</tr>
<tr>
<td>Number of axes</td>
<td>The number of axes</td>
<td>R</td>
</tr>
<tr>
<td>Table direction</td>
<td>Horizontal or vertical (for turn or mill/turn)</td>
<td>R</td>
</tr>
<tr>
<td>Machine number heads</td>
<td>1 or 2 (for turn or mill/turn)</td>
<td>R</td>
</tr>
<tr>
<td>Name</td>
<td>The workcell name</td>
<td>O</td>
</tr>
<tr>
<td>Tooling</td>
<td>Add tools to the workcell.</td>
<td>O</td>
</tr>
<tr>
<td>Tool table</td>
<td>Manipulate the tools in a tool table.</td>
<td>O</td>
</tr>
<tr>
<td>Parameters</td>
<td>The workcell parameters.</td>
<td>O</td>
</tr>
</tbody>
</table>
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 30-3: 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:

<table>
<thead>
<tr>
<th>POSITION</th>
<th>TOOL_ID</th>
<th>REGISTER</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>drill_M8</td>
<td>8MM</td>
<td>Drill</td>
</tr>
<tr>
<td>2</td>
<td>drill_M10</td>
<td>10MM</td>
<td>Drill</td>
</tr>
</tbody>
</table>
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 5: 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.

```c
/*==================================================================*
FUNCTION: UserWorkcellCreate
PURPOSE: Create a workcell.
/*==================================================================*/
int UserWorkcellCreate (
    ProAppData  p_mfg_model,
    int         dummy)
{
    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;

    UserWpieceSelCreate (pMfg, int, ProType, ProSelection*);

    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                },
    }
```
```c
{PRO_E_FEAT_NAME, PRO_VALUE_TYPE_WSTRING});

int wc_elem_type_size = sizeof(workcell_elem_table)/sizeof (ElemTable);

err = ProElementAlloc (PRO_E_FEATURE_TREE, &wc_elem_tree);

for (i=0 ; i < wc_elem_type_size; i++)
{
    err = ProElementAlloc (workcell_elem_table[i].elem_type,
                           &wc_elem);

    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");
        }
        break;
    case PRO_E_TOOL_TABLES:
        err = UserToolTableSetup (&wc_elem);
        break;
    default:
        fprintf (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);
}

if (err == PRO_TK_NO_ERROR)
{
  err = ProElementValueSet (wc_elem, value);
}

if (err == PRO_TK_NO_ERROR)
{
  err = ProElementtreeElementAdd (wc_elem_tree, NULL,
                                wc_elem);
}

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);
return ((int)err);

/*================================================================*\nFUNCTION: UserWpieceSelCreate()
PURPOSE:  Create a selection handle for the mfg workpiece
          or the 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);

if (err == PRO_TK_NO_ERROR)
{
    err = ProMfgFeatureOwnerGet (mfg_model, &comp_path);
}

if (err == PRO_TK_NO_ERROR)
{
    err = ProModelItemInit (mfg_solid, item_id, item_type, &wkpiece_item);
}

if (err == PRO_TK_NO_ERROR)
{
    err = ProSelectionAlloc (&comp_path, &wkpiece_item, selection);
}

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).

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workcell</td>
<td>The identifier of the workcell feature in which to perform the operation</td>
<td>R</td>
</tr>
<tr>
<td>Machine coordinate system</td>
<td>The identifier of the machining coordinate system</td>
<td>R</td>
</tr>
<tr>
<td>Name</td>
<td>The operation name</td>
<td>O</td>
</tr>
<tr>
<td>Comments</td>
<td>The operation comments</td>
<td>O</td>
</tr>
<tr>
<td>From point</td>
<td>The datum point from which to start the operation</td>
<td>O</td>
</tr>
<tr>
<td>Home point</td>
<td>The datum point on which to end the operation</td>
<td>O</td>
</tr>
<tr>
<td>Parameters</td>
<td>The operation parameters</td>
<td>O</td>
</tr>
</tbody>
</table>
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 30-4: Home Point Element Tree

![Home Point Element Tree Diagram]

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 \textit{not} to the datum point feature, but to the datum point geometry. To find this geometry, call the function \texttt{ProFeatureGeomitemVisit}().

Example 6: Creating an Operation

This example shows how to create an operation.

```c
/*==================================================================*
FUNCTION: UserOperationCreate
PURPOSE:  Create an operation using the 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);

for (i = 0; i < oper_elem_type_size; i++)
{
  err = ProElementAlloc (oper_elem_table[i].elem_type,
          &oper_elem);

  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));
      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:
      fprintf (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);
}
if (err == PRO_TK_NO_ERROR)
{
    err = ProElementValueSet (oper_elem, value);
}
if (err == PRO_TK_NO_ERROR)
{
    err = ProElementFree (&oper_elem_tree);
    return 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).

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature type</td>
<td>The feature type</td>
<td>R</td>
</tr>
<tr>
<td>Type</td>
<td>The sequence type</td>
<td>R</td>
</tr>
<tr>
<td>Operation</td>
<td>Operation to which to add the NC sequence</td>
<td>R</td>
</tr>
<tr>
<td>Retraction plane</td>
<td>The retraction plane</td>
<td>R</td>
</tr>
<tr>
<td>Tool</td>
<td>The tool</td>
<td>R</td>
</tr>
<tr>
<td>Csys</td>
<td>The manufacturing coordinate system</td>
<td>R</td>
</tr>
<tr>
<td>Parameters</td>
<td>The manufacturing parameters</td>
<td>R</td>
</tr>
<tr>
<td>Name</td>
<td>The name of the NC sequence</td>
<td>O</td>
</tr>
<tr>
<td>Number of axes</td>
<td>The number of axes</td>
<td>O</td>
</tr>
<tr>
<td>Machine head</td>
<td>The machine head</td>
<td>O</td>
</tr>
<tr>
<td>Fixture</td>
<td>The fixture</td>
<td>O</td>
</tr>
</tbody>
</table>

**Entities to be machined**

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Surface</td>
<td>The surface</td>
<td>R</td>
</tr>
<tr>
<td>Holes or volume</td>
<td>The holes or volume</td>
<td>R</td>
</tr>
<tr>
<td>Start path</td>
<td>The start path</td>
<td>R</td>
</tr>
<tr>
<td>End path</td>
<td>The end path</td>
<td>R</td>
</tr>
</tbody>
</table>

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 holemaking, the following parameters are required:

- CUT_FEED
- TOLERANCE
- SPINDLE_SPEED
- CLEAR_DIST

Both milling and holemaking 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 `ProFeatureElemtreeCreate()` 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 7: Adding Surfaces

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

```c
/*==================================================================* /
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 30-5: 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 8: Creating a Conventional Milling Sequence**

This example shows how to create a milling feature using an element tree.

```c
/*==================================================================*\nFUNCTION: UserMillingCreate
PURPOSE: Create a milling feature using an 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 */
    ProValue       value           = (ProValue)NULL;
    ProSelection   selection;
    ProFeature     ncseq_feature;

    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);

int         ncseq elem type size = sizeof (ncseq elem table)/
sizeof (ElemTable);

/*.---------------------------------------------*
 Allocate the feature tree element.
---------------------------------------------*/
err = ProElementAlloc (PRO_E_FEATURE_TREE, &ncseq_elem_tree);

for (i = 0 ; i < ncseq_elem_type_size; i++)
{
    /*---------------------------------------------*
 Allocate the sequence element.
---------------------------------------------*/
    err = ProElementAlloc(ncseq_elem_table[i].elem_type,
            &ncseq_elem);

    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:
    fprintf (stderr, "Error setting element type\n");
    return PRO_TK_GENERAL_ERROR;
    break;
}
/*------------------------------------------------------------------*
If element is simple, add 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);
    if (err == PRO_TK_NO_ERROR)
        err = ProElementValueSet (ncseq_elem, value);
}
/*------------------------------------------------------------------*
Add the element to the feature tree.
\*------------------------------------------------------------------*/
if (err == PRO_TK_NO_ERROR)
    err = ProElemtreeElementAdd (ncseq_elem_tree, NULL,
                                ncseq_elem);

/*------------------------------------------------------------------*
Create the NC sequence feature in the workpiece.
\*------------------------------------------------------------------*/
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, ncseq_elem_tree, NULL, 0, &ncseq_feature, &errors);
}

/*------------------------------------------------------------------*
 Free the feature tree.
 *------------------------------------------------------------------*/
err = ProElementFree (&ncseq_elem_tree);

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.
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.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>31 - 2</td>
</tr>
<tr>
<td>Callback Types</td>
<td>31 - 2</td>
</tr>
<tr>
<td>Functions</td>
<td>31 - 3</td>
</tr>
</tbody>
</table>
Overview

Pro/TOOLKIT support for third-party tool manager applications is in three parts:

• Setting up optional, user-specified search parameters, that is, the name, object, and valid values for parameters not native to Pro/ENGINEER.

• Extracting data and conditions from the Pro/ENGINEER-native query to create the query of the third-party tool database.

• Formatting for display in the Pro/ENGINEER result dialog the query data returned by the third-party tool manager application.

Callback Types

• ProMfgdbSearchAction
• ProMfgdbLoginAction
• ProMfgdbLogoffAction

These callback types specify functions the user supplies by calling ProMfgdbRegister(). Callback ProMfgdbLoginAction() specifies the function that initiates access to the external database. ProMfgdbSearchAction() specifies the function that initiates a database query, performs the search, and returns the ProArray of matches. Callback ProMfgdbLogoffAction() specifies the function that closes a connection to the external database.
Functions

Functions introduced:

- `ProMfgdbRegister()`
- `ProMfgdbDataAlloc()`
- `ProMfgdbDataDbnameAdd()`
- `ProMfgdbNameCreate()`
- `ProMfgdbSearchoptCreate()`
- `ProMfgdbDataSearchoptAdd()`
- `ProMfgdbSearchoptApplicDataAdd()`
- `ProMfgdbSearchoptAllowedValueAdd()`
- `ProMfgdbQuerynodeLeaf()`
- `ProMfgdbQuerynodeLeftChildGet()`
- `ProMfgdbQuerynodeRightChildGet()`
- `ProMfgdbQuerynodeLogicOperGet()`
- `ProMfgdbQuerynodeExprGet()`
- `ProMfgdbQueryTargetGet()`
- `ProMfgdbExprNameGet()`
- `ProMfgdbExprCategoryGet()`
- `ProMfgdbExprValueGet()`
- `ProMfgdbExprValueTypeGet()`
- `ProMfgdbExprCompopGet()`
- `ProMfgdbMatchAlloc()`
- `ProMfgdbMatchParamAdd()`

The function `ProMfgdbRegister()` registers an external database with Pro/ENGINEER. Call this function when your Pro/TOOLKIT application is registered with Pro/ENGINEER.

Function `ProMfgdbDataAlloc()` allocates memory to store the names of databases to be searched. This memory also stores optional, user-specified database search parameters used to create queries in the Search dialog.
Function **ProMfgdbDataDbnameAdd()** adds the name of an external tool database to the memory allocated by **ProMfgdbDataAlloc()**. To specify more than one database, make multiple calls to **ProMfgdbDataDbnameAdd()**.

Function **ProMfgdbNameCreate()** allocates memory for names of the groups into which search options are organized. This function also initializes the structure. Specify multiple group names with multiple calls of this function.

Function **ProMfgdbSearchoptCreate()** allocates and initializes a structure for a user-specified search option.

Function **ProMfgdbDataSearchoptAdd()** add definitions of these options to the structure defined by **ProMfgdbSearchoptCreate()**.

Function **ProMfgdbSearchoptApplicDataAdd()** specifies the category and object type for which a user-specified 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.

Function **ProMfgdbSearchoptAllowedValueAdd()** add valid values available for a search option. Do not call this function if values are unavailable. Assign multiple values with multiple calls to this function.

Pro/ENGINEER tool database queries are in tree format, as shown in the following figure. Leaf nodes contain expressions, for example, units and type. The following functions extract data and conditions from the Pro/ENGINEER-native query for use in queries of third-party tool databases.
Function **ProMfgdbQuerynodeIsLeaf()** returns TRUE if a specified node is a leaf node, that is, a node that contains an expression.

Functions **ProMfgdbQuerynodeLeftChildGet()** and **ProMfgdbQuerynodeRightChildGet()** return either the left or right branch of a query node, respectively.

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()** returned FALSE.

Function **ProMfgdbQuerynodeExprGet()** returns the expressions for the specified query. The node interrogated must be a leaf node.
Function `ProMfgdbQueryTargetGet()` returns the target object 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 must gather information contained in the expressions. The following functions return attributes and operators contained in the returned expressions of a leaf node.

Function `ProMfgdbExprNameGet()` returns the name of the attribute contained in the specified expression. Function `ProMfgdbExprCategoryGet()` returns the category of the attribute contained in the specified expression. Functions `ProMfgdbExprValueGet()` and `ProMfgdbExprValueTypeGet()` return the value and value type contained in the specified expression, respectively. Function `ProMfgdbExprCompOpGet()` returns the comparison operator (=, <, >, and so on) contained in the specified expression.

Your Pro/TOOLKIT application must format data returned by the query (a “match”) for the Pro/ENGINEER search results dialog. First use function `ProMfgdbMatchAlloc()` to allocate memory for the structure used to store each match. Function `ProMfgdbMatchParamAdd()` adds a parameter to the match structure. Make multiple calls to this function to add multiple parameters to the match. Use ProArray utilities to create a ProArray of query matches. This array is an output argument of the function specified by `ProMfgdbSearchAction()`.
This chapter describes the Pro/TOLK1T functions that enable you to create auxiliary and custom NC sequences and CL commands.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auxiliary and Custom NC Sequences</td>
<td>32 - 2</td>
</tr>
<tr>
<td>CL Commands</td>
<td>32 - 3</td>
</tr>
</tbody>
</table>
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 32-1 Element Tree for Auxiliary NC Sequence

```
PRO_E_FEATURE_TREE
  |--- PRO_E_FEAT_TYPE
  |--- PRO_E_FEAT_NAME
  |--- PRO_E_OPERATION
  |--- PRO_E_RETRACT
  |--- PRO_E_CSYS
  |--- PRO_E_TOOL
  |--- PRO_E_MFG_PARAMS
     |--- PRO_E_MFG_PARAM
        |--- PRO_E_MFG_PARAM_NAME
        |--- PRO_E_MFG_PARAMVAL
```
The following figure shows the feature element tree for custom NC sequences.

**Figure 32-2: Element Tree for Custom NC Sequence**

```
PRO_E_FEATURE_TREE
   | PRO_E_FEAT_TYPE (PRO_E_FEAT_BLD_PATH)
   | PRO_E_REF_NCSEQ
   | PRO_E_NCL_COMMANDS
```

## 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()`.
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.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Step Objects</td>
<td>33 - 2</td>
</tr>
<tr>
<td>Visiting Process Steps</td>
<td>33 - 2</td>
</tr>
<tr>
<td>Process Step Access</td>
<td>33 - 2</td>
</tr>
<tr>
<td>Creating Process Steps</td>
<td>33 - 3</td>
</tr>
</tbody>
</table>
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 an assembly process step.

  The declaration is as follows:

  ```c
  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 ‘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 `ProFeatureElemtreeCreate()` with an input feature of type `ProProcstep`.
The following figure shows the element tree of a process step feature.

Figure 33-1: Element Tree of Process Step Feature

<table>
<thead>
<tr>
<th>PRO_E_FEATURE_TREE</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_FEATURE_TYPE</td>
</tr>
<tr>
<td>PRO_E_PROCESS_STEP_TYPE</td>
</tr>
<tr>
<td>PRO_E_COMPONENTS</td>
</tr>
<tr>
<td>PRO_E_DESCRIPTION</td>
</tr>
<tr>
<td>PRO_E_POSITION</td>
</tr>
<tr>
<td>PRO_E_SIMPLFD_REP</td>
</tr>
<tr>
<td>PRO_E_EXPLD_STATE</td>
</tr>
<tr>
<td>PRO_E_GEN_STEP_TYPE</td>
</tr>
<tr>
<td>PRO_E_GEN_STEP_REFS</td>
</tr>
<tr>
<td>PRO_E_TIME_ESTIMATE</td>
</tr>
<tr>
<td>PRO_E_COST_ESTIMATE</td>
</tr>
</tbody>
</table>

**Feature Elements**

The following table describes the tree elements in more detail.

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_FEATURE_TYPE</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>PRO_FEAT_PROCESS_STEP</td>
</tr>
<tr>
<td>PRO_E_PROCESS_STEP_TYPE</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>See ‘Types of Process Step’</td>
</tr>
<tr>
<td>PRO_E_COMPONENTS</td>
<td>PRO_VALUE_TYPE_SELECTION</td>
<td>Step components</td>
</tr>
</tbody>
</table>
### 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)

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_E_DESCRIPTION</td>
<td>PRO_VALUE_TYPE_WSTRING</td>
<td>Step description</td>
</tr>
<tr>
<td>PRO_E_POSITION</td>
<td>PRO_VALUE_TYPE_TRANSFORM</td>
<td>Position transformation</td>
</tr>
<tr>
<td>PRO_E_GEN_STEP_TYPE</td>
<td>PRO_VALUE_TYPE_WSTRING</td>
<td>General step type</td>
</tr>
<tr>
<td>PRO_E_GEN_STEP_REFS</td>
<td>PRO_VALUE_TYPE_SELECTION</td>
<td>General step references</td>
</tr>
<tr>
<td>PRO_E_SIMPLFD_REP</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>Simplified representation identifier</td>
</tr>
<tr>
<td>PRO_E_EXPLODE_STATE</td>
<td>PRO_VALUE_TYPE_INT</td>
<td>Explode state identifier</td>
</tr>
<tr>
<td>PRO_E_TIME_ESTIMATE</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
<td>Time estimate (greater than 0.0)</td>
</tr>
<tr>
<td>PRO_E_COST_ESTIMATE</td>
<td>PRO_VALUE_TYPE_DOUBLE</td>
<td>Cost estimate (any value)</td>
</tr>
</tbody>
</table>
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 'Assemblies' 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.
This chapter describes how to use Pro/TOOLKIT functions to manipulate the family table for an object.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Family Table Objects</td>
<td>34 - 2</td>
</tr>
<tr>
<td>Family Table Utilities</td>
<td>34 - 2</td>
</tr>
<tr>
<td>Visiting Family Tables</td>
<td>34 - 3</td>
</tr>
<tr>
<td>Operations on Family Table Instances</td>
<td>34 - 4</td>
</tr>
<tr>
<td>Operations on Family Table Items</td>
<td>34 - 6</td>
</tr>
</tbody>
</table>
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()**

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**.
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 34 - 6 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()`  
- `ProFaminstanceAdd()`  
- `ProFaminstanceCheck()`  
- `ProFaminstanceInit()`  
- `ProFaminstanceRemove()`  
- `ProFaminstanceCreate()`  
- `ProFaminstanceSelect()`  
- `ProFaminstanceMdlGet()`  
- `ProFaminstanceErase()`  
- `ProFaminstanceLock()`  
- `ProFaminstanceRetrieve()`  
- `ProFaminstanceGenericGet()`  
- `ProFaminstanceIsVerified()`  
- `ProFaminstanceIsExtLocked()`  
- `ProFaminstanceIsFlatState()`

The functions in this section enable you to programmatically manipulate instances that appear in a family table.

For the specified family table instance, the functions `ProFaminstanceValueGet()` and `ProFaminstanceValueSet()` enable you to modify the values of family table items (columns). As input, these functions require the instance and the item handles, 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.

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.

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.

```c
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, "Instance 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;  
    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.

```c
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

```c
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
", 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);
}
This chapter describes how to store and retrieve external data. External data enables a Pro/TOOLKIT application to store its own data in the Pro/ENGINEER database in a way that is invisible to the Pro/ENGINEER user.

This method is different from other means of storage accessible through the Pro/ENGINEER user interface, such as parameters and external objects. Each of these topics is described in its own chapter in this manual.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction to External Data</td>
<td>35 - 2</td>
</tr>
<tr>
<td>Storing External Data</td>
<td>35 - 4</td>
</tr>
<tr>
<td>Retrieving External Data</td>
<td>35 - 5</td>
</tr>
</tbody>
</table>
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 (MaxLength = 512 widecharacters)
- Stream (MaxSize = 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:

```c
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, create slot, change slot value, and delete 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 programatically. 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 assign 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.
The “Notify” feature enables your Pro/TOOLKIT application to trap certain types of events in Pro/ENGINEER and arrange that your own function is called before or after each such event. Pro/ENGINEER “notifies” your Pro/TOOLKIT application of these events. The notification happens whether the events arise from user interaction with Pro/ENGINEER or as a result of calls to the corresponding functions from a Pro/TOOLKIT application.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notify Classes</td>
<td>36 - 2</td>
</tr>
<tr>
<td>Using Notify</td>
<td>36 - 5</td>
</tr>
</tbody>
</table>
Notify Classes

The “notifiable” events in Pro/ENGINEER fall into the following classes:

- **File management events**—These include all 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.
  
  - 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 (all events)**—This is the same as the post-file management event above, except the callback function is called for all secondary file management events on models other than the one explicitly specified. 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.

- **File management failed 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).
• **Model modification events**—These are events that result in a change to the content of the model. They include both pre- and post-notifications of events, such as regeneration of the model itself or of a feature; other operations on features such as create, copy, delete, suppress; and operations on parameters. See the section 'Model Modification Events' for more information.

• **Context change events**—These callbacks are called after events that change the current context. These events are changing the directory and changing the window. The change window callback helps you keep track of which model is current in Pro/ENGINEER.

• **Graphics events**—These are callbacks 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.

• **Pro/NC output events**—You can arrange for notification (only after the event) 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.

• **CL command events**—These callbacks give you the ability to create auxiliary NC sequences with programmatically created CL commands.

• **Mold layout events**—These callbacks are invoked before entering a corresponding Mold Layout dialog.

• **Weld events**—These callbacks give you the ability to customize the results generated by Pro/ENGINEER when gathering info for weld operations.

• **Pro/MOLDESIGN events**—You can arrange for notification (only before the event) of the display of certain mold layout dialog boxes. This enables you to perform necessary changes to the dialog boxes before they are displayed.

To make the changes, use the **ProRmdtDlg*Set** functions, provided in the header file **ProRmdt.h**, within the callback functions.
Model Modification Events

Notification before or after feature regeneration should be used carefully, because your callback function might be 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 a 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.

However, sometimes it is desirable for the modifications to be made before the regeneration of a feature. For example, suppose you programmatically created a feature, B, based on the geometry of an existing feature, A, using Pro/TOOLKIT functions. Each time you modify and regenerate feature A, feature B will not be updated because Pro/ENGINEER does not know that feature B is based on A.

The notify function ProFeatureRegenPreAction() is called before the regeneration of each feature. Note that it is not permitted to make modifications to the model during the model regeneration.

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.
Using Notify

Function introduced:

- **ProNotificationSet()**

  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.

Interface Changes for Release 20 Forward

Due to implementation changes, the callback functions **ProMdlCreatePreAction()** and **ProMdlRetrievePreAction()** have new definitions. Using the new user interface introduced in Release 20, the notify function **ProMdlCreatePreAction()** will be invoked after you have specified the type and subtype of the model to be created in the New dialog box.

The callback function **ProFileOpenOKAction()** will be invoked after you have clicked OK in the New dialog box and Pro/ENGINEER has entered into the new mode with the new object allocated, but before control is passed back to the top-level menu of the new mode. For example, when you create a new drawing, the callback will be invoked after the Create Drawing dialog, which is used to specify the paper size, format, and model name.

The callback function **ProMdlRetrievePreAction()** is used to invoke an alternate interface for model retrieval. If this callback is specified, the usual Pro/ENGINEER user interface for name specification will not be invoked. This callback function will be invoked after you have selected Open in the File pull-down menu in the Pro/ENGINEER menu bar.
Notification Events

The following list shows the values of ProNotifyType, the names of the corresponding callback function typedefs, and the names of the Pro/TOOLKIT include files that contain them. The information is grouped by the classes described in the previous section.

### Pre-file Management Events

<table>
<thead>
<tr>
<th>Event type</th>
<th>Callback typedef</th>
<th>Include file</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_FILE_OPEN_OK</td>
<td>ProFileOpenOKAction</td>
<td>ProNotify.h</td>
</tr>
<tr>
<td>PRO_MDL_SAVE_PRE</td>
<td>ProMdlSavePreAction</td>
<td>ProMdl.h</td>
</tr>
<tr>
<td>PRO_MDL_COPY_PRE</td>
<td>ProMdlCopyPreAction</td>
<td>ProMdl.h</td>
</tr>
<tr>
<td>PRO_MDL_RENAME_PRE</td>
<td>ProMdlRenamePreAction</td>
<td>ProMdl.h</td>
</tr>
<tr>
<td>PRO_MDL_ERASE_PRE</td>
<td>ProMdlErasePreAction</td>
<td>ProMdl.h</td>
</tr>
<tr>
<td>PRO_MDL_PURGE_PRE</td>
<td>ProMdlPurgePreAction</td>
<td>ProMdl.h</td>
</tr>
<tr>
<td>PRO_MDL_DELETE_PRE</td>
<td>ProMdlDeletePreAction</td>
<td>ProMdl.h</td>
</tr>
<tr>
<td>PRO_MDL_RETRIEVE_PRE</td>
<td>ProMdlRetrievePreAction</td>
<td>ProMdl.h</td>
</tr>
<tr>
<td>PRO_MDL_CREATE_PRE</td>
<td>ProMdlCreatePreAction</td>
<td>ProMdl.h</td>
</tr>
<tr>
<td>PRO_MDL_START</td>
<td>ProMdlStartAction</td>
<td>ProMdl.h</td>
</tr>
</tbody>
</table>

### Post-file Management Events

<table>
<thead>
<tr>
<th>Event type</th>
<th>Callback typedef</th>
<th>Include file</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_MDL_SAVE_POST</td>
<td>ProMdlSavePostAction</td>
<td>ProMdl.h</td>
</tr>
<tr>
<td>PRO_MDL_COPY_POST</td>
<td>ProMdlCopyPostAction</td>
<td>ProMdl.h</td>
</tr>
<tr>
<td>PRO_MDL_RENAME_POST</td>
<td>ProMdlRenamePostAction</td>
<td>ProMdl.h</td>
</tr>
<tr>
<td>PRO_MDL_ERASE_POST</td>
<td>ProMdlErasePostAction</td>
<td>ProMdl.h</td>
</tr>
<tr>
<td>PRO_MDL_PURGE_POST</td>
<td>ProMdlPurgePostAction</td>
<td>ProMdl.h</td>
</tr>
<tr>
<td>PRO_MDL_DELETE_POST</td>
<td>ProMdlDeletePostAction</td>
<td>ProMdl.h</td>
</tr>
<tr>
<td>PRO_MDL_RETRIEVE_POST</td>
<td>ProMdlRetrievePostAction</td>
<td>ProMdl.h</td>
</tr>
<tr>
<td>PRO_MDL_CREATE_POST</td>
<td>ProMdlCreatePostAction</td>
<td>ProMdl.h</td>
</tr>
</tbody>
</table>

### Post-file Management -- All Events

<table>
<thead>
<tr>
<th>Event type</th>
<th>Callback typedef</th>
<th>Include file</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_MDL_SAVE_POST_ALL</td>
<td>ProMdlSavePostAllAction</td>
<td>ProMdl.h</td>
</tr>
<tr>
<td>PRO_MDL_COPY_POST_ALL</td>
<td>ProMdlCopyPostAllAction</td>
<td>ProMdl.h</td>
</tr>
<tr>
<td>PRO_MDL_ERASE_POST_ALL</td>
<td>ProMdlErasePostAllAction</td>
<td>ProMdl.h</td>
</tr>
<tr>
<td>PRO_MDL_DELETE_POST_ALL</td>
<td>ProMdlDeletePostAllAction</td>
<td>ProMdl.h</td>
</tr>
<tr>
<td>PRO_MDL_RETRIEVE_POST_ALL</td>
<td>ProMdlRetrievePostAllAction</td>
<td>ProMdl.h</td>
</tr>
</tbody>
</table>

### File Management Failed Events

<table>
<thead>
<tr>
<th>Event type</th>
<th>Callback typedef</th>
<th>Include file</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_MDL_DBMS_FAILURE</td>
<td>ProMdlDbmsFailureAction</td>
<td>ProMdl.h</td>
</tr>
<tr>
<td>PRO_MDL_CREATE_CANCEL</td>
<td>ProMdlCreateCancelAction</td>
<td>ProNotify.h</td>
</tr>
</tbody>
</table>

### Model Modification Events

<table>
<thead>
<tr>
<th>Event type</th>
<th>Callback typedef</th>
<th>Include file</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_DIM_MODIFY_VALUE_PRE</td>
<td>ProDimModifyValuePreAction</td>
<td>ProDimension.h</td>
</tr>
<tr>
<td>PRO_FEATURE_CREATE_PRE</td>
<td>ProFeatureCreatePreAction</td>
<td>ProFeature.h</td>
</tr>
<tr>
<td>PRO_FEATURE_CREATE_POST</td>
<td>ProFeatureCreatePostAction</td>
<td>ProFeature.h</td>
</tr>
<tr>
<td>Event type</td>
<td>Callback typedef</td>
<td>Include file</td>
</tr>
<tr>
<td>----------------------------</td>
<td>--------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>PRO_DIRECTORY_CHANGE_POST</td>
<td>ProDirectoryChangePostAction</td>
<td>ProNotify.h</td>
</tr>
<tr>
<td>PRO_WINDOW_CHANGE_POST</td>
<td>ProWindowChangePostAction</td>
<td>ProNotify.h</td>
</tr>
<tr>
<td>PRO_MDL_DISPLAY_PRE</td>
<td>ProMdlDisplayPreAction</td>
<td>ProMdl.h</td>
</tr>
<tr>
<td>PRO_MDL_DISPLAY_POST</td>
<td>ProMdlDisplayPostAction</td>
<td>ProMdl.h</td>
</tr>
<tr>
<td>PRO_NCSEQ_CL_POST</td>
<td>ProNcseqClPostAction</td>
<td>ProNotify.h</td>
</tr>
<tr>
<td>PRO_OPER_CL_POST</td>
<td>ProMfgoperClPostAction</td>
<td>ProNotify.h</td>
</tr>
<tr>
<td>PRO_NCL_COMMAND_EXPAND</td>
<td>ProClCommandExpandAction</td>
<td>ProClCmd.h</td>
</tr>
<tr>
<td>PRO_NCL_COMMAND_GET_LOC</td>
<td>ProClCommandGetLocAction</td>
<td>ProClCmd.h</td>
</tr>
<tr>
<td>PRO_RMDT_CREATE_IMM_Pre</td>
<td>ProRmdtCreateImmPreAction</td>
<td>ProRmdt.h</td>
</tr>
<tr>
<td>PRO_RMDT_BOUND_BOX_PRE</td>
<td>ProRmdtBoundBoxPreAction</td>
<td>ProRmdt.h</td>
</tr>
<tr>
<td>PRO_RMDT_CAV_LAYOUT_PRE</td>
<td>ProRmdtCavLayoutPreAction</td>
<td>ProRmdt.h</td>
</tr>
<tr>
<td>PRO_RMDT_CREATE_WP_PRE</td>
<td>ProRmdtCreateWpPreAction</td>
<td>ProRmdt.h</td>
</tr>
</tbody>
</table>
### Pro/MOLDESIGN events

<table>
<thead>
<tr>
<th>Event type</th>
<th>Callback typedef</th>
<th>Include file</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_RMDT_MATERIAL_PRE</td>
<td>ProRmdtMaterialPreAction</td>
<td>ProRmdt.h</td>
</tr>
<tr>
<td>PRO_RMDT_MBASE_SELECT_PRE</td>
<td>ProRmdtMBaseSelectPreAction</td>
<td>ProRmdt.h</td>
</tr>
</tbody>
</table>

### Weld Events

<table>
<thead>
<tr>
<th>Event type</th>
<th>Callback typedef</th>
<th>Include file</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_DRAWING_WELD_SYMPATH_GET</td>
<td>ProDrawingWeldSympathGetAction</td>
<td>ProNotify.h</td>
</tr>
<tr>
<td>PRO_DRAWING_WELD_GROUPIDS_GET</td>
<td>ProDrawingWeldGroupidsGetAction</td>
<td>ProNotify.h</td>
</tr>
<tr>
<td>PRO_DRAWING_WELD_SYMTEXT_GET</td>
<td>ProDrawingWeldSymtextGetAction</td>
<td>ProNotify.h</td>
</tr>
</tbody>
</table>

### Canceling a Notification

Function introduced:

- **ProNotificationUnset()**

  To cancel a notification, call **ProNotificationUnset()**.
Pro/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 Pro/WELDING features. For more information, refer to the Pro/WELDING section of Pro/ENGINEER Help.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Access to Weld Features</td>
<td>37 - 2</td>
</tr>
<tr>
<td>Customizing Weld Drawing Symbols</td>
<td>37 - 3</td>
</tr>
</tbody>
</table>
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.

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 'Notify' 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.

```c
#include "ProToolkit.h"
#include "ProParameter.h"
#include "ProUtil.h"
#include "ProNotify.h"
#include "ProParamval.h"
#include "TestError.h"
#include "UtilString.h"
#include <prodevelop.h>

int user_initialize()
{
    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_SYMETXT_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);
```
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);
        sprintf(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;
 }
 void user_terminate()
 {
 }

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.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>38 - 2</td>
</tr>
<tr>
<td>Simplified Representations in Session</td>
<td>38 - 3</td>
</tr>
<tr>
<td>Retrieving Simplified Representations</td>
<td>38 - 5</td>
</tr>
<tr>
<td>Creating and Deleting Simplified Representations</td>
<td>38 - 6</td>
</tr>
<tr>
<td>Extracting Information About Simplified Representations</td>
<td>38 - 9</td>
</tr>
<tr>
<td>Modifying Simplified Representations</td>
<td>38 - 13</td>
</tr>
<tr>
<td>Gathering Components by Rule</td>
<td>38 - 14</td>
</tr>
</tbody>
</table>
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 `ProSimprep`. As with other DHandles such as `ProFeature` and `ProGeomitem`, the `ProSimprep` 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 `ProSimprepdata` 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
- `ProSimprepActionType action_type`—The rule that controls the default treatment of items in the simplified representation
- `ProSimprepitem *items`—An array of assembly components and features and the actions applied to them in the simplified representation

A `ProSimprepitem` 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 `ProSimprepitem` has its own `ProSimprepAction` assigned to it. Each `ProSimprepAction` is a visible data structure that includes a variable of type `ProSimprepActionType`. 

**ProSimprepActionType** is an enumerated type that specifies the possible treatment of items in a simplified representation. The declaration is as follows:

```c
typedef enum pro_simprep_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. */
} ProSimprepActionType;
```

### Simplified Representations in Session

Functions introduced:

- **ProSolidSimprepVisit()**
- **ProSimprepInit()**
- **ProSimprepSelect()**
- **ProSimprepActivate()**
- **ProSimprepActiveGet()**
- **ProSimprepTypeGet()**

This section describes the utility functions that relate to simplified representations.

**ProSolidSimprepVisit()** 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:

```c
typedef enum pro_simprep_types
{
    PRO_SIMPREP_MASTER_REP,    /* Master Representation */
    PRO_SIMPREP_USER_DEFINED,  /* User Defined Representation */
    PRO_SIMPREP_GRAPH_REP,     /* Graphics only representation */
    PRO_SIMPREP_GEOM_REP       /* Geometry only representation */
    PRO_SIMPREP_SYMB_REP       /* Symbolic representation */
} ProSimprepType;
```

### Retrieving Simplified Representations

Function introduced:

- `ProAssemblySimprepRetrieve()`
- `ProGeomSimprepRetrieve()`
- `ProGraphicsSimprepRetrieve()`
- `ProSymbSimprepRetrieve()`
- `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 supply the name of the assembly. To retrieve an existing simplified representation, specify its name as the second argument to this function. Pro/ENGINEER retrieves that representation and any active submodels, and returns the handle to the simplified representation as a `ProAssembly` handle.
You can retrieve geometry, graphics, and symbolic simplified representations into session using the functions
\texttt{ProGeomSimprepRetrieve()}, \texttt{ProGraphicsSimprepRetrieve()} and \texttt{ProSymbSimprepRetrieve()} respectively. Like \texttt{ProAssemblySimprepRetrieve()}, these functions retrieve the simplified representation \textit{without} bringing the master representation into memory. For both functions, supply the name of the assembly whose simplified representation is to be retrieved. Both functions output the handle to the assembly. Note that neither function displays the simplified representation.

You can retrieve part simplified representations into memory using the function \texttt{ProPartSimprepRetrieve()}.

\section*{Creating and Deleting Simplified Representations}

Functions introduced:
\begin{itemize}
  \item \texttt{ProSimprepdataAlloc()}
  \item \texttt{ProSimprepCreate()}
  \item \texttt{ProSimprepDelete()}
\end{itemize}

\textbf{Note:} Pro/TOOLKIT does not support creation of part simplified representations.

To create a simplified representation, you must allocate and fill a \texttt{ProSimprepdata} structure by calling the function \texttt{ProSimprepdataAlloc()}. As input, the function requires the name of the new simplified representation, the \textit{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 \texttt{ProSimprepCreate()}. This function returns the \texttt{ProSimprep} handle for the new representation.

The function \texttt{ProSimprepDelete()} deletes a simplified representation from its model owner. The function requires only the \texttt{ProSimprep} 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.

```c
/*====================================================================*
FUNCTION: UserSimpRepCreate
PURPOSE:  Does the following:
- Creates a simplified representation
- Adds and delete some items
- Modifies the simplified representation
\*====================================================================*/
ProError UserSimpRepCreate()
{
  ProSimprepdata       *rep;
  ProSimprep          simp_rep;
  ProSimprepAction    sr_act;
  ProSimprepItem      item;
  ProError            status;
  ProFileName         message_file;
  ProName             name;
  ProMdl              assembly;
  ProMdlType          type;
  ProSimprepActionType 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 = ProSimprepdataAlloc (name, PRO_B_FALSE,
PRO_SIMPREP_INCLUDE, &rep);
status = ProSimprepCreate (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 = ProSimprepActionInit (PRO_SIMPREP_EXCLUDE, NULL,
    &sr_act);
for (i = 0; i < num; i++)
{
    status = ProSelectionAsmcomppathGet (sels[i], &comp_path);
    status = ProSimprepdataitemInit (comp_path.comp_id_table,
        comp_path.table_num, PRO_VALUE_UNUSED, &sr_act, &item );
    status = ProSimprepdataitemAdd (rep , &item);
}
status = ProSimprepdataSet (&simp_rep, rep);
/*---------------------------------------------*/
Set the new simplified rep to be the current representation,
and redisplay the assembly.

/*---------------------------------------------*/
status = ProSimprepActivate (assembly, &simp_rep);
status = ProSolidDisplay (assembly);
/*---------------------------------------------*/
Free the data and return.

/*---------------------------------------------*/
status = ProSimprepdataFree (&rep);
return (status);
Extracting Information About Simplified Representations

Functions introduced:

- ProSimprepdataGet()
- ProSimprepdataFree()
- ProSimprepdataDefltGet()
- ProSimprepdataNameGet()
- ProSimprepdataTmpvalGet()
- ProSimprepdataitemsVisit()

**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 ProSimprepdata structure, **ProSimprepdataGet()** fills out the ProSimprepdata structure. This function dynamically allocates storage for the data structure. When the memory is no longer needed, free it using the function **ProSimprepdataFree()**.

The **ProSimprepdataDefltGet()**, **ProSimprepdataNameGet()**, and **ProSimprepdataTmpvalGet()** functions return the associated values contained in the ProSimprepdata structure. They all take two arguments—the data structure to be queried, and the appropriate data structure for the type to be retrieved. **ProSimprepdataTmpvalGet()** retrieves the value of the temp field from the specified ProSimprepdata structure.

The function **ProSimprepdataitemsVisit()** visits all the items that make up the simplified representation. The action and filter functions both have **ProSimprepitem** as their first argument.

**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.

```c
/*================================================================*/
FUNCTION: UserSimpRepInfo
PURPOSE:  Print out the simplified representation information.
/*================================================================*/
int UserSimpRepInfo (  
    ProAppData  owner,
```
int dummy)
{
    ProSimprep simp_rep;
    ProSimprepdata *data;
    ProSimprepActionType 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 = ProSimprepSelect (*(ProSolid*)owner, &simp_rep);
    if (status == PRO_TK_NO_ERROR && simp_rep.id != -1)
    {
        status = ProSimprepdataGet (&simp_rep, &data);
        if (status == PRO_TK_NO_ERROR)
        {
            status = ProSimprepdataNameGet (data, w_sr_name);
            if (status == PRO_TK_NO_ERROR)
            {
                ProWstringToString (temp, w_sr_name);
                printf ("Simp rep name           : %s\n", temp);
            }
            status = ProSimprepdataDefltGet (data, &def_act);
            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");
                }
                printf ("Default action          : %s\n", temp);
            }
            status = ProSimprepdataTmpvalGet (data, &tmp_rep);
        }
    }

    // More code...
}
if (status == PRO_TK_NO_ERROR)
{
    printf ("Temporary simp rep : %s\n", 
            tmp_rep == PRO_B_TRUE ? "Yes" : "No");
}
ProSimprepdataItemsVisit (data, NULL, UserSimpRepVisAct, 
                        (ProAppData) owner);
status = ProSimprepdataFree (&data);
}
return status;
}/*================================================================*
FUNCTION: UserSimpRepVisAct
PURPOSE:  Highlight 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);
        if (status == PRO_TK_NO_ERROR)
            status = ProAsmcompMdlGet (&comp_path, &model);
        if (status == PRO_TK_NO_ERROR)
        {
            status = ProMdlIdGet (model, &id);
            status = ProMdlTypeGet (model, &mdl_type);
            if (status == PRO_TK_NO_ERROR)
            {
                status = ProModelitemInit (model, id, mdl_type, 
                                            &mdl_item);
            }
            if (status == PRO_TK_NO_ERROR)
            {
                status = ProSelectionAlloc (&comp_path, &mdl_item, 
                                             &mdl_item);
&selection);
}
else{
/*-----------------------------------------------*/
 Feature

*------------------------------*/
  status = ProFeatureInit (*owner, item->item_path.feat_id,
    &feature);
  if (status == PRO_TK_NO_ERROR)
  {
    status = ProFeatureSelectionGet (&feature, &selection);
  }
  if (status == PRO_TK_NO_ERROR)
  {
    ProSelectionHighlight (selection, PRO_COLOR_WARNING);
  }
  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 `ProSimprepdataSet()` 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 ProSimprepActionType to PRO_SIMPREP_EXCLUDE.

To Add Items

1. Get the ProSimprepdata structure, as described in the previous section.
2. Specify the action to be applied to the item with a call to the function ProSimprepActionInit().
3. Initialize a ProSimprepitem structure for the item by calling the function ProSimprepdataitemInit().
4. Add the item to the ProSimprepdata structure using the function ProSimprepdataitemAdd().
5. Reassign the ProSimprepdata structure to the corresponding ProSimprep object by calling ProSimprepdataSet().

To Remove Items

1. Get the ProSimprepdata structure handle.
2. Pass the ProSimprepdata handle and the ProSimprepitem handle for the item to be deleted to the function ProSimprepdataitemDelete().
3. Reassign the ProSimprepdata structure to the corresponding ProSimprep object by calling the function ProSimprepdataSet().

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.
This chapter describes the Pro/TOOLKIT functions that deal with drawings. Unless otherwise specified, Pro/TOOLKIT functions that operate on drawings use screen coordinates. See the ‘Geometry’ chapter to find out more about screen coordinates and how to convert to drawing coordinates (or paper coordinates).

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating Drawings from Templates</td>
<td>39 - 2</td>
</tr>
<tr>
<td>Diagnosing Drawing Creation Errors</td>
<td>39 - 4</td>
</tr>
<tr>
<td>Drawing Setup</td>
<td>39 - 6</td>
</tr>
<tr>
<td>Access Drawing Location in Grid</td>
<td>39 - 6</td>
</tr>
<tr>
<td>Drawing Sheets</td>
<td>39 - 7</td>
</tr>
<tr>
<td>Drawing Views and Models</td>
<td>39 - 10</td>
</tr>
<tr>
<td>Detail Items</td>
<td>39 - 20</td>
</tr>
<tr>
<td>Drawing Symbol Groups</td>
<td>39 - 50</td>
</tr>
<tr>
<td>Drawing Tables</td>
<td>39 - 55</td>
</tr>
<tr>
<td>Drawing Dimensions</td>
<td>39 - 65</td>
</tr>
</tbody>
</table>
Creating Drawings from Templates

Function Introduced:

- **ProDrawingFromTmpltCreate()**

  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)

  See Figure Example 1: Drawing Creation from a Template for example code using this function.

  Call the function **ProDrawingFromTmpltCreate()** to create a drawing from a template and 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: DrawingFromTmpltCreate.c

```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 UserDrawingCreateDemo()
{
    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 = ProDrawingFromTmpltCreate (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 \texttt{ProDwgcreateErrsFree()} frees an existing errors table.

Use the function \texttt{ProDwgcreateErrsCountGet()} to return the number of drawing creation errors in the table.

The function \texttt{ProDwgcreateErrTypeGet()} returns the type of a drawing creation error.

Use the function \texttt{ProDwgcreateErrViewNameGet()} returns the name of the template view where the error occurred.

The function \texttt{ProDwgcreateErrSheetGet()} returns the drawing sheet number where the error occurred.

Use the function \texttt{ProDwgcreateErrViewGet()} returns the drawing view where the error occurred. This function is valid for the following error types:

- \texttt{PRODWCRTERR_EXPLODE_DOESNT_EXIST}
- \texttt{PRODWCRTERR_MODEL_NOT_EXPLODABLE}
- \texttt{PRODWCRTERR_SEC_NOT_PERP}
- \texttt{PRODWCRTERR_NO_RPT_REGIONS}
- \texttt{PRODWCRTERR_FIRST_REGION_USED}
- \texttt{PRODWCRTERR_NOT_PROCESS_ASSEM}
- \texttt{PRODWCRTERR_TEMPLATE_USED}
- \texttt{PRODWCRTERR_SEC_NOT_PARALLEL}
- \texttt{PRODWCRTERR_SIMP_REP_DOESNT_EXIST}

\texttt{ProDwgcreateErrObjNameGet()} returns the name of the model invalid. This function is valid for the following error types:

- \texttt{PRODWCRTERR_SAVED_VIEW_DOESNT_EXIST}
- \texttt{PRODWCRTERR_X_SEC_DOESNT_EXIST}
- \texttt{PRODWCRTERR_EXPLODE_DOESNT_EXIST}
- \texttt{PRODWCRTERR_SEC_NOT_PERP}
- \texttt{PRODWCRTERR_SEC_NOT_PARALLEL}
- \texttt{PRODWCRTERR_SIMP_REP_DOESNT_EXIST}
Drawing Setup

Functions described further:

- ProInputFileRead()
- ProOutputFileWrite()

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 chapter for information on these functions.

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 Sheets

Functions introduced:

- ProDrawingSheetsCount()
- ProDrawingSheetTrfGet()
- ProDrawingSheetInfoGet()
- ProDrawingCurrentSheetGet()
- ProDrawingCurrentSheetSet()
- ProDrawingSheetCreate()
- ProDrawingSheetDelete()
- ProDrawingSheetsReorder()
- ProDrawingFormatGet()
- ProDrawingFormatAdd()
- ProDwgSheetRegenerate()

Drawing sheets are identified in Pro/TOOLKIT by the same sheet numbers the Pro/ENGINEER user sees.

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 ‘Coordinate Systems and Transformations’ chapter for more details on transformations.)

Use the function ProDrawingSheetInfoGet() to get other information about a specified sheet. This function fills the data structure prodrw_sheet_info structure, which is declared as follows:

```c
typedef struct prodrw_sheet_info
{
    ProPlotPaperSize type;
    int width;
    int height;
    int units;
    int orientation;
} Prodrw_sheet_info;
```

The units field takes one of the following values defined in pro_unit.h:

```
UNIT_INCH UNIT_FOOT UNIT_MM UNIT_CM UNIT_M UNIT_MCM
```
The function **ProDrawingCurrentSheetGet()** outputs the number of the current sheet in the specified drawing, and **ProDrawingCurrentSheetSet()** sets a sheet to be current. **ProDrawingSheetCreate()** adds a new sheet as the new last sheet. **ProDrawingSheetDelete()** deletes a specified drawing sheet.

The function **ProDrawingSheetsReorder()** assigns a new sheet number to a specified sheet, and renumbers the remaining sheets accordingly.

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 2: Listing Drawing Sheets**

The following example shows how to list the sheets in the current drawing.

```c
/*====================================================================*
FUNCTION : UsrListSheets()
PURPOSE  : Command to list sheet info in an information window
/*====================================================================*/
int UsrListSheets()
{
    ProDrawing drawing;
    int n_sheets, sheet;
    ProName wformat;
    ProCharName format;
    ProDrawingSheetInfo info;
    char *unit;
    FILE *fp;
    ProName wfname;
    char *fname = "sheets.txt";
    /*---------------------------------------------------------------*/
    /* Open a text file to contain the information to be displayed */
    fp = fopen(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.                                  *
         *--------------------------------------------------------------------*/
        ProDrawingSheetInfoGet(drawing, sheet, &info);
        /*--------------------------------------------------------------------*
         * 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                            *
         *--------------------------------------------------------------------*/
        switch(info.units)
        {
            case UNIT_INCH : unit = "inches";
            break;
            case UNIT_FOOT : unit = "feet";
            break;
            case UNIT_MM   : unit = "mm";
            break;
            case UNIT_CM   : unit = "cm";
            break;
            case UNIT_M    : unit = "M";
            break;
            case UNIT_MCM  : unit = "mcm";
            break;
            default : unit = "**unknown";
        }
        fprintf(fp, "Sheet %d\n", sheet);
        fprintf(fp, "    Width  : %d %s\n", info.width, unit);
        fprintf(fp, "    Height : %d %s\n", info.height, unit);
        fprintf(fp, "    Format : %s\n", format);
    }
    /*--------------------------------------------------------------------*
     * Close the file, and display it                                     *
     *--------------------------------------------------------------------*/
    fclose(fp);
    ProStringToWstring(wfname, fname);
    ProInfoWindowDisplay(wfname, NULL, NULL);
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()
- ProDrawingViewScaleIsUserdefined()
- ProDrawingViewScaleGet()
- ProDrawingScaleGet()
- ProDrawingViewDisplayGet()
- ProDrawingViewSolidGet()
- ProDrawingViewTransformGet()
- ProDrawingViewNameGet()
- ProDrawingViewZclippingGet()

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.
**ProDrawingViewScaleIsUserdefined()** returns a boolean value depending on whether the drawing has a user-assigned scale or not.

**ProDrawingViewScaleGet()** outputs the scale factor applied, even if the view is not a scaled view. **ProDrawingScaleGet()** gives the overall drawing scale that is applied to all unscaled views.

**ProDrawingViewDisplayGet()** outputs a structure describing 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.

**ProDrawingViewSolidGet()** provides the handle to the solid that is being displayed in the view. **ProDrawingViewTransformGet()** outputs the matrix that describes the transform between 3d solid coordinates and 2d screen coordinates for that 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.

### Example 3: 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.

```c
/*================================================================================*/
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 *mstyle;
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 = fopen(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
    /*-----------------------------------------------*/
    fprintf(fp,"View %d\n", view+1);
    fprintf(fp,"  Solid : %s.%s.%d\n", name, type,
        mdata.version);
    fprintf(fp,"  Sheet : %d\n", sheet);
    fprintf(fp,"  Lower left : %0.3f, %0.3f\n", 

outliner[0][0],
    outline[0][1]);
fprintf(fp,"    Upper right  : %0.3f, %0.3f\n",
    outline[1][0],
    outline[1][1]);
fprintf(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;
}
    fprintf(fp,"    Disp style   : %s\n", sstyle);
}  /*--------------------------------------------------------------------*/
Close the file, and display it
\*--------------------------------------------------------------------*/
fclose(fp);
    ProStringToWstring(wfname, fname);
    ProInfoWindowDisplay(wfname, NULL, NULL);
}

Modifying Views

Functions introduced:

- ProDrawingViewMove()
- ProDrawingViewDelete()
- ProDrawingViewScaleSet()
- ProDrawingScaleSet()
- ProDrawingViewZclippingSet()
- ProDwgViewRegenerate()

The function **ProDrawingViewMove()** moves not only the view you specify, but also any views that are its children. This function performs the same operation as the Pro/ENGINEER command View, Move View.

The function **ProDrawingViewDelete()** can either force the deletion of child views, or fail if there are children.

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 **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 **ProDwgViewRegenerate()** erases the displayed view of the current object, regenerates the view from the current drawing, then redispays the view.

### Drawing Models

Functions introduced:

- **ProDrawingSolidsCollect()**
- **ProDrawingSolidsVisit()**
- **ProDrawingSolidAdd()**
- **ProDrawingSolidDelete()**
- **ProDrawingSimprepsCollect()**
- **ProDrawingAsmsimprepAdd()**
- **ProDrawingAsmsimprepDelete()**
- **ProDrawingSolidReplace()**

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 **ProDrawingSimprepsCollect()**, **ProDrawingAsmsimprepAdd()**, and **ProDrawingAsmsimprepDelete()** 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.
Example 4: Replace Drawing Model Solid with a Solid

```c
#include <ProToolkit.h>
#include <ProDrawing.h>
#include <ProFaminstance.h>

int UserDrawingSolidReplace()
{
    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++)
        /* */
```
Creating Views

Function introduced:

- **ProDrawingGeneralviewCreate()**
- **ProDrawingProjectedviewCreate()**

These functions create general and projected drawing views.

Example 5: Creating Drawing Views

The following example code adds a new sheet to a drawing, and creates three views of a selected model.

```c
/*====================================================================*
FUNCTION : UsrSolidSelect()
PURPOSE : Utility to select a solid using the file browser and retrieve it if it is not already in session.
====================================================================*/
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;

/*--------------------------------------------------------------------*\n| 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);

/*--------------------------------------------------------------------*\n| Parse out the directory, and see if it is different from the current |
\*--------------------------------------------------------------------*/
different_dir = PRO_B_FALSE;
if(slash = strchr(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;
}

/*--------------------------------------------------------------------*\n| 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);

/*--------------------------------------------------------------------*\n| If the solid is already in session, return with it                  |
\*--------------------------------------------------------------------*/
if(ProMdlInit(wroot, mtype, (ProMdl*)solid) == PRO_TK_NO_ERROR)
    return(0);

/*--------------------------------------------------------------------*\
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

```c
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

```c
ProDrawingViewOutlineGet(drawing, view, outline);
```

Create a projected view to the right of the general view

```c
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

```c
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:

- `ProDrawingViewIsBackground()`
- `ProDrawingBackgroundViewGet()`

Views are assigned not only to the solid views that the user creates, but also to each sheet. The view for a 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.

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 are five types of detail items available in Pro/TOOLKIT:

• 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 that can contain notes
  and symbol instances, as well as draft entities. You create them
  in Pro/ENGINEER using the command Detail>Tools>Group.
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 five types of item are:

- PRO_DRAFT_ENTITY
- 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()
- ProDrawingDtlentityVisit()
- ProDrawingDtlnotesCollect()
- ProDrawingDtlnoteVisit()
- ProDrawingDtlsymdefsCollect()
- ProDrawingDtlsymdefVisit()
- ProDrawingDtlsyminstsCollect()
- ProDrawingDtlsyminstVisit()
- ProDrawingDtlgroupsCollect()
- ProDrawingDtlgroupVisit()

Each of the five types of detail item has its own collect function, and its own visit function. For entities, notes, symbol instances, and groups, these find all the items in a specified sheet. Symbol definitions do not belong to a specific sheet, so for these the collect and visit functions find all the items in the drawing.
Notes and draft entities can also exist inside symbol definitions, so the corresponding collect and visit functions have an optional extra argument to specify the owning symbol definition.

Displaying Detail Items

Functions introduced:

- `ProDtentityDraw()`
- `ProDtentityErase()`
- `ProDtlnoteDraw()`
- `ProDtlnoteErase()`
- `ProDtlnoteShow()`
- `ProDtlnoteRemove()`
- `ProDtlsyminstDraw()`
- `ProDtlsyminstErase()`
- `ProDtlsyminstShow()`
- `ProDtlsyminstRemove()`

Each of the three 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 Draw function draws the detail item temporarily, so that it is removed 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 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.
Creating, Modifying and Reading Detail Items

Functions introduced:

- ProDtlentityCreate()
- ProDtlentityDataGet()
- ProDtlentityDelete()
- ProDtlentityModify()
- ProDtlnoteCreate()
- ProDtlnoteDataGet()
- ProDtlnoteDelete()
- ProDtlnoteLineEnvelopeGet()
- ProDtlnoteModify()
- ProDtlsymdefCreate()
- ProDtlsymdefDataGet()
- ProDtlsymdefDelete()
- ProDtlsymdefModify()
- 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 Text 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.

- `ProDtlnentitydata`—A draft entity
- `ProCurvedata`—The 2d geometry of the entity (described in the ‘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:

- `ProDtlnentitydataAlloc()`—allocate the entity data (see the section on Draft Entity Data).
- `ProCurvedataAlloc()`—Allocate memory for a curve structure
- `ProLinedataInit()`—Set the curve structure to describe the required line by initializing a line data structure.

The function `ProDtlnentitydataCurveSet()`—Add the curve data to the entity data (see the section on Draft Entity Data).
**ProDtlnoteErase()**—Temporarily undraw the note (see the section on Displaying Detail Items).

The function **ProDtlnoteDataGet()**—Get the data for the note (see the section on Creating, Modifying and Reading Detail Items).

**ProDtlnotedataColorSet()**—Modify the color in the data (see the section on Detail Note Data).

The function **ProDtlnoteModify()**—Use the modified data to modify the note itself (see the section on Creating, Modifying and Reading Detail Items).

**ProDtlnoteDraw()**—Redraw 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.
Draft Entity Data

Functions introduced:

- ProDtentitydataAlloc()
- ProDtentitydataFree()
- ProDtentitydataIdGet()
- ProDtentitydataCurveGet()
- ProDtentitydataCurveSet()
- ProDtentitydataColorGet()
- ProDtentitydataColorSet()
- ProDtentitydataFontGet()
- ProDtentitydataFontSet()
- ProDtentitydataWidthGet()
- ProDtentitydataWidthSet()
- ProDtentitydataViewGet()
- ProDtentitydataViewSet()
- ProDtentitydatasConstruction()
- ProDtentitydatasConstructionSet()
- ProDtentitydatasHidden()
- ProDtentitydatasHiddenSet()

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.
**ProDtlentitydataIsConstruction()** and **ProDtlentitydataConstructionSet()** get and set the flag that controls whether the entity is created normal or as a construction entity.

**ProDtlentitydataIsHidden()** and **ProDtlentitydataHiddenSet()** get and set the flag that controls whether the entity is created normal or as a hidden entity.

### Example 6: 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.

```c
#include <ProToolkit.h>
#include <ProDtlentity.h>

/*====================================================================*
 FUNCTION : UsrLineentityCreate()
 PURPOSE : Utility to create a line entity between specified points
 /*====================================================================*/
int UsrLineentityCreate(
    ProDrawing drawing,
    ProVector start,
    ProVector end,
    ProColortype color)
{
    ProDtlentitydata edata;
    ProCurvedata *curve;
    ProDtlentity entity;
    ProColor entity_color;
    int cur_sheet;
    ProView view;

    /*--------------------------------------------------------------------*\
    Allocate data for the draft entity
    *--------------------------------------------------------------------*/
    ProDtlentitydataAlloc(drawing, &edata);

    /*--------------------------------------------------------------------*\n    Allocate and initialize a curve description
    *--------------------------------------------------------------------*/
    ProCurvedataAlloc(&curve);
    ProLinedataInit(start, end, curve);

    /*--------------------------------------------------------------------*\n    Use the curve description to set the entity data curve
    *--------------------------------------------------------------------*/
    ProDtlentitydataCurveSet(edata, curve);

    /*--------------------------------------------------------------------*\n    Set the view to the current background view for this sheet
    *--------------------------------------------------------------------*/
    ProViewSetToBackground(drawing, &view);
    ProViewAlloc(drawing, &view);
    ProViewSet(drawing, &view);
    /*--------------------------------------------------------------------*\n    Set the entity color
    *--------------------------------------------------------------------*/
    ProColorAlloc(drawing, &entity_color);
    ProColortypeSet(entity_color, color);
    /*--------------------------------------------------------------------*\n    Create the draft line
    *--------------------------------------------------------------------*/
    ProDtlentityCreate(drawing, &entity, start, end, &entity_color);
    /*--------------------------------------------------------------------*\n    End of function
    *--------------------------------------------------------------------*/
    return 0;
}
```

39 - 28

ProTOOLKIT User's Guide
ProDrawingCurrentSheetGet (drawing, &cur_sheet);
ProDrawingBackgroundViewGet (drawing, cur_sheet, &view);
ProDtltentitydataViewSet (edata, view);

*--------------------------------------------------------------------*
// Set the color to the specified Pro/ENGINEER predefined color
*--------------------------------------------------------------------*/
entity_color.method = PRO_COLOR_METHOD_TYPE;
entity_color.value.type = color;
ProDtltentitydataColorSet(edata, &entity_color);

*--------------------------------------------------------------------*
// Create the entity
*--------------------------------------------------------------------*/
ProDtltentityCreate(drawing, NULL, edata, &entity);

*--------------------------------------------------------------------*
// Display the entity
*--------------------------------------------------------------------*/
ProWindowRepaint (PRO_VALUE_UNUSED);

*--------------------------------------------------------------------*
// Release the entity data memory
*--------------------------------------------------------------------*/
ProDtltentitydataFree(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;
}
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 replace 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 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()`
- `ProDtlattachFree`
- `ProDtlattachGet()`

The opaque data structure `ProDtlattach` is used for two things:
• 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 ProDtattach to describe its attachment in the drawing, and may contain any number of ProDtattach objects describing the leaders.

ProDtattachAlloc() allocates and fills a new attachment. The inputs are:

• type—the type of attachment to the drawing view
  PARAMETRIC - the attachment is to a point on a surface or an edge of a solid in a drawing view.
  or
  FREE—the attachment to a 2d location on the drawing view. If type is FREE, the view may be set to a drawing view, meaning that the attachment will move when the drawing view is moved. This may be NULL.

• view—If the type is FREE, the view may be set to a drawing view, meaning that the attachment will move when the drawing view is moved. This may be NULL.

• location—If the type is FREE, this is the location of the attachment on the drawing, in screen coordinates.

• attach_point —If the type is PARAMETRIC, this ProSelection structure identifies the surface or edge point of the solid. If you are building this structure using ProSelectionAlloc() you should also set the location on the edge or surface using ProSelectionUvParamSet(), and the drawing view in which the attachment should be using ProSelectionViewSet().

ProDtattachFree() frees an attachment, and ProDtattachGet() unpacks the above information for an existing attachment.
Detail Note Data

Functions introduced:

- `ProDtlnotedataAlloc()`  
- `ProDtlnotedataFree()`  
- `ProDtlnotedataIdGet()`  
- `ProDtlnotedataLineAdd()`  
- `ProDtlnotedataLinesSet()`  
- `ProDtlnotedataLinesCollect()`  
- `ProDtlnotedataMirrorSet()`  
- `ProDtlnotedatalmsMirrored()`  
- `ProDtlnotedataColorGet()`  
- `ProDtlnotedataColorSet()`  
- `ProDtlnotedataAttachmentGet()`  
- `ProDtlnotedataAttachmentSet()`  
- `ProDtlnotedataLeadersCollect()`  
- `ProDtlnotedataLeadersSet()`  
- `ProDtlnotedataLeaderAdd()`  
- `ProDtlnotedataElbowlengthGet()`  
- `ProDtlnotedataElbowlengthSet()`  
- `ProDtlnotedataAngleGet()`  
- `ProDtlnotedataAngleSet()`  
- `ProDtlnotedataJustifGet()`  
- `ProDtlnotedataJustifSet()`  
- `ProDtlnotedatalssDisplayed()`  
- `ProDtlnotedataDisplayedSet()`

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 and 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()** get and set the note vertical and horizontal justification. Vertical justification applies only to notes in drawing tables.

**ProDtlnotedataIsDisplayed()** and **ProDtlnotedataDisplayedSet()** get and set the flag that controls whether the note is visible or not.

**Example 7: 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.

```c
/*================================================================================*/
FUNCTION : UsrSurfNoteCreate()
PURPOSE : Utility to create a note that documents the surface name or id.
/*================================================================================*/
int UsrSurfNoteCreate(  
    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;  
    ProDtlnoteAttach 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);  
    }  
```
\*--------------------------------------------------------------------*\nDrawings
ProStringToWstring(font, "font");
proDtlnotextFontSet(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
    sprintf(name, "id %d", modelitem.id);
    sprintf(str, "Surface %s", name);
    ProStringToWstring(wstr, str);
    ProDtlnotextStringSet(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);
}
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 a '&'. 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.
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()`
- `ProDtlsymdefdataHeightGet()`
- `ProDtlsymdefdataPathSet()`
- `ProDtlsymdefdataPathGet()`

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 length of the elbow  
that connects each leader to the symbol instance. If you do not call  
ProDtlsymdefdataElbowSet() when creating a symbol  
definition, there will be no elbow.  

ProDtlsymdefdataTextangfixedGet() and  
ProDtlsymdefdataTextangfixedSet() get and set whether the  
angle of text in the symbol is fixed.  

ProDtlsymdefdataHeightGet() gives the height of the symbol  
definition. Pro/ENGINEER calculates the height; it is not possible  
to set the height of a symbol definition.  

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.

Creating a Symbol Definition

The notes and draft entities that are contained by a symbol  
definition are created using ProDtlnoteCreate() 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  
ProDtlnoteCreate() 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 8: 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.

```c
/*====================================================================*/
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(PROSYMDEFFATTACHTYPE_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

```c
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

```c
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

```c
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.
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);
/*-----------------------------------------------*/
Create the note

ProDtlnoteCreate(symdef->owner, symdef, ndata, &note);

/*-----------------------------------------------*/
Free the memory for the note description

ProDtlnotedataFree(ndata);
return(1);
}

/*====================================================================*
FUNCTION : UsrSymdefLineAdd() PURPOSE : Utility to add a line entity to a symbol definition
*/
int UsrSymdefLineAdd(
    ProDtlsymdef *symdef, 
    ProVector start, 
    ProVector end, 
    ...
Retrieving a Symbol Definition from Disk

Function introduced:

- **ProDrawingDtsymdefRetrieve()**
- **ProDrawingSystemDtsymdefRetrieve()**

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()
• ProDtlsyminstdataHeightSet()
• ProDtlsyminstdataHeightGet()
• ProDtlsyminstdataDisplayedSet()
• ProDtlsyminstdataDisplayed()
• 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.)

ProDtlsyminstdataHeightSet() and
ProDtlsyminstdataHeightGet() set and get the height of a
symbol instance. ProDtlsyminstdataHeightSet() will work only
of the height type of the symbol definition is
PRODTSYMDEFHGHHTYPE_VARIABLE. See also function
ProDtlsymdefdataHeighttypeSet() in the section on Symbol
Definition Data.)

ProDtlsyminstdataDisplayedSet() and
ProDtlsyminstdatasDisplayed() set and get the flag that
controls whether or not the instance is visible.
Manipulate ProDtlvartext 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 ProDtlvartext 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.

Example 9: Create Free Instance of Symbol Definition

The following example shows a function which creates a free instance of a symbol definition.

```c
/*====================================================================*\
FUNCTION : UsrSymInstCreate()
PURPOSE  : Create a symbol with no leaders at a specified location
\*====================================================================*/
int UsrSymInstCreate(
    ProDrawing drawing,
    ProDtlsymdef *definition,
    ProVector pos)
{
    ProDtlsyminstdata sdata;
    ProDtlattach attach;
    ProDtlsyminst syminst;
    /*---------------------------------------------*/
    Allocate the symbol instance description
    \*---------------------------------------------*/
    ProDtlsyminstdataAlloc(drawing, &sdata);
    /*---------------------------------------------*/
```

Drawings
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);
}

**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:

```c
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:

- **ProDtlsyminstSymgroupsCollect()**

  The function **ProDtlsyminstSymgroupsCollect()** 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:

- **ProDtlsymgroupSubgroupsCollect()**
- **ProDtlsymgroupDataGet()**
- **ProDtlsymgroupdataNameGet()**
- **ProDtlsymgroupdataItemsCollect()**
- **ProDtlsymgroupParentGet()**
- **ProDtlsymgroupLevelIsExclusive()**

  The function **ProDtlsymgroupSubgroupsCollect()** returns the names of all subgroups stored in the symbol definition at the indicated level.

  Use the function **ProDtlsymgroupDataGet()** to get the data for the group stored in the symbol definition.

  The function **ProDtlsymgroupdataNameGet()** returns the name of the group using the symbol group data as input, while the function **ProDtlsymgroupdataItemsCollect()** 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 **ProDtlsymgroupParentGet()** 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.

### Detail Group Data

Functions introduced:

- **ProDtlgroupdataAlloc()**
- **ProDtlgroupdataFree()**
- **ProDtlgroupdataIdGet()**
- **ProDtlgroupdataNameGet()**
- **ProDtlgroupdataIsDisplayed()**
- **ProDtlgroupdataDisplayedSet()**
- **ProDtlgroupdataItemAdd()**
- **ProDtlgroupdataItemsSet()**
- **ProDtlgroupdataItemsCollect()**

**ProDtlgroupdataAlloc()** and **ProDtlgroupdataFree()** allocate and free a detail group description in the form of a ProDtlgroup object. **ProDtlgroupdataAlloc()** also sets the name of the group. **ProDtlgroupdataIdGet()** outputs 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 description, described in terms of the ProDtlitem object. **ProDtlgroupdataItemsSet()** puts an array of items into a group description, replacing any existing items. **ProDtlgroupdataItemsCollect()** outputs an array of the items in a group description.

### Example 10: Create New Group of Items

The following example shows a command which creates a group from a set of selected detail items.

```c
/*====================================================================*/
```
FUNCTION : UsrGroupCreate() PURPOSE :
Command to create a new group with selected items
/*====================================================================*/
int UsrGroupCreate()
{
    ProError status;
    ProSelection *sel;
    int n_sel, s;
    ProModelitem *items;
    ProDrawing drawing;
    ProName name;
    ProDtlgroupdata gdata;
    ProDtlgroup group;
    /*--------------------------------------------------------------------*/
    Select notes, draft entities, symbol instances
    /*--------------------------------------------------------------------*/
    ProMessageDisplay(msgfil, "USER Select the items to belong to the group");
    status = ProSelect("any_note,draft_ent,dtl_symbol", -1,
                       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++)
        ProSelectionModelitemGet(sel[s], &items[s]);
    /*--------------------------------------------------------------------*/
    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);
    /*--------------------------------------------------------------------*/
    Allocate group data
    /*--------------------------------------------------------------------*/
    ProDtlgroupDataAlloc(drawing, name, &gdata);
    /*--------------------------------------------------------------------*/
    Set the group items
    /*--------------------------------------------------------------------*/
    ProDtlgroupDataItemsSet(gdata, items);
    /*--------------------------------------------------------------------*/
    Create the group
    /*--------------------------------------------------------------------*/
    ProDtlgroupCreate(drawing, gdata, &group);
    /*--------------------------------------------------------------------*/
    Free the data
}
A drawing table is identified by the DHandle ProDwgtable which is typedef'd to, and inherited from, ProModelitem. The “type” field in ProDwgtable has the value PRO_DRAW_TABLE.

Selecting Drawing Tables and Cells

Functions 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 then call ProSelectionModelitemGet() to get the table handle, and the special function ProSelectionDwgtblcellGet() which outputs 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.
Reading Drawing Tables

Functions introduced:

- ProDrawingTableVisit()
- ProDrawingTablesCollect()
- ProDwgtableInfoGet()
- ProDwgtableColumnsCount()
- ProDwgtableRowsCount()
- ProDwgtableColumnSizeGet()
- ProDwgtableRowSizeGet()
- ProDwgtableCellNoteGet()
- ProDwgtableIsFromFormat()

ProDrawingTableVisit() visits all the tables in a specified drawing; it conforms to the usual standard for visit functions. ProDrawingTablesCollect() is an alternative, and outputs 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 which 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:

```c
int rotation;                    // The number of 90 degree turns clockwise
double seg_origin[3];           // The screen coordinates of the top left corner of the segment.
int nrows;                      // The number of rows.
int ncols;                      // The number of columns.
double outline[2][3];           // The outline of the segment.
double seg_char_height;         // The text height used for the segment
double table_char_height;       // The text height used for the drawing
double char_width;              // The character width factor.
```
You can also count the rows and columns of an entire table (including all the segments) using `ProDwgtableRowsCount()` and `ProDwgtableColumnsCount()`.

The functions `ProDwgtableColumnSizeGet()` and `ProDwgtableRowSizeGet()` give the column width and row height for a specified table segment column or row.

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()` outputs the handle to the detail note that represents the text in a specified table cell.

The function `ProDwgtableIsFromFormat()` tells you whether a table was added to the table as a result of importing a format.

**Drawing Table Segments**

Functions introduced:
- `ProDwgtableSegMove()`
- `ProDwgtableSegCount()`
- `ProDwgtableSegSheetGet()`

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 function the coordinates of the target position in format x, y, z=0. The function moves the table segment to the target position.

Determine the number of segments in a table with the function `ProDwgtableSegCount()`. Pass the function the name of the table and it returns the number of table segments.

Use the function `ProDwgtableSegSheetGet()` to determine which sheet contains a specified drawing table segment.
Creating a Table

Functions introduced:

- `ProDwgtabledataAlloc()`
- `ProDwgtabledataOriginSet()`
- `ProDwgtabledataSizetypeSet()`
- `ProDwgtabledataColumnsSet()`
- `ProDwgtabledataRowsSet()`
- `ProDrawingTableCreate()`
- `ProDwgtableTextEnter()`
- `ProDwgtableDisplay()`

The information required to build a table is contained in an opaque data structure called `ProDwgtabledata` which 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 given in screen coordinates, or as number of text characters. It is usually much more convenient to specify as numbers of characters.

`ProDwgtabledataColumnsSet()` sets the widths of the columns, and the default justifications of text in the columns. `ProDwgtabledataRowsSet()` sets the heights of rows.

`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.
Table Modification

Functions introduced:

- `ProDwgtableRowAdd()`
- `ProDwgtableColumnAdd()`
- `ProDwgtableRowDelete()`
- `ProDwgtableColumnDelete()`
- `ProDwgtableCellsMerge()`
- `ProDwgtableCellsRemesh()`
- `ProDwgtableRotate()`

`ProDwgtableRowAdd()` and `ProDwgtableColumnAdd()` can add a row or a column before or after any existing row or column. An input argument specifies the size of the row or column. Another input specifies whether the change should be immediately displayed. When making many changes to a table, it is advisable not to display them immediately, but to use `ProDwgtableDisplay()` to update the display afterwards.

`ProDwgtableRowDelete()` and `ProDwgtableColumnDelete()` delete any specified row or column, including removing the text from the effected cells.

`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 (when using `ProDwgtableTextEnter()`, or other calls to `ProDwgtableCellsMerge()`, for example) by the row and column number of its original top left cell. Rows below, and columns to the right, retain their original numbers. The function `ProDwgtableCellsRemesh()` unmerged all the merged cells in a specified range of rows and columns.

`ProDwgtableRotate()` rotates a table 90 degrees clockwise.

Example 11: Creation of Table Listing Datum Points

The following example shows a command which creates a drawing table listing the datum points in a model shown in a drawing view.

```c
/*---------------------------------------------*/
Data structure for information needed about datum points

typedef struct point_t {
    ProFeature feature;
    ProVector position;
    ProName name;
};
```
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;
    ProMdldata mdata;
    ProCharLine line;

    ProMessageDisplay(msgfil,"USER Select csys");
    status = ProSelect("csys",1,NULL,NULL,NULL,&sel,&n_sel);
    if(status != PRO_TK_NO_ERROR || n_sel < 1)
        return(0);

    ProSelectionModelitemGet(sel[0], &csys_geom);
    ProSelectionAsmcomppathGet(sel[0], &csys_comppath);
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 csys 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
 ------------------------------------*/
UsrPointsCollect(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);
    sprintf(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)
{
    sprintf(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);
Drawings

ProPntTrfEval(points[p].position, to_csys, points[p].position);
/*-----------------------------------------------------------*
Add the XYZ to column 3,4,5
*/
sprintf(name, "%.3f", points[p].position[0]);
UsrTableTextAdd(&table, 3, p+3, name);
sprintf(name, "%.3f", points[p].position[1]);
UsrTableTextAdd(&table, 4, p+3, name);
sprintf(name, "%.3f", points[p].position[2]);
UsrTableTextAdd(&table, 5, p+3, name);
} /*-----------------------------------------------------------*

Display the table
="/--------------------------------------------------------------------*/
ProDwgtableDisplay(&table);
return(1);
} /*--------------------------------------------------------------------*
FUNCTION : UsrTableTextAdd()
PURPOSE  : Utility to add one text line to
           a table cell
="/--------------------------------------------------------------------*/
int UsrTableTextAdd(
    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 : UsrPointAction()
PURPOSE  : Visit action function called for each datum point
="/--------------------------------------------------------------------*/
ProError UsrPointAction(
    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);

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);
Repeat Regions

Functions introduced:

- `ProDwgtableCellIsComment()`
- `ProDwgtableCellComponentGet()`
- `ProDrawingTablesUpdate()`

The functions `ProDwgtableCellIsComment()` and `ProDwgtableCellComponentGet()` apply to repeat regions in tables produced by Pro/REPORT. `ProDwgtableCellIsComment()` simply tells you whether a cell in a repeat region contains a comment; `ProDwgtableCellComponentGet()` identifies the assembly component that is being shown in a cell in a repeat region.

The function `ProDrawingTablesUpdate()` updates repeat regions in all tables to account for changes to the model(s). It is equivalent to the Pro/ENGINEER command `Table, Repeat Region, 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 chapter on Dimensions and Relations; read the explanation of ProDimension at the start of that chapter 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 chapter on Dimensions and Relations, can be used to find all the dimensions stored in a drawing.
The following dimension functions, described in the chapter on Dimensions and Relations, also support dimensions stored in a drawing:

- **ProDimensionShow()**
- **ProDimensionErase()**

### Drawing Dimension Attachments and Dimension Creation

**Functions Introduced:**

- **ProDrawingDimAttachsGet()**
- **ProDrawingDimCreate()**

The function **ProDrawingDimAttachsGet()** provides information about what entities a dimension is attached to.

**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 itself
- **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.

**ProDimSense** is declared in header file **ProDimension.h**. This is the declaration:

```c
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 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 a structure 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.

To display the new dimension, call the function **ProDimensionShow()**.

### 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 12: 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.

```c
/*====================================================================*
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 this 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

```c
UsrPointsCollect(solid, &points);
ProArraySizeGet(points, &n_points);
if(n_points < 1)
    return(0);
```

Calculate where the csys is located on the drawing

```c
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

```c
ProDrawingViewOutlineGet(drawing, view, outline);
```

Allocate the attachment arrays

```c
ProArrayAlloc(2, sizeof(ProSelection), 1, (ProArray*)&attachments);
ProArrayAlloc(2, sizeof(ProDimSense), 1, (ProArray*)&senses);
```

For each datum point...

```c
for(p=0;p<n_points;p++)
{
```

Calculate the position of the point on the drawing

```c
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

```c
    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

```c
    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);
ProDimensionShow(&dimension, view, drawing, NULL);
 /*--------------------------------------------------------------------*/
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);
ProDimensionShow(&dimension, view, drawing, NULL);
 /*--------------------------------------------------------------------*/
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, &vbase_dim);
```c
    csys_3dpos, &hbase_dim);

    else
        ProDrawingDimToOrdinate(drawing, &dimension, &hbase_dim);
    /*----------------------------------------------------------*/
    /* Free the attachment selection objects */
    ProSelectionFree(&attachments[0]);
    ProSelectionFree(&attachments[1]);
    }
    return(1);
}

/*====================================================================*
FUNCTION : UsrPointAction()
PURPOSE  : Visit action function called for each datum point
\*====================================================================*/
ProError UsrPointAction(
    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 : 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,
ProGeomItem **points)
{
    /*--------------------------------------------------------------------*/
    Allocate the array
    /*--------------------------------------------------------------------*/
    ProArrayAlloc(0, sizeof(ProGeomItem), 1, (ProArray*)points);
    /*--------------------------------------------------------------------*/
    Visit the features
    /*--------------------------------------------------------------------*/
    ProSolidFeatVisit(solid, UsrFeatureAction, NULL, points);
}

Other Drawing Dimension Properties

Functions introduced:

- **ProDrawingDimIsAssociative()**
- **ProDrawingDimensionIsDisplayed()**
- **ProDrawingDimensionIsToleranceDisplayed()**
- **ProDrawingDimensionViewGet()**
- **ProDrawingDimSwitchView()**
- **ProDrawingDimensionPosGet()**
- **ProDrawingDimensionMove()**

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, and `ProDrawingDimensionMove()` allows you to move the dimension text to a new position on the drawing.
The functions in this chapter enable you to access, modify, and delete cross sections, and create planar cross sections.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Listing Cross Sections</td>
<td>40 - 2</td>
</tr>
<tr>
<td>Extracting Cross-Sectional Geometry</td>
<td>40 - 2</td>
</tr>
<tr>
<td>Visiting Cross Sections</td>
<td>40 - 5</td>
</tr>
<tr>
<td>Creating and Modifying Cross Sections</td>
<td>40 - 5</td>
</tr>
<tr>
<td>Mass Properties of Cross Sections</td>
<td>40 - 6</td>
</tr>
</tbody>
</table>
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 get the type of the specified cross section.

Extracting Cross-Sectional Geometry

Functions introduced:

- ProXsecGeometryRetrieve()
- ProXsecDisplay()
- ProXsecRegenerate()

Functions Superseded:

- prodb_xsec_component()
- prodb_regen_xsec()
- prodb_display_xsec()

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.

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()**.

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.

**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.

```
user_List_Xsecs()
{
  Prohandle   h_obj, h_xsec, xsec_geom, h_face, h_cont;
  wchar_t     w_name[PRODEV_NAME_SIZE];
  char        name[PRODEV_NAME_SIZE], path[200];
  int         n_components, c, n_faces, n_conts;
  int         memb_num, *memb_id_tab;
  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))
  {
```

Cross Sections
Get the name.

if (prodb_xsec_name (h_obj, h_xsec, w_name) == 0)
    continue;
printf ("Cross section %s\n", pro_wstr_to_str (name,
    w_name));

Regenerate, so you can extract the geometry.

prodb_regen_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);

    List the component faces and contours.

    h_face = xsec_geom;
    n_faces = 1;
    do
    {
        printf("    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))
            printf("        Contour %d\n", n_conts++);
    } while ((h_face = prodb_next_part_face (h_face)) != NULL);
}

user_member_path (h_obj, memb_num, memb_id_tab, path)
```c
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);
    sprintf (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:

- **prodb_create_parallel_xsec()**
- **ProXsecPlanarCreate()**
- **ProXsecDelete()**

Functions Superseded:

- **prodb_delete_xsec()**

The function **prodb_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 prodb_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 prodb_delete_xsec() to delete cross sections must first call prodb_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:

- prodb_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.

```c
/*===========================================*/
```
FUNCTION: user_Neutral_Axis()
PURPOSE: Calculate the mass properties of a cross section.

```c
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;
    ProSysdata         *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");
    /*---------------------------------------------*/
    Select a starting plane for the cross section.
    /*---------------------------------------------*/
    ProMessageDisplay (msgfil, "USER Pick the start plane");
    status = ProSelect ("face", 1, 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, &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.

```c
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;
    ProUtilPlaneLineX(xpos, normal, cog, normal, cog);
    ProUtilVectorCopy (cog, points[n_points++]);
}
```

Delete the cross section and the owning feature (a datum plane).

```c
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.

```c
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);
}
This chapter describes the customized plot driver functions supported by Pro/TOOLKIT.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using the Plot Driver Functionality</td>
<td>41 - 2</td>
</tr>
</tbody>
</table>
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.

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.

<table>
<thead>
<tr>
<th>Primitive Type</th>
<th>Bound Function Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROINTF_LINE</td>
<td>user_intf_line()</td>
</tr>
<tr>
<td>PROINTF_TEXT</td>
<td>user_intf_text()</td>
</tr>
<tr>
<td>PROINTF_CIRCLE</td>
<td>user_intf_circle()</td>
</tr>
<tr>
<td>PROINTF_ARC</td>
<td>user_intf_arc()</td>
</tr>
<tr>
<td>PROINTF_POLYLINE</td>
<td>user_intf_polyline()</td>
</tr>
<tr>
<td>PROINTF_FILLED_POLY</td>
<td>user_intf_filled_poly()</td>
</tr>
</tbody>
</table>

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.

```c
/*================================================================*\ 
FUNCTION: UserPlotSetup() 
PURPOSE:  Set up the Customized Plot menu 
\*/
int UserPlotSetup()
{
  int menu_id, action, status;
  ProMdl model;
  int err;
  static char type[4], null[1], *type_list[] = {type, null};
  prointerface_create ("demo");
  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);
status = ProMenubuttonActionSet ("UGPLOT", "-Plot Demo", user_Demo_Plot, NULL, 0);
status = ProMenubuttonActionSet ("UGPLOT", "-Done/Return", UserMenuDeleteAndPop, NULL, 0);
status = ProMenubuttonActionSet ("UGPLOT", "UGPLOT", UserMenuDeleteAndPop, NULL, 0);
status = ProMenuCreate (PROMENUTYPE_MAIN, "UGPLOT", &menu_id);
status = ProMenuProcess ("UGPLOT", &action);
return (PRO_TK_NO_ERROR);

/*================================================================*
Command function to invoke a plot of type "demo"
/*================================================================*/
int user_Demo_Plot()
{
    wchar_t     w_fname[PRODEV_NAME_SIZE];
    char        fname[PRODEV_NAME_SIZE];
    ProFileName msgfil;
    ProStringToWstring (msgfil, "msg_ugfund.txt");
    ProMessageDisplay (msgfil, "USER Demo plot filename [QUIT] : ");
    if (ProMessageStringRead (PRODEV_NAME_SIZE, w_fname))
        return;
    ProWstringToString (fname, w_fname);
    demo_plot_file = fopen (fname, "w");
    prointerface_2d ("demo");
/*----------------------------------------------------*
Close and display the plot file.
\*----------------------------------------------------*/
fclose (demo_plot_file);
ProInfoWindowDisplay (w_fname, NULL, NULL);
}
/*===============================================*/
Bound function for plotting a LINE in the format "demo".
/*===============================================*/
void user_demo_line (point1, point2, color)
double point1[3],
point2[3];
int    color;
{
    fprintf (demo_plot_file, "LINE..\n");
    fprintf (demo_plot_file," point1 = %f, %f\n", point1[0],
            point1[1]);
    fprintf (demo_plot_file," point2 = %f, %f\n", point2[0],
            point2[1]);
    fprintf (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];
    fprintf (demo_plot_file, "TEXT..\n");
    fprintf (demo_plot_file," point = %f, %f\n", point[0], point[1]);
    fprintf (demo_plot_file," text  = %s\n", ProWstringToString (str, text));
    fprintf (demo_plot_file," size  = %f\n", size);
    fprintf (demo_plot_file," angle = %f\n", angle);
    fprintf (demo_plot_file," slant = %f\n", slant_angle);
    fprintf (demo_plot_file," width = %f\n", width_factor);
    fprintf (demo_plot_file," color = %d\n", color);
}
Bound function for plotting a CIRCLE in the format "demo"

```c
#include <stdio.h>

#define MAXPOINTS 100

void user_demo_circle (center, radius, color)
{
    double center[3],
    radius;
    int     color;
    {
        fprintf (demo_plot_file, "CIRCLE..\n");
        fprintf (demo_plot_file," center = %f, %f\n", center[0],
            center[1]);
        fprintf (demo_plot_file," radius = %f\n", radius);
        fprintf (demo_plot_file," color  = %d\n", color);
    }
}

BOUND function for plotting an ARC in the format "demo"

```
for (p = 0; p < n_pts; p++)
    fprintf (demo_plot_file," Point %d = %f, %f\n", p,
        array[p][0], array[p][1]);
    fprintf (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;
    printf (demo_plot_file, "FILLED POLY..
";
    for (p = 0; p < n_pts; p++)
        printf (demo_plot_file," Point %d = %f, %f\n", p,
            array[p][0], array[p][1]);
        printf (demo_plot_file," color    = %d\n", color);
    }
This chapter describes the Pro/TOOLKIT diagram and cabling functions.

**Topic**

- Diagrams 42 - 2
- Pro/ENGINEER Cabling Design 42 - 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 42 - 2
• Diagram Parameters on page 42 - 4
• Diagram Wire Spools on page 42 - 5
• Diagram Connectivity on page 42 - 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 prodb_get_element_name() and prodb_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:

```c
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:

```c
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.

```c
Typedef struct pro_model_item {
    ProMdl owner;
    int id;
    protype 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 though, 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_DESC 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()**
- **ProCablesFromLogicalCreate()**
- **ProCablesFromLogicalGet()**

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 **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.

### 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 Parameters

**Functions explained further:**

- **ProOutputFileWrite()**
- **ProInputFileRead()**

  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 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.

<table>
<thead>
<tr>
<th>Location Type</th>
<th>Equivalent Command in the CBL ROUTE Menu</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_LOC_CONNECTOR</td>
<td>Connector</td>
</tr>
<tr>
<td>PRO_LOC_POINT</td>
<td>Pnt/Vertex</td>
</tr>
<tr>
<td>PRO_LOC_FREE</td>
<td>Free</td>
</tr>
<tr>
<td>PRO_LOC_DEPENDENT</td>
<td>Dependent</td>
</tr>
<tr>
<td>PRO_LOC_AXIS</td>
<td>Along Axis</td>
</tr>
<tr>
<td>PRO_LOC_USE_DIR</td>
<td>Use Dir</td>
</tr>
<tr>
<td>PRO_LOC_OFFSET</td>
<td>Offset</td>
</tr>
<tr>
<td>PRO_LOC_SPLICE</td>
<td>Splice</td>
</tr>
<tr>
<td>PRO_LOC_LOC</td>
<td>Location</td>
</tr>
<tr>
<td>PRO_LOC_OFFSET_CSYS</td>
<td>Coordinate Offset</td>
</tr>
<tr>
<td>PRO_LOC_OFFSET_AXIS</td>
<td>Axis Offset</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>type</th>
<th>refs</th>
<th>axis_flip</th>
<th>offsets</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROLOC_CONNECTOR</td>
<td>The coordinate system datum for the entry port</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>PROLOC_POINT</td>
<td>The datum point</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>PROLOC_AXIS</td>
<td>The axis</td>
<td>0 or 1 to show the routing direction</td>
<td>—</td>
</tr>
<tr>
<td>PROLOC_OFFSET</td>
<td>The coordinate system datum to define the offset directions</td>
<td>—</td>
<td>Offset distances from the previous location</td>
</tr>
<tr>
<td>PROLOC_LOC</td>
<td>An existing routing location</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

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.
Finite Element Modeling (FEM)

This chapter contains descriptions of the Pro/TOOLKIT functions that support Pro/ENGINEER Finite Element Modeling (FEM).

**Topic** | **Page**
--- | ---
Overview | 43 - 2
Exporting an FEA Mesh | 43 - 3
Getting Constraint Case Names | 43 - 5
Getting Mesh Controls | 43 - 5
Getting Constraints | 43 - 8
Getting Bar Elements, Contacts, and Runners | 43 - 11
Getting Mass Elements | 43 - 12
Getting Shell Pairs | 43 - 13
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:

- **pro_export_fea_mesh()**

  The function `pro_export_fea_mesh()` performs automatic mesh generation and outputs standard output files.

  The function uses the data structure `Pro_mesh_data`, which is defined as follows:

  ```c
  typedef struct pro_mesh_data
  {
    int  mesh_type;
    int  outp_system_type;
    int  analysis_type;
    int  midpnt_flag;
    int  fix_elems_flag;
    int  object_to_mesh_type;
    int  shell_elem_type;
    int  csys_id;
    int  quilt_id;
  } Pro_mesh_data;
  ```

  The `Pro_mesh_data` fields are as follows:

  - **mesh_type**—The mesh type. The possible values are as follows:
    - `PRO_FEA_SOLID_MESH`—Mesh solid parts using tetrahedral solid mesh elements.
    - `PRO_FEA_SHELL_MESH`—Shell mesh using triangular or quadrangular mesh elements. This type is designed for meshing surfaces.
    - `PRO_FEA_MIXED_MESH`—Mesh models with a mixture of shell and tetrahedral mesh elements.

  - **outp_system_type**—The type of output system. The possible values are as follows:
    - `PRO_FEA_PATRAN`—Write a PATRAN® neutral file in ASCII format, named `filename.pat`.
    - `PRO_FEA_ANSYS`—Write an ANSYS® output file in ASCII format, named `filename.ans`.
    - `PRO_FEA_NASTRAN`—Write an MSC/NASTRAN® file in ASCII format, named `filename.nas`.
    - `PRO_FEA_SUPERTAB`—Write a Supertab™ universal file in ASCII format, named `filename.unv`.
PRO_FEA_C_FLOW—Write a C-MOLD® file that includes information about the mold model.

PRO_FEA_COSMOS—Write a COSMOS/M® file in ASCII format, named filename.cos.

PRO_FEA_NEUTRAL—Writes a neutral file in ASCII format.

• `analysis_type`—The analysis type. The possible values are as follows:
  - PRO_FEA_STRUCTURAL—Structural analysis, including stress, strain, thermal stress, and displacement
  - PRO_FEA_THERMAL—Thermal analysis, including temperature, heat flux, and heat gradient
  - PRO_FEA_FLOW—Mold filling analysis

• `midpnt_flag`—The type of mesh midpoint—PRO_FEA_LINEAR or PRO_FEA_PARABOLIC. The number of nodes that are output depends on the type of element, FEA system to output to, and the analysis type. In general, the output is as follows:

<table>
<thead>
<tr>
<th>Element</th>
<th>PRO_FEA_LINEAR</th>
<th>PRO_FEA_PARABOLIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tetrahedron</td>
<td>4 nodes</td>
<td>10 nodes</td>
</tr>
<tr>
<td>Triangle</td>
<td>3 nodes</td>
<td>6 nodes</td>
</tr>
<tr>
<td>Quadrangle</td>
<td>4 nodes</td>
<td>8 nodes</td>
</tr>
<tr>
<td>Bar</td>
<td>2 nodes</td>
<td></td>
</tr>
<tr>
<td>Mass</td>
<td>1 node</td>
<td></td>
</tr>
</tbody>
</table>

• `fix_elems_flag`—For a parabolic mesh, this field is either TRUE or FALSE. This field is ignored for linear mesh.

• `object_to_mesh_type`—An enumerated type that specifies the object-to-mesh type. This field is ignored for non-shell mesh. The possible values are as follows:
  - PRO_FEA_SOLID
  - PRO_FEA_SURFACE (for datum quilts)
  - PRO_FEA_BOUNDARY
• *shell_elem_type*—The type of shell element. This field is ignored for solid mesh. The possible values are as follows:
  - PRO_FEA_TRIANGLE
  - PRO_FEA_QUADR

• *csys_id*—The coordinate system identifier. The default value is −1.

• *quilt_id*—The quilt identifier, if *object_to_mesh_type* is PRO_FEA_SURFACE. Otherwise, this field is ignored.

### 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:

```c
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 43 - 7 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 43 - 7 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 43 - 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:

```c
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, Contacts, and Runners

Functions introduced:

- **profem_get_bar_elements()**
- **profem_get_contacts()**
- **profem_get_runners()**

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 function **profem_get_runners()** retrieves all the runners defined for C-FLOW modeling (parts only). It uses the data structure `Pro_fem_runner`.

The data structure for these functions is defined as follows:

```c
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, Pro_fem_runner;
```

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**
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:

```c
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 43 - 13 for the definition of this data structure.
- **p_location**—The location of the element. See the section `Pro_fem_region Data Structure` on page 43 - 7 for the definition of this data structure.
Pro_fem_mass_props Data Structure

The `Pro_fem_mass_props` data structure is defined as follows:

```c
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:

```c
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 */
} 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.

```c
/* 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
#define USER_FEM_GET_RUNNERS 6

/*-----------------------------------------------*/
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];
}
}`

```
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_COLD_RUNNER, "PRO_FEM_COLD_RUNNER"},
    {PRO_FEM_HOT_RUNNER, "PRO_FEM_HOT_RUNNER"},
    {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--;
}

Finite Element Modeling (FEM)
return (contact_names[i].name);
} /*---------------------------------------------------------------*/

FUNCTION: user_get_fem_action()
PURPOSE: Process each FEM action for file input and output.

```c
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  *meshCtrls;
    Pro_fem_mass_element  *mass_elems;
    Pro_fem_contact     *contacts;
    Pro_fem_runner      *runners;
    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 = fopen (fname, "a+")) == NULL)
    {
        ProMessageDisplay(msg_fil, "USER %0s", "Cannot open
                        file for write");
        return (0);
    }
    fprintf (fp, "\n");
```
Apply the appropriate action corresponding to the menu selection.

```c
switch (action)
{
    case USER_FEM_GET_CONS_NAMES:
        nnames = profem_get_con_case_names (p_model, &wnames);
        fprintf (fp, "Constraint Case Names[%d]\n", nnames);
        for (i = 0; i < nnames; i++)
            fprintf (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);
        fprintf (fp, "Constraints[%d]\n", nnames);
        for (i = 0; i < nnames; i++)
        {
            fprintf (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++)
                fprintf (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);
        fprintf (fp, "Mass Elements[%d]\n", nitems);
        for (i = 0; i < nitems; i++)
        {
            fprintf (fp, "\t MassValue: %8.3f\n",
                        mass_elems[i].p_properties->value);
            fprintf (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, &meshCtrls);
        fprintf (fp, "Mesh Controls[%d]\n", nitems);
        for (i = 0; i < nitems; i++)
            fprintf (fp, "\t %s\n", user_mctype_to_name (meshCtrls[i].control_type));
        break;
}```
case USER_FEM_GET_BAR_ELEMS:
    nitems = profem_get_bar_elements (p_model, &bar_elems);
    fprintf (fp, "Bar Elements[%d]\n", nitems);
    for (i = 0; i < nitems; i++)
        fprintf (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);
    fprintf (fp, "Contacts[%d]\n", nitems);
    for (i = 0; i < nitems; i++)
        fprintf (fp, \t "%s\n", user_contact_to_name (contacts[i].type));
    break;

case USER_FEM_GET_RUNNERS:
    nitems = profem_get_runners (p_model, &runners);
    fprintf (fp, "Runners[%d]\n", nitems);
    for (i = 0; i < nitems; i++)
        fprintf (fp, \t "%s\n", user_contact_to_name (runners[i].type));
    break;

default:
    break;
}

/*-----------------------------------------------*/
Close the output file and display it
/*-----------------------------------------------*/
fclose (fp);
err = ProStringToWstring (path, "femdata.txt");
err = ProInfoWindowDisplay (path, NULL, NULL);
return (0);
This chapter describes the Pro/TOOLKIT functions that enable you to create and manipulate external objects.

**Topic**

- Summary of External Objects 44 - 2
- External Objects and Object Classes 44 - 3
- External Object Data 44 - 6
- External Object References 44 - 12
- Callbacks for External Objects 44 - 14
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 cannot 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.

One example of external objects is the representation of loads and boundary conditions in the FEA module—Pro/MESH. These graphical entities display all the criteria outlined below.

**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 44 - 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 ‘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 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 44 - 6
- Selection Data for External Objects on page 44 - 10
- Manipulating External Object Data on page 44 - 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 \texttt{ProWindowRepaint()}. 

\textbf{Allocating Display Data}

Function introduced:

- \texttt{ProDispdatAlloc()}

For display data, the workspace handle \texttt{ProWExtobjdata} is allocated using the function \texttt{ProDispdatAlloc()}. Because the other Pro/TOOLKIT display data functions require \texttt{ProWExtobjdata} as input, you must call \texttt{ProDispdatAlloc()} before calling the other functions in this section.

The input for \texttt{ProDispdatAlloc()} is the address of a \texttt{ProWExtobjdata} object that you declare in your application. You must set this \texttt{ProWExtobjdata} object to NULL before passing its address to \texttt{ProDispdatAlloc()}. 

\textbf{Creating the External Object Entity}

Functions introduced:

- \texttt{ProDispdatEntsSet()}
- \texttt{ProDispdatEntsGet()}

External objects are currently limited to text and wireframe entities. You can specify the entities to be displayed by creating an array of \texttt{ProCurvedata} objects that contain that necessary information. \texttt{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 \texttt{ProCurvedata} array, the coordinate system used is the default model coordinate system.

After you have created the array of \texttt{ProCurvedata} objects, you can add entities to the display data by calling the function \texttt{ProDispdatEntsSet()}. Note that \texttt{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 $2\pi$.

To obtain the entities that make up an external object, call the function \texttt{ProDispdatEntsGet()}. 

Example 1: Creating an External Object on page 44 - 18 shows how to specify an external object that is composed of line segments.
Transformation of the External Object

Functions introduced:

- ProDispdatTrfSet()
- ProDispdatTrfGet()
- ProExtobjScreentrfGet()

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 44 - 18 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 `ProExtobjScreentrfGet()`.

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 the 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 44 - 18 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:

```c
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:

```c
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:

```c
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:

<table>
<thead>
<tr>
<th>Callback Type</th>
<th>When it is Triggered</th>
</tr>
</thead>
<tbody>
<tr>
<td>display_CB</td>
<td>The external object is displayed. Currently, this is not implemented.</td>
</tr>
<tr>
<td>select_CB</td>
<td>The external object is selected. Currently, this is not implemented.</td>
</tr>
<tr>
<td>owner_modify_CB</td>
<td>The owner of the external object is modified. Currently, this is not implemented.</td>
</tr>
<tr>
<td>owner_suppress_CB</td>
<td>The owner of the external object is suppressed. Currently, this is not implemented.</td>
</tr>
<tr>
<td>owner_delete_CB</td>
<td>The owner of the external object is deleted. Currently, this is not implemented.</td>
</tr>
<tr>
<td>ref_modify_CB</td>
<td>The reference of the external object is modified.</td>
</tr>
<tr>
<td>ref_suppress_CB</td>
<td>The reference of the external object is suppressed.</td>
</tr>
<tr>
<td>ref_delete_CB</td>
<td>The reference of the external object is deleted.</td>
</tr>
</tbody>
</table>
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
\texttt{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
\texttt{ProExtobjClassWarningDisable()}.  

To enable warnings for a single external object (not the entire
class), call the function \texttt{ProExtobjWarningEnable()}. This
function is similar to \texttt{ProExtobjClassWarningEnable()} except
the first argument for \texttt{ProExtobjWarningEnable()} is a pointer to
a \texttt{ProExtobj} object. To disable the warnings for a single external
object, call the function \texttt{ProExtobjWarningDisable()}.  

\textbf{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.

\textbf{Note:} For the sake of simplicity, the example does not
implement selection data.
PURPOSE: Example of external object creation

```c
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_mdlitem;      /* target surface ProModelitem */
    ProSelection *p_surf_sel;       /* target surface ProSelection */
    ProModelitem surf_owner_mdlitem; /* 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   curvedata[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.

```c
status = UserMakeSelections (&surf, &surf_mdlitem, surf_xyz, &p_surf_sel);
if (status != PRO_TK_NO_ERROR)
    return (status);
```

Get the ProModelitem for the surface owner.

```c
status = ProMdlToModelitem (surf_mdlitem.owner, &surf_owner_mdlitem);
```

Determine the uv parameters from the xyz point.

```c
status = ProSurfaceParamEval ((ProSolid) surf_mdlitem.owner, surf, surf_xyz, surf_uv);
```

Evaluate the surface data (normal, gradient,...) at the uv point.

```c
status = ProSurfaceXyzdataEval (surf, surf_uv, surf_xyz, deriv1, deriv2, surf_norm);
```

Get the transformation for the arrow.

```c
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.

```c
status = ProExtobjCreate (&User_arrow_class, &surf_owner_mdlitem, &arrow_obj);
```

Set the object identifiers to be reusable.

```c
status = ProExtobjReusableSet (&arrow_obj);
```

Initialize the display data structure.

```c
status = ProDispdatAlloc (&arrow Disp_data);
```

Set the display scale, color, line type, and display properties.

```c
status = ProDispdatScaleSet (arrow Disp_data, scale);
status = ProDispdatColorSet (arrow Disp_data, ARROW COLOR);
status = ProDispdatLinestyleSet (arrow Disp_data, ARROW LINSTYLE);
```
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.

```c
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.

```c
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);

```c
/*====================================================================*
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.

```c
int UserMakeSelections (ProSurface *p_surf, ProModelitem *p_mdl_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);
    }

    status = ProSelectionModelitemGet (*p_sel, p_mdl_item);
    status = ProGeomitemToSurface ((ProGeomitem *) p_mdl_item, p_surf);

    status = ProSelectionPoint3dGet (*p_sel, xyz);

    *pp_surf_sel = p_sel;
    return (PRO_TK_NO_ERROR);
}
```
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.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>45 - 2</td>
</tr>
<tr>
<td>Skeleton Model Functions</td>
<td>45 - 5</td>
</tr>
<tr>
<td>Assembly Component Functions</td>
<td>45 - 5</td>
</tr>
<tr>
<td>External Reference Control Functions</td>
<td>45 - 6</td>
</tr>
<tr>
<td>Feature and CopyGeom Feature Functions</td>
<td>45 - 7</td>
</tr>
<tr>
<td>External Reference Data Gathering</td>
<td>45 - 9</td>
</tr>
</tbody>
</table>
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.

**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()`
- `ProAsmcompsUnplaced()`
- `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:

```c
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:

```c
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.

```c
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 */
} ProCopiedRefState;
```
External Reference Data Gathering

Functions introduced:

- `ProFeatureExternChildrenGet()`
- `ProFeatureExternParentsGet()`
- `ProSolidExternChildrenGet()`
- `ProSolidExternParentsGet()`
- `ProExtRefInfoFree()`
- `ProExtRefStateGet()`
- `ProExtRefTypeGet()`
- `ProExtRefAsmcompsGet()`
- `ProExtRefOwnMdIGet()`
- `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" */
}
The enumerated type `ProRefState` defines the possible states of top-level solids (part, assembly, or component) to which a lower-level solid refers:

```c
typedef enum
{
    PRO_REF_NOT_FOUND  = 0,  /* references are not found */
    PRO_REF_ACTIVE     = 1,  /* referenced entity on the "master" model is up to date */
    PRO_REF_FROZEN     = 3,  /* "master" model currently not in session */
    PRO_REF_SUPPRESSED = 4,  /* referenced entity is suppressed in the "master" model */
    PRO_REF_MISSING    = 5   /* referenced entity is missing in the "master" model */
} ProRefState;  /* state of referenced entity in "master" model */
```

Structures `ProExtFeatRef` and `ProExtRefInfo` provide pointers to a structure containing external references for a specified feature:

```c
typedef struct ext_feat_ref *ProExtFeatRef;

typedef struct
{
    ProExtRefType    type;
    ProExtFeatRef   *ext_refs;
    int              n_refs;
} ProExtRefInfo;
```
This chapter describes the Pro/TOOLKIT functions that enable you to create animation frames and movies.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>46 - 2</td>
</tr>
<tr>
<td>Animation Objects</td>
<td>46 - 3</td>
</tr>
<tr>
<td>Animation Frames</td>
<td>46 - 4</td>
</tr>
<tr>
<td>Playing Animations</td>
<td>46 - 4</td>
</tr>
</tbody>
</table>
Introduction

Pro/TOOLKIT provides the functions that enable you to animate parts, subassemblies, and external objects. 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 46-1: Animation Dialog Box
  
  ![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, subassembly, or external object. 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()**.
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 46 - 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()**

  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.

**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.

```c
/*---------------------------------------------*
Pro/TOOLKIT includes
\*---------------------------------------------*/
#include "ProToolkit.h"
#include "ProObjects.h"
#include "ProSelection.h"
#include "ProAsmcomppath.h"
#include "ProAnimate.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

```c
int UserAsmcompAnimate()
{
    ProMdl  model;    /* current model */
    ProModelitem anim_mdlitem; /* 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"); /* translation function */
    Get the current model.
    err = ProMdlCurrentGet (&model);
    if (err != PRO_TK_NO_ERROR)
    {
        ProMessageDisplay (msg_file, "USER %s",
            "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_mdlitem);
err = ProModelitemMdlGet (&anim_mdlitem, &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];    
}
Access to Pro/INTRALINK

This chapter describes the functions that enable you to interact with Pro/INTRALINK.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registering a Workspace with a Pro/ENGINEER Session</td>
<td>47 - 2</td>
</tr>
<tr>
<td>Checking Out Files from a Commonspace</td>
<td>47 - 2</td>
</tr>
</tbody>
</table>
Registering a Workspace with a Pro/ENGINEER Session

Functions introduced:

- ProWorkspaceRegister()
- ProCurrentWorkspaceGet()

When you start Pro/ENGINEER from an Application Manager that has a Pro/Workspace application currently registered with it, the Workspace is automatically registered with the Pro/ENGINEER session as well. However, when you start Pro/ENGINEER independently of a Pro/INTRALINK session, the Pro/ENGINEER session will have no connection to the Workspace. Independent start-up situations include initiating Pro/ENGINEER from the command line, or using an asynchronous Pro/TOOLKIT application. In these cases, you must use the function ProWorkspaceRegister() to register the Workspace application with the Pro/ENGINEER session. This function returns PRO_TK_E_FOUND if the requested Workspace or another Workspace is already registered; only one Workspace can be registered with a Pro/ENGINEER session at a time. Use the function ProCurrentWorkspaceGet() to determine whether a Workspace is currently registered, and if so, output its name.

Checking Out Files from a Commonspace

Function introduced:

- ProObjModelsCheckout()

The function ProObjModelsCheckout() accesses one or more objects in the Commonspace, creating links to or copies of those objects in a Workspace. The argument list includes an array of names of objects to be checked out, and a corresponding array of object versions. You can also specify the dependency criteria. This argument identifies which dependents of the requested objects should be checked out with them. The target Workspace must be registered with the Pro/ENGINEER session.
Example 1: Copying an Assembly from the Commonspace

The following example code shows how to copy an assembly from the Commonspace.

```c
int UserObjMdlCheckout()
{
    static wchar_t msgfil[80];
    int status;
    ProName obj_model, workspace;

    ProStringToWstring (msgfil, "obj_models_chkout.txt");

    status = ProMessageDisplay (msgfil, "USER Enter name of the assembly to be copied from common space");

    status = ProMessageStringRead (PRO_NAME_SIZE, obj_model);

    status = ProCurrentWorkspaceGet (workspace);

    if (status == PRO_TK_NO_ERROR)
        status = ProObjModelsCheckOut (PRO_B_FALSE, workspace, 1, obj_model, NULL, 2);
    else
    {
        status = ProMessageDisplay (msgfil, "USER Enter name of workspace to register");
        status = ProMessageStringRead (PRO_NAME_SIZE, workspace);
        status = ProWorkspaceRegister (workspace);
        if (status == PRO_TK_NO_ERROR)
            status = ProObjModelsCheckOut (PRO_B_FALSE, workspace, 1, obj_model, NULL, 2);
        else
            status = ProMessageDisplay (msgfil, "USER The Workspace could not be registered; checkout terminated");
    }

    return (status);
}
```
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.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Piping Terminology</td>
<td>48 - 2</td>
</tr>
<tr>
<td>Linestock Management Functions</td>
<td>48 - 3</td>
</tr>
<tr>
<td>Pipeline Connectivity Analysis</td>
<td>48 - 5</td>
</tr>
</tbody>
</table>
Piping Terminology

Pro/TOOLKIT supports Pro/Piping. Pro/Piping uses specific terminology. This section defines this terminology.

A pipeline is a 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:

```c
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

- \texttt{ProPnetworkExtensionVisit()}
- \texttt{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 \texttt{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 \texttt{ProPextension}. The function \texttt{ProPnetworkExtensionVisit()} visits all the extensions in a network.

The function \texttt{ProPextensionFlowGet()} tells you the flow direction in relation to the sequence of members in the extension.

Members

Functions introduced:

- \texttt{ProPextensionMemberVisit()}
- \texttt{ProPmemberTypeGet()}

An extension is conceptually divided into objects called members, described by the opaque object \texttt{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 \texttt{ProPterminator}.
- Junction—The item that describes how the pipeline branches, described by the opaque object \texttt{ProPjunction}.
- Series—A non branching sequence of pipeline objects, described by the opaque object \texttt{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 **ProObjectSegmentGet()** outputs the Segment contained by an Object of the appropriate type. The Segment is described in the next section.

The function **ProObjectFittingGet()** outputs the fitting contained by an object of the appropriate type. The fitting is represented by the opaque object ProPfitting. The assembly component that represents the fitting can be found using the function **ProPfittingAsmcompGet()**.

The function **ProObjectStubinGet()** 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 **ProObjectSelectionGet()** 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 **ProPipelineLnstkGet()** 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.
This chapter describes the Pro/TOOLKIT utility functions.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration Options</td>
<td>49 - 2</td>
</tr>
<tr>
<td>Registry File Data</td>
<td>49 - 3</td>
</tr>
<tr>
<td>Trail Files</td>
<td>49 - 3</td>
</tr>
<tr>
<td>Pro/ENGINEER License Data</td>
<td>49 - 3</td>
</tr>
<tr>
<td>Current Directory</td>
<td>49 - 3</td>
</tr>
<tr>
<td>File Handling</td>
<td>49 - 4</td>
</tr>
<tr>
<td>Wide Strings</td>
<td>49 - 5</td>
</tr>
<tr>
<td>Running ModelCHECK</td>
<td>49 - 7</td>
</tr>
</tbody>
</table>
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:

```c
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:

- ProToolkitApplExecPathGet()
- ProToolkitApplTextPathGet()

These functions retrieve the values of the exec_file and text_dir statements in the Pro/TOOLKIT registry file. The output is in the form of wide strings.

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()**
- **ProDirectoryChoose()**
- **ProFilenameParse()**
- **ProPathCreate()**
- **ProInfoWindowDisplay()**
- **ProFileEdit()**

The function **ProFilesList()** outputs a list of the contents of a directory, given the directory path. You can filter the list so it includes 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 outputs an array of subdirectory names, regardless of the filter used.

The function **ProFileOpen()** invokes the 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 **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 outputs the selected directory path.

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 outputs 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.

**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. It can be configured to run interactively as well as 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 will be 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.
This chapter describes various methods of importing and exporting files in Pro/TOOLKIT.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exporting Information Files</td>
<td>50 - 2</td>
</tr>
<tr>
<td>Exporting 2D Models</td>
<td>50 - 4</td>
</tr>
<tr>
<td>Exporting 3D Models</td>
<td>50 - 6</td>
</tr>
<tr>
<td>Shrinkwrap Export</td>
<td>50 - 14</td>
</tr>
<tr>
<td>Importing Parameter Files</td>
<td>50 - 19</td>
</tr>
<tr>
<td>Importing 2D Models</td>
<td>50 - 21</td>
</tr>
<tr>
<td>Importing 3D Models</td>
<td>50 - 22</td>
</tr>
</tbody>
</table>
Exporting Information Files

Functions Introduced:

- ProOutputFileWrite()
- ProIntfSliceFileExport()
- ProExportVRML()

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.

<table>
<thead>
<tr>
<th>Export Format</th>
<th>Pro/TOOLKIT function</th>
<th>Type Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bill of materials</td>
<td>ProOutputFileWrite()</td>
<td>PRO_BOM_FILE</td>
</tr>
<tr>
<td>Drawing setup file</td>
<td>ProOutputFileWrite()</td>
<td>PRO_DWG_SETUP_FILE</td>
</tr>
<tr>
<td>Feature identifier</td>
<td>ProOutputFileWrite()</td>
<td>PRO_FEAT_INFO</td>
</tr>
<tr>
<td>Material file (currently assigned material)</td>
<td>ProOutputFileWrite()</td>
<td>PRO_MATERIAL_FILE</td>
</tr>
<tr>
<td>CL Data output, NC Sequence file</td>
<td>ProOutputFileWrite()</td>
<td>PRO_MFG_FEAT_CL</td>
</tr>
<tr>
<td>CL Data operation file</td>
<td>ProOutputFileWrite()</td>
<td>PRO_MFG_OPER_CL</td>
</tr>
<tr>
<td>Information on Pro/Engineer Objects</td>
<td>ProOutputFileWrite()</td>
<td>PRO_MODEL_INFO</td>
</tr>
<tr>
<td>Program file</td>
<td>ProOutputFileWrite()</td>
<td>PRO_PROGRAM_FILE</td>
</tr>
<tr>
<td>Cable Parameters file</td>
<td>ProOutputFileWrite()</td>
<td>PRO_CABLE_PARAMS_FILE</td>
</tr>
<tr>
<td>Connector Parameters file</td>
<td>ProOutputFileWrite()</td>
<td>PRO_CONNECTOR_PARAMS_FILE</td>
</tr>
<tr>
<td>Spool file</td>
<td>ProOutputFileWrite()</td>
<td>PRO_SPOOL_FILE</td>
</tr>
</tbody>
</table>

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`. 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.**

For some of the options, you need to provide some more information, using the last four arguments. The following list shows which arguments to set and when:

- For `PRO_FEA_INFO`, `PRO_MFG_FEA_CL`, and `PRO_MFGOPER_CL`, set the following argument:
  - `arg2`—The integer identifier of the feature.

- For `PRO_CGM_FILE`, set the following arguments:
  - `arg2`—Set to `PRO_EXPORT_CGM_CLEAR_TEXT` or `PRO_EXPORT_CGM_MIL_SPEC`.
  - `arg3`—Set to `PRO_EXPORT_CGM_ABSTRACT` or `PRO_EXPORT_CGM_METRIC`.

- For `PRO_CONNECTOR_PARAMS`, set the following parameters:
  - `arg1`—Represents the component path (`memb_id_tab`)
  - `arg2`—Represents the component number (`memb_num`)
Exporting 2D Models

Functions Introduced:

- `Pro2dExport()`
- `ProProductviewExport()`
- `ProPrintPrinterOptionsGet()`
- `ProPrintMdlOptionsGet()`
- `ProPrintPlacementOptionsGet()`
- `ProPrintPCFOptionsGet()`
- `ProPrintExecute()`

<table>
<thead>
<tr>
<th>Export Format</th>
<th>Pro/TOOLKIT Functions</th>
<th>Type Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEP</td>
<td><code>Pro2dExport()</code></td>
<td><code>PRO_STEP_FILE</code></td>
</tr>
<tr>
<td>CGM file</td>
<td><code>ProOutputFileWrite()</code></td>
<td><code>PRO_CGM_FILE</code></td>
</tr>
<tr>
<td>MEDUSA</td>
<td><code>Pro2dExport()</code></td>
<td><code>PRO_MEDUSA_FILE</code></td>
</tr>
<tr>
<td>HPGL</td>
<td><code>ProProductviewExport()</code></td>
<td>N/A</td>
</tr>
<tr>
<td>DXF</td>
<td><code>ProOutputFileWrite()</code></td>
<td><code>PRO_DXF_FILE</code></td>
</tr>
<tr>
<td>IGES</td>
<td><code>ProOutputFileWrite()</code></td>
<td><code>PRO_IGES_FILE</code></td>
</tr>
<tr>
<td>DWG</td>
<td><code>ProOutputFileWrite()</code></td>
<td><code>PRO_DWG_FILE</code></td>
</tr>
<tr>
<td>SET</td>
<td><code>ProOutputFileWrite()</code></td>
<td><code>PRO_SET_FILE</code></td>
</tr>
<tr>
<td>Plot file</td>
<td><code>ProPrintExecute()</code></td>
<td>N/A</td>
</tr>
</tbody>
</table>

The function `Pro2dExport()` creates interface files from existing two-dimensional models by exporting the interface data. The exported model can be a drawing, layout, diagram, or drawing format. This function supports the STEP and Medusa formats.

The function `ProProductviewExport()` exports a part, assembly or drawing to the product view format. During 2D export, it exports drawings to HPGL format.
Use the function **ProPrintPrinterOptionsGet()** to get the options for a specified printer. Specify the printer type as the input argument for this function. The types of printers supported 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 list in "Add Printer" in Pro/ENGINEER's print dialog.

The function gets the initialized printer options. The options include the file related options, print command options and printer specific options.

Use the function **ProPrintMdlOptionsGet()** to get the initialized model options for the model to be printed.

Use the function **ProPrintPlacementOptionsGet()** to get the current print placement options, viz., print scale, offset, zoom, and so on.

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.
Exporting 3D Models

Pro/TOOLKIT provides export capabilities for three dimensional geometry to various formats.

Functions Introduced:

- ProIntf3DFileWrite()
- ProOutputBrepRepresentationAlloc()
- ProOutputBrepRepresentationFlagsSet()
- ProOutputBrepRepresentationIsSupported()
- ProOutputBrepRepresentationFree()
- ProOutputInclusionAlloc()
- ProOutputInclusionFlagsSet()
- ProOutputInclusionFree()
- ProOutputLayerOptionsAlloc()
- ProOutputLayerOptionsAutoidSet()
- ProOutputLayerOptionsSetupfileSet()
- ProOutputLayerOptionsFree()
- ProOutputAssemblyConfigurationIsSupported()
- ProRasterFileWrite()
- ProIntfSliceFileExport()
- ProExportVRML()
- ProProductviewExport()

<table>
<thead>
<tr>
<th>Export Format</th>
<th>Pro/TOOLKIT Functions</th>
<th>Type Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEP file (Standard for the Exchange of Product Model Data)</td>
<td>ProIntf3DFileWrite()</td>
<td>PRO_INTF_EXPORT_STE P</td>
</tr>
<tr>
<td>SET file (Standard for Exchange and Transfer)</td>
<td>ProIntf3DFileWrite()</td>
<td>PRO_INTF_EXPORT_SET</td>
</tr>
<tr>
<td>VDA file</td>
<td>ProIntf3DFileWrite()</td>
<td>PRO_INTF_EXPORT_VDA</td>
</tr>
<tr>
<td>IGES (3D) file</td>
<td>ProIntf3DFileWrite()</td>
<td>PRO_INTF_EXPORT_IGES</td>
</tr>
<tr>
<td><strong>Export Format</strong></td>
<td><strong>Pro/TOOLKIT Functions</strong></td>
<td><strong>Type Constant</strong></td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-----------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>CATIA file</td>
<td>ProInt3DFileWrite()</td>
<td>PRO_INTF_EXPORT_CATIA</td>
</tr>
<tr>
<td>CATIA (.model) file</td>
<td>ProInt3DFileWrite()</td>
<td>PRO_INTF_EXPORT_CATIA_MODEL</td>
</tr>
<tr>
<td>SAT file (ACIS format for Pro/ENGINEER)</td>
<td>ProInt3DFileWrite()</td>
<td>PRO_INTF_EXPORT_SAT</td>
</tr>
<tr>
<td>NEUTRAL file (ASCII text)</td>
<td>ProInt3DFileWrite()</td>
<td>PRO_INTF_EXPORT_NEUTRAL</td>
</tr>
<tr>
<td>CADDs file</td>
<td>ProInt3DFileWrite()</td>
<td>PRO_INTF_EXPORT_CADDS</td>
</tr>
<tr>
<td>CATIA (.session) file</td>
<td>ProInt3DFileWrite()</td>
<td>PRO_INTF_EXPORT_CATIA_SESSION</td>
</tr>
<tr>
<td>PDGS file</td>
<td>ProInt3DFileWrite()</td>
<td>PRO_INTF_EXPORT_PDGS</td>
</tr>
<tr>
<td>CATIA facets file</td>
<td>ProIntSliceFileExport()</td>
<td>PRO_CATIAFACETS_FILE</td>
</tr>
<tr>
<td>INVENTOR file</td>
<td>ProIntSliceFileExport()</td>
<td>PRO_INVENTOR_FILE</td>
</tr>
<tr>
<td>Render file</td>
<td>ProIntSliceFileExport()</td>
<td>PRO_RENDERER_FILE</td>
</tr>
<tr>
<td>SLA ASCII file</td>
<td>ProIntSliceFileExport()</td>
<td>PRO_SLA_ASCII_FILE</td>
</tr>
<tr>
<td>SLA Binary file</td>
<td>ProIntSliceFileExport()</td>
<td>PRO_SLA_BINARY_FILE</td>
</tr>
<tr>
<td>JPEG file</td>
<td>ProRasterFileWrite()</td>
<td>PRORASTERTYPE_JPEG</td>
</tr>
<tr>
<td>BMP file</td>
<td>ProRasterFileWrite()</td>
<td>PRORASTERTYPE_BMP</td>
</tr>
<tr>
<td>TIFF file</td>
<td>ProRasterFileWrite()</td>
<td>PRORASTERTYPE_TIFF</td>
</tr>
<tr>
<td>EPS file (Postscript)</td>
<td>ProRasterFileWrite()</td>
<td>PRORASTERTYPE_EPS</td>
</tr>
<tr>
<td>OL, ED</td>
<td>ProProductviewExport()</td>
<td>N/A</td>
</tr>
<tr>
<td>VRML</td>
<td>ProExportVRML</td>
<td>N/A</td>
</tr>
<tr>
<td>Shrinkwrap</td>
<td>ProSolidShrinkwrapCreate()</td>
<td>N/A</td>
</tr>
</tbody>
</table>
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**
- **CATIA MODEL**
- **SAT** (ACIS format in Pro/ENGINEER)
- **NEUTRAL**
- **CADDs**
- **CATIA SESSION**
- **PDGS**

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 `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 6 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.
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 function ProProductviewExport() exports parts to OL format and drawings to ED format.

Example 1: 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.

```c++
#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 nonassociative 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

<table>
<thead>
<tr>
<th>Export Format</th>
<th>Pro/TOOLKIT Functions</th>
<th>Type Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shrinkwrap</td>
<td>ProSolidShrinkwrapCreate()</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Functions 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()**
- **ProShrinkwrapoptionsAdditionalsurfacesSet()**

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 the function **ProShrinkwrapoptionsAdditionalsurfacesSet()** 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

Functions Introduced:

- **ProShrinkwrapoptionsAdditionalcomponentsSet()**
  
  Use the function **ProShrinkwrapoptionsAdditionalcomponentsSet()** to select individual components of the assembly to be merged into the Shrinkwrap model.

**Example 2: To Export a model to VRML format**

This example creates a faceted shrinkwrap model in VRML format

```c
#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);
```
Example 3: 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.

```c
#include <ProToolkit.h>
#include <ProMenu.h>
#include <ProMdl.h>
#include <ProShrinkwrap.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, PRO_MDL_PART, &new_model);
    status = ProSolidShrinkwrapCreate (solid, new_model, NULL, options);

    return status;
}
```
Free input argument memory

ProShrinkwrapoptionsFree (options);

return PRO_TK_NO_ERROR;

}  
#undef MSGFIL

### Importing Parameter Files

<table>
<thead>
<tr>
<th>Import Format</th>
<th>Pro/TOOLKIT Functions</th>
<th>Type Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relations file</td>
<td>ProInputFileRead()</td>
<td>PRO_RELATION_FILE</td>
</tr>
<tr>
<td>Program file</td>
<td>ProInputFileRead()</td>
<td>PRO_PROGRAM_FILE</td>
</tr>
<tr>
<td>Configuration options file</td>
<td>ProInputFileRead()</td>
<td>PRO_CONFIG_FILE</td>
</tr>
<tr>
<td>Setup file</td>
<td>ProInputFileRead()</td>
<td>PRO_DWG_SETUP_FILE</td>
</tr>
<tr>
<td>Spool file</td>
<td>ProInputFileRead()</td>
<td>PRO_SPOOL_FILE</td>
</tr>
<tr>
<td>Cable Parameters file</td>
<td>ProInputFileRead()</td>
<td>PRO_CABLE_PARAMS_FILE</td>
</tr>
<tr>
<td>Connector Parameters file</td>
<td>ProInputFileRead()</td>
<td>PRO_CONNECTOR_PARAMS_FILE</td>
</tr>
<tr>
<td>Model Tree Configuration file</td>
<td>ProInputFileRead()</td>
<td>PRO_ASSEM_TREE_CFG_FILE</td>
</tr>
<tr>
<td>Wirelist file</td>
<td>ProInputFileRead()</td>
<td>PRO_WIRELIST_FILE</td>
</tr>
</tbody>
</table>

**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 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 component path (`memb_id_tab`)
- `arg2`—Represents the component number (`memb_num`)

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**

<table>
<thead>
<tr>
<th>Import Format</th>
<th>Pro/TOOLKIT Functions</th>
<th>Type Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEP file</td>
<td>Pro2dImportCreate(),</td>
<td>PRO_STEP_FILE</td>
</tr>
<tr>
<td></td>
<td>Pro2dImportAppend()</td>
<td></td>
</tr>
<tr>
<td>IGES (2D) file</td>
<td>Pro2dImportCreate(),</td>
<td>PRO_IGES_FILE</td>
</tr>
<tr>
<td></td>
<td>Pro2dImportAppend()</td>
<td></td>
</tr>
<tr>
<td>DXF file</td>
<td>Pro2dImportCreate(),</td>
<td>PRO_DXF_FILE</td>
</tr>
<tr>
<td></td>
<td>Pro2dImportAppend()</td>
<td></td>
</tr>
<tr>
<td>DWG file</td>
<td>Pro2dImportCreate(),</td>
<td>PRO_DWG_FILE</td>
</tr>
<tr>
<td></td>
<td>Pro2dImportAppend()</td>
<td></td>
</tr>
<tr>
<td>SET file</td>
<td>Pro2dImportAppend()</td>
<td>PRO_SET_FILE</td>
</tr>
<tr>
<td>MEDUSA file</td>
<td>Pro2dImportCreate()</td>
<td>PRO_MEDUSA_FILE</td>
</tr>
<tr>
<td>IGES (2D) file</td>
<td>ProInputFileRead()</td>
<td>PRO_IGES_SECTION</td>
</tr>
</tbody>
</table>

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()`
- `ProIntfimportLayerFilter()`

<table>
<thead>
<tr>
<th>Import Format</th>
<th>Pro/TOOLKIT Functions</th>
<th>Type Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACIS file</td>
<td><code>ProIntfimportModelCreate()</code></td>
<td><code>PRO_INTF_IMPORT_ACIS</code></td>
</tr>
<tr>
<td>CADDs file</td>
<td><code>ProIntfimportModelCreate()</code></td>
<td><code>PRO_INTF_IMPORT_CADDs</code></td>
</tr>
<tr>
<td>CATIA file</td>
<td><code>ProIntfimportModelCreate()</code></td>
<td><code>PRO_INTF_IMPORT_CATIA</code></td>
</tr>
<tr>
<td>CATIA (.model) file</td>
<td><code>ProIntfimportModelCreate()</code></td>
<td><code>PRO_INTF_IMPORT_CATIA_MODEL</code></td>
</tr>
<tr>
<td>CATIA (.session) file</td>
<td><code>ProIntfimportModelCreate()</code></td>
<td><code>PRO_INTF_IMPORT_CATIA_SESSION</code></td>
</tr>
<tr>
<td>Pro/Desktop file (.pdt)</td>
<td><code>ProIntfimportModelCreate()</code></td>
<td><code>PRO_INTF_IMPORT_DESKTOP</code></td>
</tr>
<tr>
<td>DXF file</td>
<td><code>ProIntfimportModelCreate()</code></td>
<td><code>PRO_INTF_IMPORT_DXF</code></td>
</tr>
<tr>
<td>ICEM file</td>
<td><code>ProIntfimportModelCreate()</code></td>
<td><code>PRO_INTF_IMPORT_ICEM</code></td>
</tr>
<tr>
<td>IGES file</td>
<td><code>ProIntfimportModelCreate()</code></td>
<td><code>PRO_INTF_IMPORT_IGES</code></td>
</tr>
<tr>
<td>Neutral file</td>
<td><code>ProIntfimportModelCreate()</code></td>
<td><code>PRO_INTF_IMPORT_NEUTRAL</code></td>
</tr>
<tr>
<td>Parasolid-based CADDs system file</td>
<td><code>ProIntfimportModelCreate()</code></td>
<td><code>PRO_INTF_IMPORT_PARASOLID</code></td>
</tr>
<tr>
<td>PDGS file</td>
<td><code>ProIntfimportModelCreate()</code></td>
<td><code>PRO_INTF_IMPORT_PDGS</code></td>
</tr>
<tr>
<td>POLTXT file</td>
<td><code>ProIntfimportModelCreate()</code></td>
<td><code>PRO_INTF_IMPORT_POLTXT</code></td>
</tr>
<tr>
<td>SET file</td>
<td><code>ProIntfimportModelCreate()</code></td>
<td><code>PRO_INTF_IMPORT_SET</code></td>
</tr>
<tr>
<td>STEP file</td>
<td><code>ProIntfimportModelCreate()</code></td>
<td><code>PRO_INTF_IMPORT_STEP</code></td>
</tr>
<tr>
<td>STL file</td>
<td><code>ProIntfimportModelCreate()</code></td>
<td><code>PRO_INTF_IMPORT_STL</code></td>
</tr>
<tr>
<td>VDA file</td>
<td><code>ProIntfimportModelCreate()</code></td>
<td><code>PRO_INTF_IMPORT_VDA</code></td>
</tr>
</tbody>
</table>
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 `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 section Modifying the Imported Layers.

<table>
<thead>
<tr>
<th>Import Format</th>
<th>Pro/TOOLKIT Functions</th>
<th>Type Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRML file</td>
<td>ProIntfimportModelCreate()</td>
<td>PRO_INTF_IMPORT_VRML</td>
</tr>
</tbody>
</table>
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 4: Importing a 2D Model

This example demonstrates how to use the function `ProIntfimportSourceTypeGet()` to determine the type of the model before importing it.

```c
#include <ProToolkit.h>
#include <ProMenu.h>
#include <ProMdl.h>
#include <ProIntfimport.h>

#define MSGFIL L##"msg_uginterface.txt"

/*====================================================================*
FUNCTION: UserCATIAImportNewModel()
PURPOSE: Imports a CATIA file to an appropriate model type
/*====================================================================*/
int UserCATIAImportNewModel(ProPath input_file, ProName new_model_name, ProMdl* model)
{
    ProIntfImportType import_type = PRO_INTF_IMPORT_CATIA;
    ProError status;
    ProMdlType mdl_type;
    ProMdl created_model;

    *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(s)
 /*--------------------------------------------------------------------*/
    status = ProIntfimportModelCreate(input_file,
                                       import_type,
                                       mdl_type
                                       new_model_name,
                                       NULL, NULL,
                                       &created_model);

    if (status == PRO_TK_NO_ERROR)
    {
        *model = created_model;
        return PRO_TK_NO_ERROR;
    }
    else
Example 5: Importing a 2D 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().

```c
#include <ProToolkit.h>
#include <ProMenu.h>
#include <ProMdl.h>
#include <ProIntfimport.h>

#define MSGFIL L##"msg_uginterface.txt"

/*====================================================================*\nSTRUCTURE: UserImportLayerFilterdata
PURPOSE:    Contains user data to be passed to the import layer filter
function.
/*====================================================================*/
typedef struct
{
    ProName find_layer;
} UserImportLayerFilterdata;

/*====================================================================*\nFUNCTION: 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);

    return status;
}
#undef MSGFIL
```
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 a new model while filtering
layers
/*====================================================================*/
int UserIGESImportNewModelViaFilter(ProPath input_file,
                                    ProName new_model_name,
                                    ProName excluded_layer,
                                    ProMdl* model)
{
    ProError status;
    ProMdl created_model;

    /*--------------------------------------------------------------------*
    Set up the layer filter data - search for needed layer
    *--------------------------------------------------------------------*/
    UserImportLayerFilterdata data;
    ProUtilWstrcpy (data.find_layer, excluded_layer);
    *model = NULL;

    /*--------------------------------------------------------------------*
    Import IGES file
    *--------------------------------------------------------------------*/
    status = ProIntfimportModelCreate(input_file,
                                       PRO_INTF_IMPORT_IGES,
                                       PRO_MDL_PART,
                                       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
This chapter describes how to create import features in Pro/ENGINEER.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating Import Features from Files</td>
<td>51 - 2</td>
</tr>
<tr>
<td>Creating Import Features from Arbitrary Geometric Data</td>
<td>51 - 3</td>
</tr>
<tr>
<td>Redefining the Import Feature</td>
<td>51 - 15</td>
</tr>
<tr>
<td>Import Feature Properties</td>
<td>51 - 16</td>
</tr>
<tr>
<td>Extracting Pro/ENGINEER Geometry as Interface Data</td>
<td>51 - 18</td>
</tr>
</tbody>
</table>
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()

The function ProIntfDataSourceInit() is used to build the interface data source required by the function ProImportfeatCreate().

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:

<table>
<thead>
<tr>
<th>Type Constant</th>
<th>Import Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO_INTF_NEUTRAL_FILE</td>
<td>Neutral file</td>
</tr>
<tr>
<td>PRO_INTF_IGES</td>
<td>IGES 3D file</td>
</tr>
<tr>
<td>PRO_INTF_STEP</td>
<td>STEP file</td>
</tr>
<tr>
<td>PRO_INTF_VDA</td>
<td>VDA file</td>
</tr>
<tr>
<td>PRO_INTF_SET</td>
<td>SET file</td>
</tr>
<tr>
<td>PRO_INTF_PDGS</td>
<td>PDGS file</td>
</tr>
<tr>
<td>PRO_INTF_CATIA</td>
<td>CATIA file</td>
</tr>
<tr>
<td>PRO_INTF_DESKTOP</td>
<td>Pro/DESKTOP file</td>
</tr>
<tr>
<td>PRO_INTF_ICEM</td>
<td>ICEM file</td>
</tr>
<tr>
<td>PRO_INTF_ACIS(*.sat)</td>
<td>ACIS format file</td>
</tr>
<tr>
<td>PRO_INTF_DXF</td>
<td>DXF file</td>
</tr>
<tr>
<td>PRO_INTF_CDRS</td>
<td>CDRS file</td>
</tr>
<tr>
<td>PRO_INTF_STL</td>
<td>STL file</td>
</tr>
<tr>
<td>PRO_INTF_VRML</td>
<td>VRML file</td>
</tr>
<tr>
<td>PRO_INTF_PARASOLID</td>
<td>Parasolid-based CADDs system file</td>
</tr>
<tr>
<td>PRO_INTF_AI</td>
<td>AI file</td>
</tr>
</tbody>
</table>
• **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 functions returns the handle to the ProIntfDataSource which should be passed to the function **ProImportfeatCreate()**.

The function **ProImportfeatCreate()** is used to create the import feature in the Pro/ENGINEER solid model. The input arguments of this method 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 interface data source.

• **p_csys**—Specifies the co-ordinate system of the part with which the user wants to align the import feature. When this is NULL, the function creates a new default coordinate system in the Pro/ENGINEER model and the import feature will be aligned with respect to this coordinate system.

• **p_attributes**—Attributes for the creation of the new import feature. Please see the section Import Feature Attributes for more information.

This function returns the ProFeature handle for the created import feature.

---

**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 are 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**

Functions Introduced:

- **ProIntfDataAlloc()**
  
  Use the function `ProIntfDataAlloc()` to allocate memory for the interface data structure.

**Adding Surfaces**

Functions 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()
- ProPlanedatalInit()
- ProCylinderdataInit()
- ProConedatalInit()
- ProTorusdataInit()
- ProSrfrevdataInit()
- ProTabcyldatalInit()
- ProRulsrfdataInit()
- ProSplinesrfdataInit()
- ProCylsplsrfdataInit()
- ProBsplinesrfdataInit()
- ProFilsrfdataInit()

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 ProSurfaceshapedata and assign it to variable p_surf_shape of function ProSurfacedataInit(). Ensure that the function used to create the ProSurfaceshapedata matches with the ProSrftype value used in this function.

- ProPlanedataInit()
- ProCylinderdataInit()
- ProConedataInit()
- ProTorusdataInit()
- ProSrfrevdataInit()
- ProTabcyldataInit()
- ProRulsrfdataInit()
- ProSplinesrfdataInit()
- ProCylsplsrfrfdataInit()
- ProBsplinesrfdataInit()
- ProFilrsrfdataInit()

Refer to the ‘Geometry Representations’ appendix for more information on how to use the above functions.
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()**
- **ProContourdataEdgeIdArraySet()**

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**

Functions 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()**
- **ProEdgedatalnit()**
- **ProCurvedataAlloc()**
- **ProLinedatalnit()**
- **ProArcdatalnit()**
- **ProSplinedatalnit()**
- **ProBsplinedatalnit()**

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. Creates a copy of the structure ProCurvedata. Use the function **ProCurvedataFree** to free the original data.

Specify the handle returned by the function **ProLinedataInit()**, or **ProArdedataInit()**, or **ProSplinedataInit()**, or **ProBsplinedataInit()**.

This function **ProEdgedataInit()** returns the edge data structure.

**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()`
- `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 `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.

### Appending the Edge Data to the Interface Data

**Functions 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 *ProDatum* object.
    
    – **ProDatumCsysdataInit()**
    
    – **ProDatumCurvedataInit()**
    
    – **ProDatumPlanedataInit()**

  These functions return the handle to the populated datum 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.
- `pDatumdata`—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()`
- `ProIntfDataFree()`

Use the function `ProIntfDataSourceInit()` to build the interface data source required for the function `ProImportfeatCreate()`. The input arguments 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`. This handle must be passed as the input argument for function `ProImportfeatCreate()`.

Use the function `ProImportfeatCreate()` to create the import features in a Pro/ENGINEER model. The input arguments 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. The handle returned by function `ProIntfDataSourceInit()` should be passed to this variable.
• **p_csys**—Specifies the co-ordinate system of the part with which you want to align the import feature. If the value of this variable is specified as NULL, the function creates a new default coordinate system in the Pro/ENGINEER model. The import feature will be aligned with respect to this new coordinate system.

• **p_attributes**—Specifies attributes to create the new import feature. Refer to the section Import Feature Attributes for more information.

The function `ProImportfeatCreate()` returns the handle to the created import feature in the Pro/ENGINEER model.

Use the function `ProIntfDataFree()` to free 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
- 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()`.
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 `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.
Asynchronous Mode

This chapter explains using Pro/TOOLKIT in Asynchronous Mode.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>52 - 2</td>
</tr>
<tr>
<td>Simple Asynchronous Mode</td>
<td>52 - 3</td>
</tr>
<tr>
<td>Full Asynchronous Mode</td>
<td>52 - 8</td>
</tr>
</tbody>
</table>
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 the ‘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). Note that 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. Because Pro/ENGINEER calls user_initialize() when applications start, your synchronous application can take control by initiating all operations in user_initialize() (thus forestalling any user interaction). This 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.

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:

- ProEngineerConnect()
- ProEngineerConnectWS()
- ProEngineerDisconnect()
- ProEngineerConnectIdExtract()

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.

```c
/*====================================================================*
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;
```
Asynchronous Mode

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);

Asynchronous Mode
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()
• 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 ‘Menus’ chapter.)

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**

```c
#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(
    ProeTerminationStatus 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

```c
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

```c
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," 	
");

        /*--------------------------------------------------------------------*
 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);
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.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProArgument and Argument Management</td>
<td>53 - 2</td>
</tr>
<tr>
<td>Creating Pro/TOOLKIT DLL Task Libraries</td>
<td>53 - 3</td>
</tr>
<tr>
<td>Launching Synchronous J-Link Applications</td>
<td>53 - 9</td>
</tr>
</tbody>
</table>
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:

```c
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 the `ProTkdllFunction()`:

```c
typedef ProError (*ProTkdllFunction) (ProArgument* inputs, ProArgument** outputs);
```

Use the preprocessor macro `PRO_TK_DLL_EXPORT` on any function that must be accessible to external applications. This macro provides platform specific instructions to the compiler to make the function visible to external applications of the function. This macro must be placed on the function prototype, if it exists, and on 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, an application 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 for assigning 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 to 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.

```c
#include <ProToolkit.h>
#include <ProMdl.h>
#include <ProArray.h>
#include <ProModelitem.h>
#include <ProImportfeat.h>
#include <ProToolkitDll.h> /* for ProArgument */

/*====================================================================*
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);
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;
        status = ProArgumentByLabelGet (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);
        if (status != PRO_TK_NO_ERROR)
            return PRO_TK_BAD_INPUTS;
        width = data.v.d;
        status = ProArgumentByLabelGet (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);
    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);
 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);
 if (status == PRO_TK_NO_ERROR)
 {
     if (data.v.b)
         attr.cut_or_add = 1;
 }

Create import feature with parameters

```c
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);

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

```c
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.
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`. 
This chapter describes how to use Help files.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>How Help Files Work</td>
<td>54 - 2</td>
</tr>
<tr>
<td>Creating Help Files</td>
<td>54 - 3</td>
</tr>
</tbody>
</table>
How Help Files Work

Pro/ENGINEER uses the Netscape Navigator browser as the help file viewer. You need two files to make a help file display on the screen:

- `helpfiles.html`—Contains the description to be displayed on the screen. These files are in HTML format.
- `proe.h`—Contains the cross-referencing between a specific “menu_title menu_item” combination and the HTML help file to be displayed. This file must be located in the directory `<doc_loadpoint>/html/$LANG`.

Pro/ENGINEER registers its own help files in the `ptc_help.txt` file located in the directory `<PROENGINEER_loadpoint>/text/`. This file defines the PTC_DOCS variable that specifies the loadpoint of all the Pro/ENGINEER help files. You can append your application help file loadpoint to PTC_DOCS either in the `ptc_help.txt` file, or on the command line (by setting the environment variable PTC_DOCS). If you set the environment variable PTC_DOCS, its value is appended to the area defined in the `ptc_help.txt` file.

For example, set the environment variable as follows:

**On UNIX systems:**

```
setenv PTC_DOCS /usr/home/tk_appl/text
```

**On NT systems:**

```
set PTC_DOCS c:/usr/home/tk_appl/text
```

In this example, the file `proe.h` must be located in the directory `/usr/home/tk_appl/text/html/usascii/`, if the environment variable LANG is set to English.

PTC recommends that the location for the `<doc_loadpoint>` is the text directory of the Pro/TOOLKIT application. In this way, the application and its help files are all combined in one directory structure.

You can have multiple document loadpoint directories registered. This is useful if you will have more than one Pro/TOOLKIT application running at the same time. In this case, your PTC_DOCS environment variable will contain multiple loadpoint paths, separated by a tilde (~) without empty spaces. For example:

```
setenv PTC_DOCS/usr/home/tk_help1/~usr/home/tk_help2
```
Another important environment variable is PTC_HELP, which contains the full path to the Netscape executable. This variable is optional because the Netscape executable is already registered with the value PTC_DOCS in the ptc_help.txt file. Pro/ENGINEER looks for the Netscape executable in the following locations, in order:

1. In the directory specified by the PTC_HELP variable.
2. Under the document loadpoint directory specified by the PTC_DOCS environment variable. The command is as follows:
   \[\text{<doc_loadpoint>/$PRO_MACHINE_TYPE/netscape/netscape}\]

When you click the right mouse button on a menu item, Pro/ENGINEER searches the proe.h file until it finds the corresponding “menu_title menu_item.” On that same line in the file is the HTML help file name. The system locates this file and displays it.

A sample of an proe.h for the pt_install_test application is as follows:

```html
Window-Name="ProHelp"
Window-Size="600,800"

<ID_MAP>
mains-install#test = "guides/install.html" : target =    "topic"
</Frame_group>
</ID_MAP>
```

The error “No help file found” might occur when you use help files. This error occurs when Pro/ENGINEER cannot locate the proper “menu_title menu_item” combination. The Netscape Navigator will warn you that it cannot find the file or directory.

**Creating Help Files**

To Create a Help File

1. Create the description for each menu item. Format the text in HTML.
2. Save the file to filename.html in the directory structure under $doc_loadpoint/html/$LANG.
3. Edit the `proe.h` file. Add the “menu_title menu_item” combination, making sure of the following:
   - Spaces in the menu title or menu item are replaced with a pound (#) sign.
   - The menu title and menu item are separated by an at-about (@) character.
   - Use only lowercase characters for the menu item and menu title.

If you have only modified an HTML help file, you do not need to change anything in the `proe.h` file.

4. To the right of the “menu_title menu_item” combination, enter the path and name of the HTML help file. Note that the path for HTML help files is with respect to the directory `<doc_loadpoint>/html/$LANG`.

5. Save the file `proe.h` under `<doc_loadpoint>/html/$LANG`.

6. The new or modified help files are now available when you run the Pro/TOOLKIT application program.
Summary of Technical Changes

This appendix contains a list of new and enhanced functions for Pro/TOOLKIT under Pro/ENGINEER Wildfire. See the 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.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Updates for Pro/TOOLKIT Wildfire</td>
<td>A - 2</td>
</tr>
<tr>
<td>Superseded Functions</td>
<td>A - 20</td>
</tr>
<tr>
<td>Miscellaneous Technical Changes</td>
<td>A - 21</td>
</tr>
</tbody>
</table>

Note: Reference information on all functions is available in the Pro/TOOLKIT APIWizard online browser. Use the APIWizard Search function to find information on a function. See section Online Documentation — Pro/TOOLKIT APIWizard for information on the APIWizard.
Updates for Pro/TOOLKIT Wildfire

This section describes new functions in Pro/TOOLKIT Wildfire.

It also includes critical functions that require investigation of existing applications and adjustment of existing code.

Critical Functions

External Analysis

The ProAnalysisGeomitem structure for Pro/ENGINEER Wildfire has been expanded to allow creation of surface geometry. The new member ProAnalysisSrfData allows creation of surface and edge geometry from an external analysis feature.

Feature Element Trees

The following feature element trees have been altered. Code that attempts to create, modify or access information about the following features using element trees have to be updated to run with Pro/ENGINEER Wildfire. Applications built using Release 2001 and earlier element trees will not function properly with Pro/ENGINEER Wildfire.

<table>
<thead>
<tr>
<th>Feature Type or Form</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datum plane</td>
<td>New optional elements - does not require rewrite of existing applications. New elements may, however, be present in retrieved element trees.</td>
</tr>
<tr>
<td>Coordinate system</td>
<td>New element tree</td>
</tr>
<tr>
<td>Datum axis</td>
<td>New element tree</td>
</tr>
<tr>
<td>Datum point</td>
<td>New element tree</td>
</tr>
<tr>
<td>Hole</td>
<td>New sketcher element, new optional elements</td>
</tr>
<tr>
<td>Section</td>
<td>Modified element tree</td>
</tr>
<tr>
<td>Extrude</td>
<td>New element tree</td>
</tr>
<tr>
<td>Revolve</td>
<td>New element tree</td>
</tr>
<tr>
<td>Edge Chamfer</td>
<td>New element tree. This feature type is not currently supported in Pro/ENGINEER Wildfire</td>
</tr>
</tbody>
</table>
Datum Axis

The datum axis element tree has been modified to work with the new Pro/ENGINEER Wildfire user interface. The new axis element tree is published in the header file ProDtmAxis.h. Applications creating, modifying or accessing axis features via the element tree must be updated to the new tree definition.

Datum Coordinate System

The element tree for coordinate systems has been modified to fit with the new Pro/ENGINEER Wildfire user interface. The new element tree is published in the header file ProDtmCsys.h. Applications using element trees for creating, modifying, and accessing coordinate system features must be updated to the new element tree.

Datum Point

The datum point element tree has been redesigned to fit the new Pro/ENGINEER Wildfire user interface. The updated datum point element tree is published in the header file ProDtmPnt.h. Applications creating, modifying, or accessing the contents of datum point features must be updated to refer to the new tree definition.

Endge Chamfer

The edge chamfer element tree has been modified to work with the new Pro/ENGINEER Wildfire user interface. The new edge chamfer element tree contains special application elements that are not currently supported using Pro/TOOLKIT. PTC plans to restore access to the edge chamfer element tree as soon as these elements can be supported in Pro/TOOLKIT application code.

Note: The corner chamfer element tree is unchanged and is still supported.

Extrude

The extrude element tree has been modified to work with the new Pro/ENGINEER Wildfire user interface. The new extrude element tree is published in the header file ProExtrude.h. Applications creating, modifying or accessing extrude features via the element tree must be updated to the new tree definition.
The new extrude element tree allows creation of extruded protrusions, cuts and surfaces. Solid features can be toggled to become thin. Surface features can apply needed end conditions. Extrude features can be redefined using the element tree to apply these different attributes as needed.

The elements PRO_E_STD_DIRECTION and PRO_E_STD_MTRLSLIDE are directly dependent upon the presence of a fully defined PRO_E_STD_SECTION element tree, including PRO_E_SKETCHER. Values assigned to these elements before fully defining the PRO_E_STD_SECTION will be ignored.

Flat Surface

The element tree for the Flat surface (called Fill in the User Interface) has changed, because of the changes applied to the PRO_E_STD_SECTION element. Information on the element tree changes is published in the header file ProFlatSrf.h.

Hole

The hole element tree has been modified slightly to work with the new Pro/ENGINEER Wildfire user interface. The new hole element tree is published in the header file ProHole.h.

Applications creating, modifying, or accessing sketched holes should use the PRO_E_SKETCHER element in place of the PRO_E_HOLE_SKETCHER element in previous releases. The standard hole element tree also contains new elements that control exit countersink geometry.

Revolve

The revolve element tree has been modified to work with the new Pro/ENGINEER Wildfire user interface. The new revolve element tree is published in the header file ProRevolve.h. Applications creating, modifying or accessing revolve features via the element tree must be updated to the new tree definition.

The new revolve element tree allows creation of revolved protrusions, cuts and surfaces. Solid features can be toggled to become thin. Surface features can apply the required end conditions. Revolve features can be redefined using the element tree to apply these different attributes as needed.
The elements PRO_E_STD_DIRECTION and PRO_E_STD_MATRLSIDE are directly dependent upon the presence of a fully defined PRO_E_STD_SECTION element tree, including PRO_E_SKETCHER. Values assigned to these elements before fully defining the PRO_E_STD_SECTION will be ignored.

**Sketched Datum Curve**

The sketched datum curve element tree has been modified to work with the new Pro/ENGINEER Wildfire user interface. Changes to the element tree are limited to the changes in the definition of the PRO_E_STD_SECTION element, published in ProStdSection.h. Applications creating, modifying or accessing sketched datum curve features via the element tree must be updated to the new tree definition.

**Sketched Features**

The standard section element tree has been modified to work with the new Pro/ENGINEER Wildfire user interface. The new section element tree is published in the header file ProStdSection.h. Applications creating, modifying or accessing sections features via the element tree must be updated to the new tree definition. This affects the following feature types:

- Extrude
- Revolve
- Fill (flat surface)
- Datum curve (sketched)
- Datum point (sketched)
- Rib

Sketched feature element trees include run-time data allocated by Pro/ENGINEER that is necessary in establishing the feature. Thus the proper technique for creating sketched features using internal sketches is:

- Create the incomplete feature using `ProFeatureCreate()`.
- Extract the current feature tree using `ProFeatureElemtreeCreate()`.
- Extract and populate the PRO_E_SKETCHER element using ProSection functions.
- Redefine the feature using the extracted feature tree obtained from `ProFeatureElemtreeCreate()`.
Patterns

Fill Patterns

Pro/ENGINEER Wildfire introduces a new type of pattern called fill patterns. The function `proptn_get_pattern()` will contain a new pattern type, PRO_PTYPE_FILL, for fill patterns. It will include one direction containing all pattern members. Fill patterns are not currently supported via element trees for creation and inspection. The function `ProPatternElemtreeCreate()` will return PRO_TK_NOT_IMPLEMENTED for fill patterns. Element tree support for fill pattern creation and inspection will be provided as soon as possible.

Headers

Patterns are treated as features in Pro/ENGINEER Wildfire. A new feature type, PRO_E_PATTERN_HEAD, is used for the pattern header feature. The results of the pattern header feature for users of previous versions of Pro/TOOLKIT is as follows:

- Models that contain patterns automatically get one extra feature in the regeneration list, of type PRO_FEAT_PATTERN_HEAD. This changes the feature numbers of all subsequent features, including those in the pattern.

The pattern access functions `proptn_get_pattern()`, `ProPatternLeaderGet()` are unaffected by the addition of the pattern header feature.

ProAsmcompMdlNameGet()

The function `ProAsmcompMdlNameGet()` previously took a ProName to contain the output argument. This has been increased to ProFamilyName to accommodate family table instance names. The argument you provide to this function must be at least PRO_FAMILY_NAME_SIZE wide characters or a memory overwrite may result.

ProFeatureCreatePreAction

The Pro/ENGINEER notification PRO_FEATURE_CREATE_PRE is called for the following datum features:

- Datum planes
- Datum axes
- Datum points
• Coordinate systems
• More datum curve types

In Release 2001 these features did not trigger a PRO_FEATURE_CREATE_PRE notification.

Selection

The functions ProSelect() and pro_select() enable the user to make one or more selections based on the input filters and options; however, the user interface for this has migrated to a new mechanism and the Get Select() menu is no longer displayed. It is therefore more important to provide detailed instructions to the user through the message window or dialogs that the application expects the user to make a selection.

System Colors

Pro/ENGINEER Wildfire features a new system of standard colors. Applications using any of the standard color constants defined in ProToolkit.h will experience changes in application created graphics, text, or other colors. Applications using customized RGB values for colors will be impacted because of reduced contrast between the custom application color and entities in the window.

Top Level Menu Manager Menus

The top-level menu manager menus for all modes have been migrated to positions in the Toolbar and Menubar. Applications will not be able to add menu buttons directly to these menus. Many menu manager submenus for these modes remain active in this release; however PTC recommends using custom menubar, toolbar, or dialog user interfaces instead to prevent impacts in future releases.

Version String

The version name of this Pro/ENGINEER release is "Wildfire". This string will be passed to Pro/TOOLKIT applications through their user_initialize() function. Applications using numerical routines to attempt to determine version compatibility will not be successful.

Pro2dImportCreate()

The function Pro2dImportCreate() no longer supports the SET format. To import a two-dimensional SET file, create the new drawing or model, and use the function Pro2dImportAppend().
Functions ProFemmeshExport(), pro_export_fea_mesh()

Pro/ENGINEER and Pro/TOOLKIT support for the FEM solver types COSMOS and PATRAN have been discontinued.

ProFileOpen()

The file and directory selection dialog box in Pro/ENGINEER Wildfire is capable of finding items from URLs and pseudo-URLs from sources such as Pro/INTRALINK, Windchill, and Groove.

ProMdlDataGet(), prodb_get_object_info()

Pro/ENGINEER Wildfire is capable of retrieving models from URLs and pseudo-URLs from sources such as HTTP, Pro/INTRALINK, Windchill, and Groove.

New Functions

The following table lists the Pro/TOOLKIT functions that are new for the Wildfire release.

<table>
<thead>
<tr>
<th>New Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly</td>
<td></td>
</tr>
<tr>
<td>ProAsmcompAssemble()</td>
<td>Assembles a component to the assembly or sub-assembly using the parametric constraints</td>
</tr>
<tr>
<td>ProAsmcompConstraintsGet()</td>
<td>Specifies the position of the component in an assembly relative to other components.</td>
</tr>
<tr>
<td>ProAsmcompConstraintsSet()</td>
<td></td>
</tr>
<tr>
<td>ProAsmcompPositionSet()</td>
<td>Specifies the initial position of the component before constraints are applied.</td>
</tr>
<tr>
<td>ProAsmcompPositionGet()</td>
<td></td>
</tr>
<tr>
<td>ProAsmcompIsFrozen()</td>
<td>Determines if the specified component is frozen.</td>
</tr>
<tr>
<td>ProAsmcompIsPackaged()</td>
<td>Determines if the specified component is completely packaged.</td>
</tr>
<tr>
<td>ProAsmcompIsUnderconstrained()</td>
<td>Determines if the specified component is underconstrained.</td>
</tr>
<tr>
<td>ProAsmcompconstraintAlloc()</td>
<td>Allocates memory for the constraint data structure</td>
</tr>
<tr>
<td>ProAsmcompconstraintTypeGet()</td>
<td>Specifies the constraint type of the specified constraint</td>
</tr>
<tr>
<td>ProAsmcompconstraintTypeSet()</td>
<td></td>
</tr>
<tr>
<td>New Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ProAsmcompconstraintAsmreferenceGet()</td>
<td>Specifies the orientation of the assembly for a specified assembly component constraint.</td>
</tr>
<tr>
<td>ProAsmcompconstraintAsmreferenceSet()</td>
<td></td>
</tr>
<tr>
<td>ProAsmcompconstraintCompreferenceGet()</td>
<td>Specifies the orientation of the component for a specified assembly component constraint.</td>
</tr>
<tr>
<td>ProAsmcompconstraintCompreferenceSet()</td>
<td></td>
</tr>
<tr>
<td>ProAsmcompconstraintOffsetGet()</td>
<td>Specifies the offset value for the Mate or Align constraint type.</td>
</tr>
<tr>
<td>ProAsmcompconstraintOffsetSet()</td>
<td></td>
</tr>
<tr>
<td>ProAsmcompconstraintAttributesGet()</td>
<td>Specifies the constraint attributes for the specified assembly component constraint.</td>
</tr>
<tr>
<td>ProAsmcompconstraintAttributesSet()</td>
<td></td>
</tr>
<tr>
<td>ProAsmcompconstraintuserdataGet()</td>
<td>Specifies the user data for the given constraint.</td>
</tr>
<tr>
<td>ProAsmcompconstraintuserdataSet()</td>
<td></td>
</tr>
<tr>
<td>ProAsmcompconstraintFree()</td>
<td>Frees the constraint data structure from the memory.</td>
</tr>
<tr>
<td>ProAsmcompconstraintArrayFree()</td>
<td>Provides a single function to use to free an entire ProArray of ProAsmcompconstraint structures.</td>
</tr>
</tbody>
</table>

**Cabling**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProCableCosmeticFeatureCreate()</td>
<td>Creates a cabling cosmetic feature like a marker, a tape or tie wrap.</td>
</tr>
<tr>
<td>ProCableTapeWindsGet()</td>
<td>Provides access to the number of winds in a tape cosmetic feature.</td>
</tr>
<tr>
<td>ProCableTapeWindsSet()</td>
<td></td>
</tr>
<tr>
<td>ProCableSectionDelete()</td>
<td>Deletes the section of cables that lie between designated locations.</td>
</tr>
</tbody>
</table>

**Configuration Options**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProConfigoptArrayGet()</td>
<td>Provides an array containing all the values set for multiple valued configuration options, and provides support for PRO_PATH_SIZE strings.</td>
</tr>
</tbody>
</table>

**Data Exchange**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProIntf3DFileWrite()</td>
<td>Exports a Pro/ENGINEER model to the specified output format.</td>
</tr>
<tr>
<td><strong>New Function</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>---------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ProOutputBrepRepresentationAlloc()</td>
<td></td>
</tr>
<tr>
<td>ProOutputBrepRepresentationFlagsSet()</td>
<td></td>
</tr>
<tr>
<td>ProOutputBrepRepresentationIsSupported()</td>
<td></td>
</tr>
<tr>
<td>ProOutputBrepRepresentationFree()</td>
<td></td>
</tr>
<tr>
<td>ProOutputInclusionAlloc()</td>
<td>Supports export of 3D models.</td>
</tr>
<tr>
<td>ProOutputInclusionFlagsSet()</td>
<td></td>
</tr>
<tr>
<td>ProOutputInclusionFree()</td>
<td></td>
</tr>
<tr>
<td>ProOutputLayerOptionsAlloc()</td>
<td></td>
</tr>
<tr>
<td>ProOutputLayerOptionsAutoidSet()</td>
<td></td>
</tr>
<tr>
<td>ProOutputLayerOptionsSetupfileSet()</td>
<td></td>
</tr>
<tr>
<td>ProOutputLayerOptionsFree()</td>
<td></td>
</tr>
<tr>
<td>ProOutputAssemblyConfigurationIsSupported()</td>
<td></td>
</tr>
<tr>
<td>ProIntfimportModelCreate()</td>
<td>Imports files of other formats and creates a new model.</td>
</tr>
<tr>
<td>ProIntfimportSourceTypeGet()</td>
<td>Supports Import of 2D models.</td>
</tr>
<tr>
<td>ProIntfimportLayerFilter()</td>
<td></td>
</tr>
<tr>
<td>ProProductviewExport()</td>
<td>Supports export to the Productview, that is, .OL and .ED file formats.</td>
</tr>
<tr>
<td>ProSoildShrinkwrapCreate()</td>
<td>Exports the specified model as a Shrinkwrap model.</td>
</tr>
<tr>
<td>New Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>ProShrinkwrapoptionsAlloc()</td>
<td></td>
</tr>
<tr>
<td>ProShrinkwrapoptionsFree()</td>
<td></td>
</tr>
<tr>
<td>ProShrinkwrapoptionsQualitySet()</td>
<td></td>
</tr>
<tr>
<td>ProShrinkwrapoptionsAutoholefillingSet()</td>
<td></td>
</tr>
<tr>
<td>ProShrinkwrapoptionsIgnoreskeletonSet()</td>
<td></td>
</tr>
<tr>
<td>ProShrinkwrapoptionsIgnorequiltsSet()</td>
<td></td>
</tr>
<tr>
<td>ProShrinkwrapoptionsAssignmasspropsSet()</td>
<td></td>
</tr>
<tr>
<td>ProShrinkwrapoptionsDatumrefsSet()</td>
<td></td>
</tr>
<tr>
<td>ProShrinkwrapoptionsIgnoresmallsurfacesSet()</td>
<td></td>
</tr>
<tr>
<td>ProShrinkwrapoptionsAdditionalsurfacesSet()</td>
<td></td>
</tr>
<tr>
<td>ProShrinkwrapoptionsFacetedformatSet()</td>
<td></td>
</tr>
<tr>
<td>ProShrinkwrapoptionsFramesFileSet()</td>
<td></td>
</tr>
<tr>
<td>ProShrinkwrapoptionsAdditionalcomponentsSet()</td>
<td></td>
</tr>
</tbody>
</table>

**Dimensions**

| ProSymbolDesignate()                             | Designates dimensions, dimension tolerances and surface finish to the specified model. |
| ProSymbolUndesignate()                           |                                               |
| ProSymbolDesignationVerify()                     |                                               |
| ProDimensionIsBasic()                            | Specifies whether the dimension is a basic dimension. |
| ProDimensionIsInspection()                       | Specifies whether the dimension is an inspection dimension. |

**Drawings**

<p>| ProDrawingDtsymdefRetrieve()                    | Supports retrieval of symbols from the system symbol directory at an arbitrary subdirectory location. |</p>
<table>
<thead>
<tr>
<th><strong>New Function</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>ProDtlentitydataIsConstruction()</td>
<td>Provides access and modifies the flag that controls whether the entity is created normal or as a construction entity.</td>
</tr>
<tr>
<td>ProDtlentitydataConstructionSet()</td>
<td></td>
</tr>
<tr>
<td>ProDtlentitydataIsHidden()</td>
<td>Provides access and modifies the flag that controls whether the entity is created normal or as a hidden entity.</td>
</tr>
<tr>
<td>ProDtlentitydataHiddenSet()</td>
<td></td>
</tr>
<tr>
<td>ProDtlsyminstdataGroupoptionsSet()</td>
<td>Provides access the groups contained in a detail symbol definition.</td>
</tr>
<tr>
<td>ProDtlsymsgroupSubgroupsCollect()</td>
<td></td>
</tr>
<tr>
<td>ProDtlsymsgroupDataGet()</td>
<td>Specify information about available groups in the symbol definition.</td>
</tr>
<tr>
<td>ProDtlsymsgroupdataNameGet()</td>
<td></td>
</tr>
<tr>
<td>ProDtlsymsgroupdataItemsCollect()</td>
<td></td>
</tr>
<tr>
<td>ProDtlsymsgroupParentGet()</td>
<td></td>
</tr>
<tr>
<td>ProDtlsymsgroupLevelIsExclusive()</td>
<td></td>
</tr>
<tr>
<td>ProDtlsymsgroupdataAlloc()</td>
<td>Modifies the set of groups in the symbol definition.</td>
</tr>
<tr>
<td>ProDtlsymsgroupdataNameSet()</td>
<td></td>
</tr>
<tr>
<td>ProDtlsymsgroupdataItemsSet()</td>
<td></td>
</tr>
<tr>
<td>ProDtlsymsgroupdataItemAdd()</td>
<td></td>
</tr>
<tr>
<td>ProDtlsymsgroupFree()</td>
<td></td>
</tr>
<tr>
<td>ProDtlsymsgroupSubgroupCreate()</td>
<td></td>
</tr>
<tr>
<td>ProDtlsymsgroupModify()</td>
<td></td>
</tr>
<tr>
<td>ProDtlsymsgroupDelete()</td>
<td></td>
</tr>
<tr>
<td>ProDtlsymsgroupLevelExclusiveSet()</td>
<td></td>
</tr>
<tr>
<td>ProDrawingDimensionIsDisplayed()</td>
<td>Specifies whether a drawing dimension is displayed in the drawing.</td>
</tr>
<tr>
<td>ProDrawingDimensionIsToleranceDisplayed()</td>
<td>Specifies whether the tolerance value is displayed on the drawing dimension.</td>
</tr>
<tr>
<td>ProDrawingViewIsBackground()</td>
<td>Determines if the specified view is a background view.</td>
</tr>
<tr>
<td>ProSelectionViewGet()</td>
<td>Outputs the ProView handle for an item in a drawing, that is not inside a solid view.</td>
</tr>
<tr>
<td>ProDrawingViewScaleIsUserdefined()</td>
<td>Returns a boolean value depending on whether the drawing has a user-assigned scale or not.</td>
</tr>
</tbody>
</table>
### New Function

<table>
<thead>
<tr>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provide access to the Z-clipping geometry reference.</td>
<td>ProDrawingViewZclippingGet()</td>
</tr>
<tr>
<td>ProDrawingViewZclippingSet()</td>
<td></td>
</tr>
</tbody>
</table>

### Embedded Web Browser

<table>
<thead>
<tr>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable you to find and change the size of the embedded browser in the Pro/ENGINEER window.</td>
<td>ProWindowBrowserSizeGet()</td>
</tr>
<tr>
<td>ProWindowBrowserSizeSet()</td>
<td></td>
</tr>
<tr>
<td>Enable you to find and change the URL displayed in the embedded browser in the Pro/ENGINEER window.</td>
<td>ProWindowURLGet()</td>
</tr>
<tr>
<td>ProWindowURLShow()</td>
<td></td>
</tr>
</tbody>
</table>

### Family Table

<table>
<thead>
<tr>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifies whether the instance has been locked by an external application.</td>
<td>ProFaminstanceIsExtLocked()</td>
</tr>
<tr>
<td>Specifies whether the instance has been verified, and whether the verification succeeded or failed.</td>
<td>ProFaminstanceIsVerified()</td>
</tr>
</tbody>
</table>

### Feature Element Trees

<table>
<thead>
<tr>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copies the selected section from one feature and adds it to another feature.</td>
<td>ProFeaturesSketchAdd()</td>
</tr>
<tr>
<td>Includes the element tree feature creation along with the section copy operation.</td>
<td>ProFeaturesSketchedCreate()</td>
</tr>
<tr>
<td>Returns information about a copied feature.</td>
<td>ProFeatureCopyinfoGet()</td>
</tr>
</tbody>
</table>

### Interoperability

<table>
<thead>
<tr>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locates an argument within a ProArray of ProArgument structures passed between applications.</td>
<td>ProArgumentByLabelGet()</td>
</tr>
<tr>
<td>Releases memory in an array of ProArgument structures.</td>
<td>ProArgumentProarrayFree()</td>
</tr>
<tr>
<td>Allocates and copy memory into the ProValuedata structure for a char* argument.</td>
<td>ProValuedataStringSet()</td>
</tr>
<tr>
<td>Allocates and copy memory into the ProValuedata structure for a wchar_t* argument.</td>
<td>ProValuedataWstringSet()</td>
</tr>
<tr>
<td>Allocate and copy memory into the ProValuedata structure for a ProMatrix argument.</td>
<td>ProValuedataTransformGet()</td>
</tr>
<tr>
<td>ProValuedataTransformSet()</td>
<td></td>
</tr>
<tr>
<td>New Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>ProToolkitDllLoad()</td>
<td>Registers and starts a Pro/TOOLKIT DLL.</td>
</tr>
<tr>
<td>ProToolkitTaskExecute()</td>
<td>Calls a properly designated function in the Pro/TOOLKIT DLL library.</td>
</tr>
<tr>
<td>ProToolkitDllUnload()</td>
<td>Shuts down a Pro/TOOLKIT DLL previously loaded by ProToolkitDllLoad().</td>
</tr>
<tr>
<td>ProToolkitDllIdGet()</td>
<td>Returns a string representation of the DLL application</td>
</tr>
<tr>
<td>ProToolkitDllHandleGet()</td>
<td>Obtains the Pro/TOOLKIT DLL handle using the string representation of the DLL application.</td>
</tr>
<tr>
<td>ProJlinkApplicationStart()</td>
<td>Registers and starts a J-Link library class.</td>
</tr>
<tr>
<td>ProJlinkTaskExecute()</td>
<td>Calls a properly designated task method in the J-Link library.</td>
</tr>
<tr>
<td>ProJlinkApplicationStop()</td>
<td>Shuts down a Pro/TOOLKIT DLL previously loaded by ProJlinkApplicationStart().</td>
</tr>
</tbody>
</table>

**Layers**

| ProLayeritemLayersGet()               | Specifies all the layers containing a given layer item. |

**Model Items**

| ProModelitemByNameInit()              | Returns a pointer to an item, given the name and type of the item. |
| ProModelitemDefaultnameGet()         | Specifies the default name for a new model item of a particular type. |
| ProModelitemNameCanChange()          | Identifies if the name of the model item can be modified. |
| ProModelitemUsernameDelete()         | Deletes the user-defined name of the model item from the Pro/ENGINEER database. |
| ProSolidRetrievalErrorsGet()         | Returns the data structure containing errors that occur during model retrieval. |

**ModelCheck**

<p>| ProModelcheckExecute()                | Enables modelcheck verification from the Pro/TOOLKIT application. |</p>
<table>
<thead>
<tr>
<th>New Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Parts</strong></td>
<td></td>
</tr>
<tr>
<td>ProPartDensityGet()</td>
<td>Modifies the density of a part without requiring assignment of a material.</td>
</tr>
<tr>
<td>ProPartDensitySet()</td>
<td></td>
</tr>
<tr>
<td><strong>ProValueData</strong></td>
<td></td>
</tr>
<tr>
<td>ProValuedataTransformGet()</td>
<td>Specifies how to access the matrix correctly.</td>
</tr>
<tr>
<td>ProValuedataTransformSet()</td>
<td></td>
</tr>
<tr>
<td><strong>Selection</strong></td>
<td></td>
</tr>
<tr>
<td>ProSelectionDrawingGet()</td>
<td>Provides a selected drawing.</td>
</tr>
<tr>
<td><strong>Sheetmetal</strong></td>
<td></td>
</tr>
<tr>
<td>ProSmtMdlIsFlatStateInstance()</td>
<td>Checks if the model is a flat state instance model.</td>
</tr>
<tr>
<td>ProFaminstanceIsFlatState()</td>
<td>Checks if the family instance of the model is a sheetmetal flat instance or not.</td>
</tr>
<tr>
<td><strong>Simplified Representations</strong></td>
<td></td>
</tr>
<tr>
<td>ProPartSimprepRetrieve()</td>
<td>Allows retrieval and activation of simplified representations.</td>
</tr>
<tr>
<td><strong>Solid</strong></td>
<td></td>
</tr>
<tr>
<td>ProSolidRetrievalErrorsGet()</td>
<td>Returns the data structure containing errors that occur during model retrieval.</td>
</tr>
<tr>
<td><strong>System Colors</strong></td>
<td></td>
</tr>
<tr>
<td>ProTextColorModify()</td>
<td>Enable you to select a different color to be used for text or graphics.</td>
</tr>
<tr>
<td>ProGraphicsColorModify()</td>
<td></td>
</tr>
<tr>
<td>ProColorByTypeGet()</td>
<td>Returns standard colormap entry corresponding to a particular entity in Pro/ENGINEER.</td>
</tr>
<tr>
<td><strong>User Interface</strong></td>
<td></td>
</tr>
<tr>
<td>ProCmdIconSet()</td>
<td>Designates an icon to be used with a Pro/TOOLKIT created command.</td>
</tr>
<tr>
<td>ProUITabSelectednamesGet()</td>
<td>Modifies the name of the currently selected (visible) tab panel.</td>
</tr>
<tr>
<td>ProUITabSelectednamesSet()</td>
<td></td>
</tr>
<tr>
<td>ProCmdDesignate()</td>
<td>Designates the command as available in the Screen Customization dialog of Pro/ENGINEER.</td>
</tr>
</tbody>
</table>

Summary of Technical Changes | A - 15
<table>
<thead>
<tr>
<th><strong>New Function</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>ProDirectoryChoose()</td>
<td>Prompts you to select a directory using the Pro/ENGINEER dialog box for browsing directories.</td>
</tr>
<tr>
<td>ProUIInputpanelDoubleGet()</td>
<td>Provide access to the value contained in the input panel, depending on the type of value, as determined by the InputType attribute.</td>
</tr>
<tr>
<td>ProUIInputpanelDoubleSet()</td>
<td></td>
</tr>
<tr>
<td>ProUIInputpanelIntegerGet()</td>
<td></td>
</tr>
<tr>
<td>ProUIInputpanelIntegerSet()</td>
<td></td>
</tr>
<tr>
<td>ProUIInputpanelStringGet()</td>
<td></td>
</tr>
<tr>
<td>ProUIInputpanelStringSet()</td>
<td></td>
</tr>
<tr>
<td>ProUIInputpanelWidestringGet()</td>
<td></td>
</tr>
<tr>
<td>ProUIInputpanelWidestringSet()</td>
<td></td>
</tr>
<tr>
<td>ProUIInputpanelOrdinalGet()</td>
<td></td>
</tr>
<tr>
<td>ProUIInputpanelOrdinalSet()</td>
<td></td>
</tr>
<tr>
<td>ProUIInputpanelNumeratorGet()</td>
<td></td>
</tr>
<tr>
<td>ProUIInputpanelNumeratorSet()</td>
<td></td>
</tr>
<tr>
<td>ProUIInputpanelDenominatorGet()</td>
<td></td>
</tr>
<tr>
<td>ProUIInputpanelDenominatorSet()</td>
<td></td>
</tr>
<tr>
<td>ProUIInputpanelMinintegerSet()</td>
<td>Control the limitations and formatting assigned for the input panel value.</td>
</tr>
<tr>
<td>ProUIInputpanelMaxintegerSet()</td>
<td></td>
</tr>
<tr>
<td>ProUIInputpanelMindoubleSet()</td>
<td></td>
</tr>
<tr>
<td>ProUIInputpanelMaxdoubleSet()</td>
<td></td>
</tr>
<tr>
<td>ProUIInputpanelDigitsSet()</td>
<td></td>
</tr>
<tr>
<td>New Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ProUIInputpanelInputActionSet()</td>
<td>Assign action functions for a variety of user actions in custom UI dialogs.</td>
</tr>
<tr>
<td>ProUIInputpanelFocusinActionSet()</td>
<td></td>
</tr>
<tr>
<td>ProUIInputpanelFocusoutActionSet()</td>
<td></td>
</tr>
<tr>
<td>ProUIListTriggerhighlightActionSet()</td>
<td></td>
</tr>
<tr>
<td>ProUIListFocusinActionSet()</td>
<td></td>
</tr>
<tr>
<td>ProUIListFocusoutActionSet()</td>
<td></td>
</tr>
<tr>
<td>ProUIMacromenuInputActionSet()</td>
<td></td>
</tr>
<tr>
<td>ProUIMacromenuTriggerhighlightActionSet()</td>
<td></td>
</tr>
<tr>
<td>ProUIMacromenuFocusinActionSet()</td>
<td></td>
</tr>
<tr>
<td>ProUIMacromenuFocusoutActionSet()</td>
<td></td>
</tr>
<tr>
<td>ProUITextareaActivateActionSet()</td>
<td></td>
</tr>
<tr>
<td>ProUITextareaFocusinActionSet()</td>
<td></td>
</tr>
<tr>
<td>ProUITextareaFocusoutActionSet()</td>
<td></td>
</tr>
<tr>
<td>ProUIMessageDialogDisplay()</td>
<td>Displays the UI message dialog.</td>
</tr>
<tr>
<td>ProUIPushButtonTextSet()</td>
<td>Sets the label for the push button.</td>
</tr>
<tr>
<td>ProUIMacroExecute()</td>
<td>Executes macros stored in Pro/ENGINEER and returns the control back to the calling Pro/TOOLKIT application.</td>
</tr>
<tr>
<td>ProUITabLabelsGet()</td>
<td>Modify the user-visible text for the tab panels.</td>
</tr>
<tr>
<td>ProUITabLabelsSet()</td>
<td></td>
</tr>
</tbody>
</table>

**Table Inquiry Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProUITableIsEnabled()</td>
<td>Table Inquiry Functions</td>
</tr>
<tr>
<td>ProUITableIsVisible()</td>
<td>Table Inquiry Functions</td>
</tr>
<tr>
<td>ProUITableIsAutohighlightEnabled()</td>
<td>Table Inquiry Functions</td>
</tr>
<tr>
<td>New Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>ProUITableColumnSet()</td>
<td></td>
</tr>
<tr>
<td>ProUITableVisibleSet()</td>
<td></td>
</tr>
<tr>
<td>ProUITableLockedSet()</td>
<td></td>
</tr>
<tr>
<td>ProUITableMinSet()</td>
<td></td>
</tr>
<tr>
<td>ProUITableHelpSet()</td>
<td></td>
</tr>
<tr>
<td>ProUITableTruncateSet()</td>
<td></td>
</tr>
<tr>
<td>ProUITableEnable()</td>
<td></td>
</tr>
<tr>
<td>ProUITableDisable()</td>
<td></td>
</tr>
<tr>
<td>ProUITableShow()</td>
<td></td>
</tr>
<tr>
<td>ProUITableHide()</td>
<td></td>
</tr>
<tr>
<td>ProUITableAutohighlight()</td>
<td></td>
</tr>
<tr>
<td>ProUITableComponentCopy()</td>
<td></td>
</tr>
<tr>
<td>ProUITableComponentDelete()</td>
<td></td>
</tr>
<tr>
<td>ProUITableRownamesGet()</td>
<td></td>
</tr>
<tr>
<td>ProUITableRownamesSet()</td>
<td></td>
</tr>
<tr>
<td>ProUITableRowlabelsGet()</td>
<td></td>
</tr>
<tr>
<td>ProUITableRowlabelsSet()</td>
<td></td>
</tr>
<tr>
<td>ProUITableRowRename()</td>
<td></td>
</tr>
<tr>
<td>ProUITableRowIndexGet()</td>
<td></td>
</tr>
<tr>
<td>ProUITableRowLabelGet()</td>
<td></td>
</tr>
<tr>
<td>ProUITableRowLabelSet()</td>
<td></td>
</tr>
<tr>
<td>ProUITableRowsInsert()</td>
<td></td>
</tr>
<tr>
<td>ProUITableRowsDelete()</td>
<td></td>
</tr>
<tr>
<td>ProUITableRowsSet()</td>
<td></td>
</tr>
</tbody>
</table>

Table Modifications Functions

Table Row Functions
<table>
<thead>
<tr>
<th><strong>New Function</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>ProUITableColumnnamesGet()</td>
<td>Table Column Functions</td>
</tr>
<tr>
<td>ProUITableColumnnamesSet()</td>
<td></td>
</tr>
<tr>
<td>ProUITableColumnlabelsGet()</td>
<td></td>
</tr>
<tr>
<td>ProUITableColumnlabelsSet()</td>
<td></td>
</tr>
<tr>
<td>ProUITableColumnresizingsSet()</td>
<td></td>
</tr>
<tr>
<td>ProUITableColumnwidthsSet()</td>
<td></td>
</tr>
<tr>
<td>ProUITableColumnRename()</td>
<td></td>
</tr>
<tr>
<td>ProUITableColumnIndexGet()</td>
<td></td>
</tr>
<tr>
<td>ProUITableColumnLabelGet()</td>
<td></td>
</tr>
<tr>
<td>ProUITableColumnLabelSet()</td>
<td></td>
</tr>
<tr>
<td>ProUITableColumnWidthGet()</td>
<td></td>
</tr>
<tr>
<td>ProUITableColumnWidthSet()</td>
<td></td>
</tr>
<tr>
<td>ProUITableColumnResizingFactorGet()</td>
<td></td>
</tr>
<tr>
<td>ProUITableColumnResizingFactorSet()</td>
<td></td>
</tr>
<tr>
<td>ProUITableColumnsInsert()</td>
<td></td>
</tr>
<tr>
<td>ProUITableColumnsDelete()</td>
<td></td>
</tr>
<tr>
<td>ProUITableResetColumnWidth()</td>
<td></td>
</tr>
<tr>
<td>ProUITableCellLabelGet()</td>
<td></td>
</tr>
<tr>
<td>ProUITableCellLabelSet()</td>
<td>Table Cell Functions</td>
</tr>
<tr>
<td>ProUITableIsCellSensitive()</td>
<td></td>
</tr>
<tr>
<td>ProUITableCellEnable()</td>
<td></td>
</tr>
<tr>
<td>ProUITableCellDisable()</td>
<td></td>
</tr>
<tr>
<td>ProUITableCellComponentCopy()</td>
<td></td>
</tr>
<tr>
<td>ProUITableCellComponentNameGet()</td>
<td></td>
</tr>
<tr>
<td>ProUITableCellComponentNameSet()</td>
<td></td>
</tr>
<tr>
<td>ProUITableCellComponentDelete()</td>
<td></td>
</tr>
</tbody>
</table>
The following table lists the functions that are superseded in Release Wildfire.

<table>
<thead>
<tr>
<th>Old Function</th>
<th>New Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProDrawingViewIsOverlay()</td>
<td>ProDrawingViewIsBackground()</td>
</tr>
<tr>
<td>ProDrawingViewOverlayviewGet()</td>
<td>ProSelectionViewGet()</td>
</tr>
<tr>
<td>ProConfigoptGet()</td>
<td>ProConfigoptArrayGet()</td>
</tr>
</tbody>
</table>
Miscellaneous Technical Changes

Release Wildfire includes the following technical changes:

Datum Planes

Two new optional elements have been added to the datum plane element tree:

- PRO_E_STD_FEATURE_NAME—the name of the datum plane.
- PRO_E_DTMPLN_FLIP_DIR—whether or not to flip the direction of the datum plane.

Field Datum Point

Pro/TOOLKIT in Pro/ENGINEER Wildfire supports feature creation, modification and access via the element tree for a field datum point. This element tree is documented in the header file ProDtmPnt.h.

Sketched Datum Point

Pro/TOOLKIT in Pro/ENGINEER wildfire supports feature creation, modification, and access via the element tree for a sketched datum point. This element tree is documented in the header file ProDtmPnt.h.

Rib Feature

Pro/TOOLKIT now supports the feature element tree for creating, modifying and accessing Rib features. The element tree is published in the header file ProRib.h.

Notification

The new notification type PRO_GLOBAL_INTERF_CALC_POST is called after a global interference check has been run on an assembly.
Platform Support

64-bit installations/32-bit applications

Pro/TOOLKIT applications compiled against Pro/ENGINEER Wildfire on 32-bit machines in spawn and asynchronous mode may be run with a comparable 64-bit installation of Pro/ENGINEER. DLL mode applications must be compiled on the same architecture as the Pro/ENGINEER installation.

Platform support: HPUX 11.0 support

Pro/ENGINEER Wildfire is built using native HPUX 11.0 compiler. Pro/TOOLKIT compilation in all modes (EXE, DLL, and asynchronous) is supported on HPUX 11.0.

Note: For the information about the supported compilers and operating system patches for Pro/TOOLKIT, see http://www.ptc.com/partners/hardware/current/toolkit.htm.

Drawings

The declaration for the function ProDrawingViewNameGet() has been moved to ProDrawing.h. Include ProDrawing.h in your application source code in order to locate this symbol.

The function ProDtlnotetextAlloc() will assign the value -1 for the text height, width, and thickness elements. A negative value instructs Pro/ENGINEER to use the defaults for the indicated drawing. Existing code which assigns a value to these parameters should not be affected.

The function ProDtlnotetextStringGet() no longer returns the symbolic braces {0: } surrounding the text. In addition, the special symbols (for example '}]' to represent a single closing brace in the text) are no longer provided. This behavior will also be reflected in the Pro/Develop function prodtl_get_item().

Simplified Representations

Symbolic

Pro/ENGINEER Wildfire features a new simplified representation type called symbolic. A new constant has been added to represent this type in Pro/TOOLKIT from ProSimprepTypeGet() and the ProSimprepdata structure.
Parts

Pro/TOOLKIT supports retrieval and identification of part simplified representations. The following functions support part mode:

- ProSimprepActiveGet()
- ProSimprepTypeGet()
- ProSimprepInit()
- ProSimprepDelete()
- ProSimprepSelect()
- ProSolidSimprepVisit()

Sketcher

The new structures Pro2dEllipsedef and Pro2dConicdef structures contain the necessary information to create and access conic and ellipse sketcher entities using Pro/TOOLKIT.

User interface

Pro/TOOLKIT in Pro/ENGINEER Wildfire offers functions to access and modify the sensitivity and visibility of user interface components, allowing greater flexibility in programming dialog interactions.

Data Exchange

Layouts, Diagrams and Reports

The functions Pro2dImportCreate(), Pro2dImportAppend() and Pro2dExport() now support the other 2D Pro/ENGINEER model formats.

Importing MEDUSA

The function Pro2dImportCreate() supports the MEDUSA format.

Import Feature

Several new formats supported by the function ProImportfeatCreate():

- CDRS
The function `ProInputFileRead()` now accepts a ProPath input argument, allowing you to specify a full input file name and path.

**Note:** Not all options accept a full path - check the function documentation for more information.

The function `ProOutputFileWrite()` accepts a ProFileName input argument, allowing you to specify a full output file name including the extension. The export of two-dimensional IGES and DXF supports a user specified file name instead of generating the output from the name of the drawing.

Some options for this function have been superseded by the function `ProIntf3DFileWrite()`.

The function `ProOutputFileWrite()` accepts a ProFileName input argument, allowing you to specify a full plot file name including extension.

### Parameters

Pro/ENGINEER Wildfire features restricted parameters (restricted by the user in the UI) and also locked parameters (restricted from changes by a PTC module). Pro/TOOLKIT functions attempting to modify these parameters will fail with an error return `PRO_TK_CANT_ACCESS`.

### ProSolidRegenerate

The function `ProSolidRegenerate()` accepts an integer bitmask with various options for regeneration. The function previously accepted a ProBoolean. The ProBoolean values `PRO_B_TRUE` and `PRO_B_FALSE` cause the function to behave as it did previously, but passing other integer values will invoke some of the new regeneration options. The new options include:

- An option to update instances in memory
- An option to force regeneration of all features
- Several regeneration failure options
UDF Placement

The function \texttt{ProUdfCreate()} supports placement of features using curve or edge vertices as references. The function \texttt{ProUdfreferenceAlloc()}, which takes a ProSelection, accepts a ProSelection representing an edge or curve vertex.

To acquire the edge or curve vertex interactively use \texttt{ProSelect()} with the selection filter "edge_end" or "curve_end".

Use \texttt{ProModelitemInit()} with the type PRO\_CURVE\_START, PRO\_CURVE\_END, PRO\_EDGE\_START, or PRO\_EDGE\_END to create a programmatic selection.

\textbf{Note:} Pro/ENGINEER does not allow you to use a curve vertex where an edge vertex is expected or an edge vertex where a curve vertex is expected.
This appendix describes how to use the Registry file to have a foreign program communicate with Pro/ENGINEER.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registry File Fields</td>
<td>B - 2</td>
</tr>
<tr>
<td>Sample Registry Files</td>
<td>B - 4</td>
</tr>
</tbody>
</table>
# Registry File Fields

The following table lists the fields in the registry file, `protk.dat`.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>name</strong></td>
<td>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 <code>ProSizeConst.h</code>). Currently the maximum size is 31 characters for the name, plus the end-of-string character.</td>
</tr>
<tr>
<td><strong>host</strong></td>
<td>Specifies the host name of the machine that the foreign program runs on. This is used only when <code>comm</code> is <code>rpc</code>.</td>
</tr>
</tbody>
</table>
| **startup** | Specifies the method Pro/ENGINEER should use to communicate with the Pro/TOOLKIT application. This field can take one of three values; spawn, daemon, or DLL.  
- **Spawn**— If the value is `spawn`, Pro/ENGINEER starts the foreign program.  
- **Daemon**—If the value is `daemon`, you must also enter the `rpc` number. You must start the foreign program before starting Pro/ENGINEER. The foreign program requires two arguments:  
  - The `rpc` program number assigned to `frnpgm1`.  
  - The full path to the text tree additions, as required by the foreign program.  
  
  This method (`startup` is `daemon`) currently uses the `rpc` communications method only.  
  You can start the foreign program using the following command:  
  ```bash  
  frnpgm1 <rpc_number> /home/protk  
  ```  
  **Note:** If the Pro/TOOLKIT application runs on a remote host, this field should be set to `daemon`. |
<p>| <strong>fail_tol</strong> | 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. |</p>
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exec_file</td>
<td>Specifies the name of the file produced by compiling and linking the Pro/TOOLKIT application. In DLL mode, this is a dynamically linkable library; in multiprocessing mode, it is the executable of a complete program.</td>
</tr>
<tr>
<td></td>
<td>If startup mode is spawn, this field specifies the path name to the foreign program, including the file name.</td>
</tr>
<tr>
<td></td>
<td>When your application is distributed, it is assumed that the executable file is in the Pro/TOOLKIT directory &lt;tk_loadpoint&gt;/machine/obj.</td>
</tr>
<tr>
<td>text_dir</td>
<td>Specifies the full path name to directory that contains the text directory. The text directory contains the message files menu files, resource files and UI bitmaps used by the application. These files allow multi-language support for menu buttons and messages used in the Pro/TOOLKIT application. Please refer to the chapter 'Menus' and 'Messages' for more information.</td>
</tr>
<tr>
<td></td>
<td>The text_dir does not need to include the directory text, it is added automatically by Pro/ENGINEER.</td>
</tr>
<tr>
<td></td>
<td>The search priority for messages and menu files is as follows:</td>
</tr>
<tr>
<td></td>
<td>1) Current working directory</td>
</tr>
<tr>
<td></td>
<td>2) text_dir/text</td>
</tr>
<tr>
<td></td>
<td>3) &lt;proe_loadpoint&gt;/text</td>
</tr>
<tr>
<td></td>
<td>The text_dir should be different from the Pro/ENGINEER text tree.</td>
</tr>
<tr>
<td>rpc_num</td>
<td>Specifies the rpc number Pro/ENGINEER uses to communicate with the foreign program. This line is required only if startup is daemon.</td>
</tr>
<tr>
<td></td>
<td>For testing use a decimal number in the following range:</td>
</tr>
<tr>
<td></td>
<td>536870912 - 1073741823</td>
</tr>
<tr>
<td></td>
<td>A number in this range should only be used for testing because it is not guaranteed to be unique or exclusive.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> You can get a unique registered rpc program number from Sun Microsystems, Incorporated®.</td>
</tr>
<tr>
<td>comm</td>
<td>Specifies the communication method. The valid types are pipe or rpc. This line may be omitted, in which case:</td>
</tr>
<tr>
<td></td>
<td>• If startup is spawn, the default value is pipe.</td>
</tr>
<tr>
<td></td>
<td>• If startup is daemon, the default value is rpc.</td>
</tr>
</tbody>
</table>
### 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]
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]

Example 3

In this example, the foreign program is spawned by Pro/ENGINEER. The program runs on the same machine as Pro/ENGINEER and communicates via the rpc mechanism. Note that in this example the rpc number is optional.

File: protk.dat
Example 4

In this example, the foreign program is started independently and prior to Pro/ENGINEER. The program and Pro/ENGINEER run on different machines and communicate via the rpc mechanism. Note that in this example the rpc number is required.

File: protk.dat

[Start of file on next line]

name      Product1
host      remote_name
startup   daemon
fail_tol  true
rpc_num   <rpc_number>  (required)
end

[End of file on previous line]

Example 5

In this example, the foreign program is started independently and prior to Pro/ENGINEER. The program and Pro/ENGINEER run on the same machines and communicate via the rpc mechanism. Note that in this example the rpc number is required.

File: protk.dat

[Start of file on next line]

name      Product1
host      hostname
startup   daemon
fail_tol  true
rpc_num   <rpc_number>  (required)
end

[End of file on previous line]
Debugging Pro/TOOLKIT Applications

This appendix describes debugging Pro/TOOLKIT applications.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debugging in Existing Process</td>
<td>C - 2</td>
</tr>
<tr>
<td>Debugging in a Separate Window</td>
<td>C - 3</td>
</tr>
</tbody>
</table>
Debugging in Existing Process

The following example shows how to attach the debugger to an existing process. In this example, *startup* and *comm* are assumed to be the default values (spawn and pipe, respectively).

Example 1: Debugging a Program that is Already Running

To Attach the Debugger to a Foreign Program that is Already Running

1. Start Pro/ENGINEER to initiate the foreign program.
   
   /bin/proe

2. Find the process identification number.

   ps -auxw | grep pt_install_test

3. Start the debugger in a different window.

   (dbx) debug pt_install_test [process id]

4. Set the break points.

   (dbx) stop in user_choice

5. Allow the foreign program to continue.

   (dbx) cont

When Pro/ENGINEER reaches the break point, control is handed over to the debugger.
The sample script is as follows:

```
ps -auxw | grep pt_install_test
dbx
(dbx) debug pt_install_test [process id]
Reading symbolic information...
Read 973 symbols
(dbx) stop in user_choice
(1) stop in user_choice
(dbx) cont
stopped in user_choice at line 24 in file "uchoice.c"
24 toggle = FALSE; /* Let this be the default. */
(dbx) cont
program exited with 0
(dbx) quit
```

### Debugging in a Separate Window

The following example shows how to invoke the debugger in a separate window.

#### Example 2: Invoking the Debugger as a Separate Process

A foreign program is normally invoked by Pro/ENGINEER because the default startup mode is spawn.

For debugging the foreign program, change the startup mode to daemon. This allows the foreign program to start before Pro/ENGINEER, in a separate window.

**How to Debug a Foreign Program**

1. Set the following fields in the `protk.dat` file:
   ```
   startup  daemon
   host     hostname
   comm     rpc
   rpc_num  <rpc_number>
   ```
2. Invoke the debugger `dbx`.
   ```
   dbx pt_install_test
   ```
3. Set the appropriate break points in the foreign program.
   ```
   (dbx) stop in user_initialize
   ```
4. Start the foreign program with the `rpc_num` and `text_dir`. 
(dbx) run <rpc_number> /home/user

5. Start Pro/ENGINEER as the user normally would.

/bin/proe

When Pro/ENGINEER makes the call to the foreign program, control is handed over to the debugger.

The sample script is as follows:

1 prompt % dbx pt_install_test
Reading symbolic information...
Read 14010 symbols
warning: main routine not compiled with the -g option
(dbx) stop in user_initialize
(2) stop in user_initialize
(dbx) run 1012345678 /host/user
Running: pt_install_test 1012345678 /home/user
stopped in user_initialize at line 74 in file "user_init.c"

74 menu_id = promenu_create ("MAIN","main.mnu");

(dbx) n
stopped in user_initialize at line 75 in file "user_init.c"

75 menu_id = promenu_expand ("MAIN","main.aux");

(dbx) n
(dbx) quit
2 prompt %

Example 3: Debugging Applications on NT/Windows 95 Systems

The following example shows how to debug an application on an NT or Windows 95 system.

%e How to Debug a Pro/TOOLKIT Application on NT and Windows 95 Systems

1. Add these changes to your makefile:
   
   CCFLAGS = ... /Od /Z7
   
   $LINK = .../debug:full.../debugtype:both

2. Recompile your application program.

3. Run Pro/ENGINEER (to start your Pro/TOOLKIT application program).

4. Use the task manager to get the process identifier (pid) of your application.
5. Run the Microsoft Visual C++ debugger using the following command:

    msdev -p <pid>

6. Use the File > Open command to see your source code.
7. Choose the commands Debug, Breakpoint, and so on.
8. Select your Pro/ENGINEER button, and the debugger stops at your breakpoint.
9. Debug your application as you would any other program.

**Example 4: Debugging Pro/TOOLKIT DLL Applications**

Debugging DLL applications is somewhat easier than debugging rpc applications because the application becomes a part of Pro/ENGINEER, rather than running separately. In particular, you can debug problems in your user_initialize() function, which is difficult or impossible to do in a standalone application.

When you debug DLL applications, set the environment variable PRO_TK_DBG_DLL_LOAD to display any operating system errors that occur while loading the DLL. The command is as follows:

    # setenv PRO_TK_DBG_DLL_LOAD 1

**Note:** PTC recommends that you rebuild all DLL applications for UNIX systems with “-hidden” added to LDFLAGS.

**How to Debug DLLs on UNIX Systems**

1. Find and copy “pro” (the startup script).
2. In the copy, replace the call to <path>/pro with the call to invoke your preferred debugger (dbx or a platform-specific GUI debugger) on pro.
3. Run the script.
4. When the debugger is up, set a breakpoint in prodev_init_one_app and start.
5. When the debugger hits the breakpoint, the system loads your DLL Pro/TOOLKIT application but does not initialize it. Remove this breakpoint and set breakpoints as desired in your code, and then continue.
This appendix describes how to update legacy applications using Pro/DEVELOP functions with current Pro/TOOLKIT functions.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Relationship Between Pro/TOOLKIT and Pro/DEVELOP</td>
<td>D - 2</td>
</tr>
<tr>
<td>Converting from Pro/DEVELOP</td>
<td>D - 2</td>
</tr>
<tr>
<td>Equivalent Pro/DEVELOP Functions</td>
<td>D - 16</td>
</tr>
<tr>
<td>Data Structures for Pro/DEVELOP-Style Functions</td>
<td>D - 30</td>
</tr>
</tbody>
</table>
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 Converting from Pro/DEVELOP.

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.

<table>
<thead>
<tr>
<th>Pro/TOOLKIT</th>
<th>Pro/DEVELOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>N/A</td>
</tr>
<tr>
<td>Model</td>
<td>Object</td>
</tr>
<tr>
<td>Solid (part or assembly)</td>
<td>Model</td>
</tr>
<tr>
<td>Surface</td>
<td>Face or surface</td>
</tr>
<tr>
<td>Component of an assembly</td>
<td>Member</td>
</tr>
<tr>
<td>Component path</td>
<td>Member identifier table (memb_id_tab)</td>
</tr>
<tr>
<td>External data</td>
<td>Generic application data</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Pro/DEVELOP Function</th>
<th>Pro/TOOLKIT Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>promenu_create()</td>
<td>ProMenuFileRegister()</td>
</tr>
<tr>
<td>promenu_expand()</td>
<td>ProMenuAuxfileRegister()</td>
</tr>
<tr>
<td>promenu_on_button()</td>
<td>ProMenubuttonActionSet()</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Pro/DEVELOP Function</th>
<th>Pro/TOOLKIT Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>prodb_get_feature_ids()</td>
<td>ProSolidFeatVisit()</td>
</tr>
<tr>
<td>prodb_first_part_face(),</td>
<td>ProSolidSurfaceVisit()</td>
</tr>
<tr>
<td>prodb_next_part_face()</td>
<td></td>
</tr>
</tbody>
</table>

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:

- prodb_first_member()
- prodb_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:

- prodb_first_datum()
- prodb_next_datum()
- prodb_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 ‘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 PRODEVDAT is now either PROTANDAT 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.

<table>
<thead>
<tr>
<th>Pro/DEVELOP Prohandle for the item type...</th>
<th>Can be cast directly to the Pro/TOOLKIT object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>ProMdl</td>
</tr>
<tr>
<td>Assembly</td>
<td>ProAssembly</td>
</tr>
<tr>
<td>Part</td>
<td>ProPart</td>
</tr>
<tr>
<td>Model (part or assembly)</td>
<td>ProSolid</td>
</tr>
<tr>
<td>Surface</td>
<td>ProSurface</td>
</tr>
<tr>
<td>Contour</td>
<td>ProContour</td>
</tr>
<tr>
<td>Edge</td>
<td>ProEdge</td>
</tr>
<tr>
<td>Curve</td>
<td>ProCurve</td>
</tr>
<tr>
<td>Datum quilt</td>
<td>ProQuilt</td>
</tr>
<tr>
<td>Point</td>
<td>ProPoint</td>
</tr>
<tr>
<td>Axis</td>
<td>ProAxis</td>
</tr>
<tr>
<td>Coordinate system</td>
<td>ProCsys</td>
</tr>
</tbody>
</table>

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.
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 \textit{not} equivalent to a Pro/DEVELOP `Prohandle`. The integer identifier still maps directly, however.

The following diagram applies to converting features.

Figure D-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 D-3: Pro/DEVELOP Part and Assembly Mapping

<table>
<thead>
<tr>
<th>Pro/DEVELOP Representation</th>
<th>Pro/TOOLKIT Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer ID</td>
<td>-   identical</td>
</tr>
<tr>
<td>pro_solid_ptr_from_id()</td>
<td>pro_solid_id_from_ptr()</td>
</tr>
<tr>
<td>Prohandle</td>
<td>ProSolid</td>
</tr>
<tr>
<td>prodb_get_object_info()</td>
<td>prodb_get_object_ptr()</td>
</tr>
<tr>
<td>Name and type</td>
<td>-   identical</td>
</tr>
<tr>
<td>pro_solid_id_from_ptr()</td>
<td>ProPostfixIdToSolid()</td>
</tr>
<tr>
<td>Name and type</td>
<td>pro_solid_id_from_ptr()</td>
</tr>
<tr>
<td>name and type</td>
<td>ProSolidToPostfixId()</td>
</tr>
<tr>
<td>ProMdlInit()</td>
<td>pro_solid_ptr_from_id()</td>
</tr>
<tr>
<td>ProMdlNameGet()</td>
<td>prodb_get_object_info()</td>
</tr>
<tr>
<td>ProMdlTypeGet()</td>
<td>prodb_get_object_ptr()</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Pro/DEVELOP</th>
<th>Pro/TOOLKIT</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Ptc_surf</code></td>
<td><code>ProSurfacedata</code></td>
</tr>
<tr>
<td><code>Ptc_curve</code></td>
<td><code>ProCurvedata</code></td>
</tr>
<tr>
<td><code>Pro_linestyle</code></td>
<td><code>ProLinestyle</code></td>
</tr>
<tr>
<td><code>pro_mode</code></td>
<td><code>ProMode</code></td>
</tr>
<tr>
<td><code>Pro_text_attributes</code></td>
<td><code>ProTextAttribute</code></td>
</tr>
<tr>
<td><code>int</code></td>
<td><code>ProMouseButton</code></td>
</tr>
<tr>
<td><code>int</code></td>
<td><code>ProDrawMode</code></td>
</tr>
<tr>
<td><code>int</code></td>
<td><code>ProColortype</code></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Select3d Field</th>
<th>Pro/TOOLKIT Read Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>sel_type</td>
<td>ProSelectionModelitemGet(), then read modelitem.type. See the note that follows this table.</td>
</tr>
<tr>
<td>selected_id</td>
<td>ProSelectionModelitemGet(), then read modelitem.id.</td>
</tr>
<tr>
<td>selected_ptr</td>
<td>ProSelectionModelitemGet(), then Pro*Init(), depending on the type.</td>
</tr>
<tr>
<td>select_pnt</td>
<td>ProSelectionPoint3dGet().</td>
</tr>
<tr>
<td>sel_param</td>
<td>ProSelectionUvParamGet().</td>
</tr>
<tr>
<td>sel_depth</td>
<td>ProSelectionDepthGet().</td>
</tr>
<tr>
<td>part_ptr</td>
<td>ProSelectionAsmcomppathGet(), and ProAsmcomppathMdlGet().</td>
</tr>
<tr>
<td>assembly_ptr</td>
<td>ProSelectionAsmcomppathGet(), then read comppath.owner.</td>
</tr>
<tr>
<td>memb_num</td>
<td>ProSelectionAsmcomppathGet(), then read comppath.table_num.</td>
</tr>
<tr>
<td>memb_id_tab</td>
<td>ProSelectionAsmcomppathGet(), then read comppath.comp_id_tab.</td>
</tr>
<tr>
<td>view_ptr</td>
<td>ProSelectionViewGet().</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th><code>pro_select()</code> option</th>
<th><code>Select3d sel_type</code></th>
<th><code>ProType</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>point</td>
<td>SEL_3D_PNT</td>
<td>PRO_POINT</td>
</tr>
<tr>
<td>axis</td>
<td>SEL_3D_AXIS</td>
<td>PRO_AXIS</td>
</tr>
<tr>
<td>datum</td>
<td>SEL_3D_SRF</td>
<td>PRO_SURFACE</td>
</tr>
<tr>
<td>csys</td>
<td>SEL_3D_CSYS</td>
<td>PRO_CSYS</td>
</tr>
<tr>
<td>feature</td>
<td>SEL_3D_FEAT</td>
<td>PRO_FEATURE</td>
</tr>
<tr>
<td>edge</td>
<td>SEL_3D_EDG</td>
<td>PRO_EDGE</td>
</tr>
<tr>
<td>edge_end</td>
<td>SEL_3D_VERT</td>
<td>PRO_EDGE_START or PRO_EDGE_END</td>
</tr>
<tr>
<td>curve</td>
<td>SEL_3D_CURVE</td>
<td>PRO_CURVE</td>
</tr>
<tr>
<td>curve_end</td>
<td>SEL_CURVE_END</td>
<td>PRO_CRV_START or PRO_CRV_END</td>
</tr>
<tr>
<td>sldedge</td>
<td>SEL_3D_EDGE</td>
<td>PRO_EDGE</td>
</tr>
<tr>
<td>qltedge</td>
<td>SEL_3D_EDGE</td>
<td>PRO_EDGE</td>
</tr>
<tr>
<td>surface</td>
<td>SEL_3D_SRF</td>
<td>PRO_SURFACE</td>
</tr>
<tr>
<td>sldface</td>
<td>SEL_3D_SRF</td>
<td>PRO_SURFACE</td>
</tr>
<tr>
<td>qltface</td>
<td>SEL_3D_SRF</td>
<td>PRO_SURFACE</td>
</tr>
<tr>
<td>dtmqlt</td>
<td>SEL_3D_SRF_LIST</td>
<td>PRO_QUILT</td>
</tr>
<tr>
<td>part</td>
<td>SEL_3D_PART</td>
<td>PRO_PART</td>
</tr>
<tr>
<td>part_or_asm</td>
<td>SEL_3D_PART</td>
<td>PRO_PART or PRO_ASSEMBLY</td>
</tr>
<tr>
<td>dimension</td>
<td>DTL_DIM</td>
<td>PRO_DIMENSION</td>
</tr>
<tr>
<td>ref_dim</td>
<td>DTL_REFDIM</td>
<td>PRO_REF_DIMENSION</td>
</tr>
<tr>
<td>pro_select() option</td>
<td>Select3d sel_type</td>
<td>ProType</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>gtol</td>
<td>DTL_GTOL</td>
<td>PRO_GTOL</td>
</tr>
<tr>
<td>dtl_symbol</td>
<td>DTL_SYMBOL</td>
<td>PRO_SYMBOL_INSTANCE</td>
</tr>
<tr>
<td>dwg_table</td>
<td>SEL_DWG_TABLE</td>
<td>PRO_DRAW_TABL</td>
</tr>
<tr>
<td>any_note</td>
<td>DTL_USER_NOTE</td>
<td>PRO_NOTE</td>
</tr>
<tr>
<td>note_3d</td>
<td>SEL_3D_NOTE</td>
<td>PRO_NOTE</td>
</tr>
<tr>
<td>dgm_obj</td>
<td>SEL_DGM_REF_OBJ</td>
<td>PRO_DIAGRAM_OBJECT</td>
</tr>
<tr>
<td>dgm_non_cable_wire</td>
<td>DTL_WIRE</td>
<td>PRO_DIAGRAM_WIRE</td>
</tr>
<tr>
<td>draft_ent</td>
<td>DTL_DRAFT_ENT</td>
<td>PRO_DRAFT_ENTITY</td>
</tr>
<tr>
<td>ext_obj</td>
<td>SEL_3D_EXT_OBJ</td>
<td>PRO_EXTOBJ</td>
</tr>
<tr>
<td>table_cell</td>
<td>SEL_TABLE_CELL</td>
<td>PRO_DRAW_TABLE_CELL</td>
</tr>
</tbody>
</table>

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_language
  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.

<table>
<thead>
<tr>
<th>Pro/DEVELOP Function</th>
<th>Equivalent Pro/TOOLKIT Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Core Functions</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Synchronous Mode</strong></td>
<td></td>
</tr>
<tr>
<td>pro_term()</td>
<td>ProEngineerEnd()</td>
</tr>
<tr>
<td><strong>Asynchronous Mode</strong></td>
<td></td>
</tr>
<tr>
<td>prodev_start_proengineer()</td>
<td>ProEngineerStart()</td>
</tr>
<tr>
<td>prodev_set_interrupt_func()</td>
<td>ProInterruptfuncSet()</td>
</tr>
<tr>
<td>prodev_handle_interrupt()</td>
<td>ProEventProcess()</td>
</tr>
<tr>
<td>prodev_set_proe_term_func()</td>
<td>ProTermFuncSet()</td>
</tr>
<tr>
<td>user_proe_term_func()</td>
<td>ProTerminationAction()</td>
</tr>
<tr>
<td>prodev_get_proe_status()</td>
<td>ProEngineerStatusGet()</td>
</tr>
<tr>
<td><strong>User-Supplied Main</strong></td>
<td></td>
</tr>
<tr>
<td>prodev_main()</td>
<td>ProToolkitMain()</td>
</tr>
<tr>
<td><strong>Menus</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Adding a Menu Button</strong></td>
<td></td>
</tr>
<tr>
<td>promenu_create()</td>
<td>ProMenuFileRegister()</td>
</tr>
<tr>
<td>promenu_expand()</td>
<td>ProMenuAuxfileRegister()</td>
</tr>
<tr>
<td>promenu_on_button()</td>
<td>ProMenubuttonActionSet()</td>
</tr>
<tr>
<td>promenu_load_action()</td>
<td>ProMenubuttonGenactionSet()</td>
</tr>
<tr>
<td><strong>New Menus</strong></td>
<td></td>
</tr>
<tr>
<td>promenu_action()</td>
<td>ProMenuProcess()</td>
</tr>
<tr>
<td>promenu_exit_up()</td>
<td>ProMenuDelete()</td>
</tr>
<tr>
<td>promenu_make()</td>
<td>ProMenuCreate()</td>
</tr>
<tr>
<td>promenu_no_exit()</td>
<td>ProMenuHold()</td>
</tr>
<tr>
<td>promenu_exit_action_up()</td>
<td>ProMenuDeleteWithStatus()</td>
</tr>
<tr>
<td>promenu_make_compound()</td>
<td>ProCompoundmenuCreate()</td>
</tr>
</tbody>
</table>

**Preempting Pro/ENGINEER Commands**
<table>
<thead>
<tr>
<th>Pro/DEVELOP Function</th>
<th>Equivalent Pro/TOOLKIT Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>promenu_load_pre_func()</td>
<td>ProMenubuttonPreactionSet()</td>
</tr>
<tr>
<td>promenu_load_post_func()</td>
<td>ProMenubuttonPostactionSet()</td>
</tr>
<tr>
<td><strong>Manipulating Menus</strong></td>
<td></td>
</tr>
<tr>
<td>promenu_set_item_location()</td>
<td>ProMenubuttonLocationSet()</td>
</tr>
<tr>
<td>promenu_set_item_visible()</td>
<td>PropMenubuttonVisibilitySet()</td>
</tr>
<tr>
<td>promenu_remove_item()</td>
<td>ProMenubuttonDelete()</td>
</tr>
<tr>
<td><strong>Data Menus</strong></td>
<td></td>
</tr>
<tr>
<td>promenu_set_mode()</td>
<td>ProMenuModeSet()</td>
</tr>
<tr>
<td>promenu_set_data_mode()</td>
<td>ProMenuDatamodeSet()</td>
</tr>
<tr>
<td><strong>Setting Menu Buttons</strong></td>
<td></td>
</tr>
<tr>
<td>promenu_set_item()</td>
<td>ProMenubuttonHighlight()</td>
</tr>
<tr>
<td>promenu_reset_item()</td>
<td>ProMenubuttonUnhighlight()</td>
</tr>
<tr>
<td><strong>Controlling Accessibility of Menu Buttons</strong></td>
<td></td>
</tr>
<tr>
<td>promenu_make_item_accessible()</td>
<td>ProMenubuttonActivate()</td>
</tr>
<tr>
<td>promenu_make_item_inaccessible()</td>
<td>ProMenubuttonDeactivate()</td>
</tr>
<tr>
<td><strong>Pushing and Popping Menus</strong></td>
<td></td>
</tr>
<tr>
<td>promenu_is_up()</td>
<td>ProMenuVisibilityGet()</td>
</tr>
<tr>
<td>promenu_push()</td>
<td>ProMenuPush()</td>
</tr>
<tr>
<td>promenu_pop()</td>
<td>ProMenuPop()</td>
</tr>
<tr>
<td><strong>Run-Time Menus</strong></td>
<td></td>
</tr>
<tr>
<td>proselect_strings()</td>
<td>ProMenuStringsSelect()</td>
</tr>
<tr>
<td><strong>Entering Pro/ENGINEER Commands</strong></td>
<td></td>
</tr>
<tr>
<td>proload_cmd_sequence()</td>
<td>ProMacroLoad()</td>
</tr>
<tr>
<td>promenu_push_command()</td>
<td>ProMenuCommandPush()</td>
</tr>
<tr>
<td><strong>Message Window</strong></td>
<td></td>
</tr>
<tr>
<td>promsg_print()</td>
<td>ProMessageDisplay()</td>
</tr>
<tr>
<td>promsg_clear()</td>
<td>ProMessageClear()</td>
</tr>
<tr>
<td><strong>Writing a Message to an Internal Buffer</strong></td>
<td></td>
</tr>
<tr>
<td>promsg_sprint()</td>
<td>ProMessageToBuffer()</td>
</tr>
<tr>
<td><strong>Getting Keyboard Input</strong></td>
<td></td>
</tr>
<tr>
<td>promsg_getint()</td>
<td>ProMessageIntegerRead()</td>
</tr>
<tr>
<td>Pro/DEVELOP Function</td>
<td>Equivalent Pro/TOOLKIT Function</td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>promsg_getdouble()</td>
<td>ProMessageDoubleRead()</td>
</tr>
<tr>
<td>promsg_getstring()</td>
<td>ProMessageStringRead()</td>
</tr>
<tr>
<td>promsg_getpasswd()</td>
<td>ProMessagePasswordRead()</td>
</tr>
</tbody>
</table>

### Graphics and Object Display

#### Manipulating Windows

<table>
<thead>
<tr>
<th>Function</th>
<th>Equivalent Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>pro_clear_window()</td>
<td>ProWindowClear()</td>
</tr>
<tr>
<td>pro_view_repaint()</td>
<td>ProWindowRepaint()</td>
</tr>
<tr>
<td>pro_refresh_window()</td>
<td>ProWindowRefresh()</td>
</tr>
<tr>
<td>pro_get_current_window()</td>
<td>ProWindowCurrentGet()</td>
</tr>
<tr>
<td>pro_set_current_window()</td>
<td>ProWindowCurrentSet()</td>
</tr>
<tr>
<td>pro_open_object_window()</td>
<td>ProObjectwindowCreate()</td>
</tr>
<tr>
<td>pro_close_object_window()</td>
<td>ProWindowDelete()</td>
</tr>
</tbody>
</table>

#### Model Orientation

<table>
<thead>
<tr>
<th>Function</th>
<th>Equivalent Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>pro_get_cur_window_matrix()</td>
<td>ProWindowCurrentMatrixGet()</td>
</tr>
<tr>
<td>pro_get_view_matrix()</td>
<td>ProViewMatrixGet()</td>
</tr>
<tr>
<td>pro_set_view_matrix()</td>
<td>ProViewMatrixSet()</td>
</tr>
<tr>
<td>pro_reset_view()</td>
<td>ProViewReset()</td>
</tr>
<tr>
<td>pro_rotate_view()</td>
<td>ProViewRotate()</td>
</tr>
<tr>
<td>pro_store_view()</td>
<td>ProViewStore()</td>
</tr>
<tr>
<td>pro_retrieve_view()</td>
<td>ProViewRetrieve()</td>
</tr>
<tr>
<td>pro_get_view_names()</td>
<td>ProViewNamesGet()</td>
</tr>
</tbody>
</table>

#### Displaying Pro/ENGINEER Objects

<table>
<thead>
<tr>
<th>Function</th>
<th>Equivalent Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>progr_display_object()</td>
<td>ProSolidDisplay() (for parts and assemblies) ProMdlDisplay()</td>
</tr>
<tr>
<td>pro_show_select()</td>
<td>ProSelectionHighlight()</td>
</tr>
<tr>
<td></td>
<td>ProSelectionDisplay()</td>
</tr>
<tr>
<td></td>
<td>ProSelectionUnhighlight()</td>
</tr>
</tbody>
</table>

#### Graphics Colors and Line Styles

<table>
<thead>
<tr>
<th>Function</th>
<th>Equivalent Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>progr_text_color()</td>
<td>ProTextColorSet()</td>
</tr>
<tr>
<td>progr_color()</td>
<td>ProGraphicsColorSet()</td>
</tr>
<tr>
<td>progr_get_color_map()</td>
<td>ProColormapGet()</td>
</tr>
<tr>
<td>progr_set_color_map()</td>
<td>ProColormapSet()</td>
</tr>
<tr>
<td><strong>Pro/DEVELOP Function</strong></td>
<td><strong>Equivalent Pro/TOOLKIT Function</strong></td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>-----------------------------------------------------</td>
</tr>
<tr>
<td>progr_set_line_style()</td>
<td>ProLinestyleSet()</td>
</tr>
<tr>
<td>progr_get_line_style_def()</td>
<td>ProLinestyleDataGet()</td>
</tr>
<tr>
<td><strong>Displaying Graphics</strong></td>
<td></td>
</tr>
<tr>
<td>progr_move_3d()</td>
<td>ProGraphicsPenPosition()</td>
</tr>
<tr>
<td>progr_draw_3d()</td>
<td>ProGraphicsLineDraw()</td>
</tr>
<tr>
<td>progr_put_polyline()</td>
<td>ProGraphicsPolylineDraw()</td>
</tr>
<tr>
<td>progr_put_multi_polylines()</td>
<td>ProGraphicsMultiPolylinesDraw()</td>
</tr>
<tr>
<td>progr_put_arc()</td>
<td>ProGraphicsArcDraw()</td>
</tr>
<tr>
<td>progr_put_circle()</td>
<td>ProGraphicsCircleDraw()</td>
</tr>
<tr>
<td>progr_draw_polygon_2d()</td>
<td>ProGraphicsPolygonDraw()</td>
</tr>
<tr>
<td><strong>Displaying Text</strong></td>
<td></td>
</tr>
<tr>
<td>progr_put_text()</td>
<td>ProGraphicsTextDisplay()</td>
</tr>
<tr>
<td>pro_get_text_attributes()</td>
<td>ProCurrentTextAttributesGet()</td>
</tr>
<tr>
<td>pro_set_text_attributes()</td>
<td>ProCurrentTextAttributesSet()</td>
</tr>
<tr>
<td>progr_get_default_font_id()</td>
<td>ProTextfontDefaultIdGet()</td>
</tr>
<tr>
<td>progr_get_font_id()</td>
<td>ProTextfontIdGet()</td>
</tr>
<tr>
<td>progr_get_font_name()</td>
<td>ProTextfontNameGet()</td>
</tr>
<tr>
<td><strong>Getting Mouse Input</strong></td>
<td></td>
</tr>
<tr>
<td>promenu_get_pick()</td>
<td>ProMousePickGet()</td>
</tr>
<tr>
<td>pro_sample_xy()</td>
<td>ProMouseTrack()</td>
</tr>
<tr>
<td>progr_set_draw_mode()</td>
<td>ProGraphicsModeSet()</td>
</tr>
<tr>
<td>pro_getbox()</td>
<td>ProMouseBoxInput()</td>
</tr>
<tr>
<td><strong>Display Lists</strong></td>
<td></td>
</tr>
<tr>
<td>pro_create_2d_disp_list()</td>
<td>ProDisplist2dCreate()</td>
</tr>
<tr>
<td>pro_display_2d_disp_list()</td>
<td>ProDisplist2dDisplay()</td>
</tr>
<tr>
<td>pro_delete_2d_disp_list()</td>
<td>ProDisplist2dDelete()</td>
</tr>
<tr>
<td>pro_create_3d_disp_list()</td>
<td>ProDisplist3dCreate()</td>
</tr>
<tr>
<td>pro_display_3d_disp_list()</td>
<td>ProDisplist3dDisplay()</td>
</tr>
<tr>
<td>pro_delete_3d_disp_list()</td>
<td>ProDisplist3dDelete()</td>
</tr>
<tr>
<td><strong>Layers</strong></td>
<td></td>
</tr>
<tr>
<td>prolayer_get_names()</td>
<td>ProMdlLayerNamesGet()</td>
</tr>
<tr>
<td>prolayer_add_item()</td>
<td>ProLayerItemAdd()</td>
</tr>
<tr>
<td>Pro/DEVELOP Function</td>
<td>Equivalent Pro/TOOLKIT Function</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>prolayer_remove_item()</td>
<td>ProLayerItemRemove()</td>
</tr>
<tr>
<td>prolayer_display()</td>
<td>ProLayerDisplaystatusSet()</td>
</tr>
<tr>
<td>prolayer_get_display()</td>
<td>ProLayerDisplaystatusGet()</td>
</tr>
<tr>
<td>prolayer_create_layer()</td>
<td>ProLayerCreate()</td>
</tr>
<tr>
<td>prolayer_delete_layer()</td>
<td>ProLayerDelete()</td>
</tr>
<tr>
<td>prolayer_get_items()</td>
<td>ProLayerItemsGet()</td>
</tr>
</tbody>
</table>

**Database Support**

### Session Objects

<table>
<thead>
<tr>
<th>Function</th>
<th>Pro/TOOLKIT Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>pro_get_current_object()</td>
<td>ProMdlCurrentGet()</td>
</tr>
<tr>
<td>pro_get_current_mode()</td>
<td>ProModeCurrentGet()</td>
</tr>
<tr>
<td>prodb_findDeclared_objects()</td>
<td>ProMdlDeclaredList()</td>
</tr>
<tr>
<td>prodb_findNoobject_depend()</td>
<td>ProMdlDependenciesList()</td>
</tr>
<tr>
<td>prodb_get_object_info()</td>
<td>ProMdlDataGet()</td>
</tr>
<tr>
<td>prodb_get_object_ptr()</td>
<td>ProMdlInit()</td>
</tr>
<tr>
<td>prodb_first_name_in_list()</td>
<td>ProSessionMdlList()</td>
</tr>
<tr>
<td>prodb_next_name_in_list()</td>
<td>ProSessionMdlList()</td>
</tr>
<tr>
<td>prodb_was_object_modified()</td>
<td>ProMdlModificationVerify()</td>
</tr>
<tr>
<td>pro_solid_id_from_ptr()</td>
<td>ProSolidToPostfixId()</td>
</tr>
<tr>
<td>pro_solid_ptr_from_id()</td>
<td>ProPostfixIdToSolid()</td>
</tr>
</tbody>
</table>

### File Management Operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Pro/TOOLKIT Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>prodb_create_object()</td>
<td>ProSolidCreate()</td>
</tr>
<tr>
<td>prodb_create_obj()</td>
<td>ProSolidCreate()</td>
</tr>
<tr>
<td>prodb_retrieve_object()</td>
<td>ProMdlRetrieve()</td>
</tr>
<tr>
<td>prodb_save_object()</td>
<td>ProMdlSave()</td>
</tr>
<tr>
<td>prodb_rename_object()</td>
<td>ProMdlRename()</td>
</tr>
<tr>
<td>prodb_copy_object()</td>
<td>ProMdlCopy()</td>
</tr>
<tr>
<td>prodb_erase_object()</td>
<td>ProMdlErase()</td>
</tr>
<tr>
<td>prodb_backup_object()</td>
<td>ProMdlBackup()</td>
</tr>
</tbody>
</table>

### Simplified Representations

<table>
<thead>
<tr>
<th>Function</th>
<th>Pro/TOOLKIT Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>prodb_get_simplfd_rep_info()</td>
<td>ProSimprepActiveGet()</td>
</tr>
<tr>
<td>prodb_get_simplfd_rep_list()</td>
<td>ProSolidSimprepVisit()</td>
</tr>
<tr>
<td>prodb_retrieve_simplfd_rep()</td>
<td>ProAsmSimprepRetrieve()</td>
</tr>
<tr>
<td>Pro/DEVELOP Function</td>
<td>Equivalent Pro/TOOLKIT Function</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td><strong>Selecting Objects</strong></td>
<td></td>
</tr>
<tr>
<td>pro_select()</td>
<td>ProSelect()</td>
</tr>
<tr>
<td>pro_get_selection()</td>
<td>ProSelect()</td>
</tr>
<tr>
<td>pro_set_and_get_selection()</td>
<td>ProSelect()</td>
</tr>
<tr>
<td><strong>Tracing a Ray Through a Model</strong></td>
<td></td>
</tr>
<tr>
<td>pro_ray_x_model()</td>
<td>ProSolidRayIntersectionCompute()</td>
</tr>
<tr>
<td><strong>Element Information</strong></td>
<td></td>
</tr>
<tr>
<td>pro_element_info()</td>
<td>PRO_IDENTIFY: Pro<em>IdGet(), Pro</em>Init()</td>
</tr>
<tr>
<td></td>
<td>PRO_BELONG_TO: ProGeomItemFeatureGet()</td>
</tr>
<tr>
<td><strong>Regenerating Models</strong></td>
<td></td>
</tr>
<tr>
<td>pro_regenerate()</td>
<td>ProSolidRegenerate()</td>
</tr>
<tr>
<td>pro_regenerate_object()</td>
<td>ProSolidRegenerate()</td>
</tr>
<tr>
<td><strong>Part Accuracy</strong></td>
<td></td>
</tr>
<tr>
<td>prodb_get_model_accuracy()</td>
<td>ProSolidAccuracyGet()</td>
</tr>
<tr>
<td>prodb_set_model_accuracy()</td>
<td>ProSolidAccuracySet()</td>
</tr>
<tr>
<td><strong>Mass Properties</strong></td>
<td></td>
</tr>
<tr>
<td>prodb_mass_prop()</td>
<td>ProSolidMassPropertyGet()</td>
</tr>
<tr>
<td><strong>Utilities</strong></td>
<td></td>
</tr>
<tr>
<td>pro_str_to_wstr()</td>
<td>ProStringToWstring()</td>
</tr>
<tr>
<td>pro_wstr_to_str()</td>
<td>ProWstringToString()</td>
</tr>
<tr>
<td>pro_wchar_t_check()</td>
<td>ProWcharSizeVerify()</td>
</tr>
<tr>
<td>pro_show_file()</td>
<td>ProInfoWindowDisplay()</td>
</tr>
<tr>
<td>pro_show_info_window()</td>
<td>ProInfoWindowDisplay()</td>
</tr>
<tr>
<td>prodb_edit_file()</td>
<td>ProFileEdit()</td>
</tr>
<tr>
<td><strong>Session Tools</strong></td>
<td></td>
</tr>
<tr>
<td>pro_get_prodevdat_info()</td>
<td>ProToolkitApplExecPathGet()</td>
</tr>
<tr>
<td></td>
<td>ProToolkitApplTextPathGet()</td>
</tr>
<tr>
<td>pro_getenvironment()</td>
<td>ProConfigoptGet()</td>
</tr>
<tr>
<td>pro_change_dir()</td>
<td>ProDirectoryChange()</td>
</tr>
<tr>
<td>pro_get_current_directory()</td>
<td>ProDirectoryCurrentGet()</td>
</tr>
<tr>
<td>pro_is_option_ordered()</td>
<td>ProOptionOrderedVerify()</td>
</tr>
<tr>
<td>Pro/DEVELOP Function</td>
<td>Equivalent Pro/TOOLKIT Function</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>pro_get_config()</td>
<td>ProConfigoptGet()</td>
</tr>
<tr>
<td>pro_set_config()</td>
<td>ProConfigoptSet()</td>
</tr>
<tr>
<td>progr_invalidate_display_list()</td>
<td>ProDisplstInvalidate()</td>
</tr>
<tr>
<td><strong>Exporting and Importing Files</strong></td>
<td></td>
</tr>
<tr>
<td>pro_export_file_from_pro()</td>
<td>ProOutputFileWrite()</td>
</tr>
<tr>
<td>pro_export_plot_file()</td>
<td>ProPlotfileWrite()</td>
</tr>
<tr>
<td>pro_read_file_to_pro()</td>
<td>ProInputFileRead()</td>
</tr>
<tr>
<td><strong>Material Names</strong></td>
<td></td>
</tr>
<tr>
<td>prodb_get_material_props()</td>
<td>ProPartMaterialdataGet()</td>
</tr>
<tr>
<td>prodb_get_material_name()</td>
<td>ProPartMaterialNameGet()</td>
</tr>
<tr>
<td>prodb_set_material_name()</td>
<td>ProPartMaterialSet()</td>
</tr>
<tr>
<td><strong>Storing Generic Data</strong></td>
<td></td>
</tr>
<tr>
<td>proappdata_register_class()</td>
<td>ProExtdataClassRegister()</td>
</tr>
<tr>
<td>proappdata_create_data()</td>
<td>ProExtdataSlotCreate()</td>
</tr>
<tr>
<td>proappdata_write_data()</td>
<td>ProExtdataSlotWrite()</td>
</tr>
<tr>
<td>proappdata_read_data()</td>
<td>ProExtdataSlotRead()</td>
</tr>
<tr>
<td>proappdata_delete_data()</td>
<td>ProExtdataSlotDelete()</td>
</tr>
<tr>
<td>proappdata_list_classes()</td>
<td>ProExtdataClassNamesList()</td>
</tr>
<tr>
<td>proappdata_list_data_in_class()</td>
<td>ProExtdataSlotIdsList()</td>
</tr>
<tr>
<td><strong>Geometry</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Traversing the Geometry of a Part</strong></td>
<td></td>
</tr>
<tr>
<td>prodb_first_part_face()</td>
<td>ProSolidSurfaceVisit()</td>
</tr>
<tr>
<td>prodb_next_part_face()</td>
<td>ProSolidSurfaceVisit()</td>
</tr>
<tr>
<td>prodb_getsolid_surfaces()</td>
<td>ProSolidSurfaceVisit()</td>
</tr>
<tr>
<td>prodb_first_face_contour()</td>
<td>ProSurfaceContourVisit()</td>
</tr>
<tr>
<td>prodb_next_face_contour()</td>
<td>ProSolidSurfaceVisit()</td>
</tr>
<tr>
<td>prodb_first_cntr_edge()</td>
<td>ProContourEdgeVisit()</td>
</tr>
<tr>
<td>prodb_next_cntr_edge()</td>
<td>ProContourEdgeVisit()</td>
</tr>
<tr>
<td>prodb_get_datum_curves()</td>
<td>ProSolidFeatVisit(), ProFeatureGeomitemVisit()</td>
</tr>
<tr>
<td>prodb_get_datum_surfaces()</td>
<td>ProSolidQuiltVisit(), ProQuiltSurfaceVisit()</td>
</tr>
<tr>
<td>Pro/DEVELOP Function</td>
<td>Equivalent Pro/TOOLKIT Function</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>prodb_edge_data()</td>
<td>ProEdgeNeighborsGet()</td>
</tr>
<tr>
<td>prodb_edge_direction()</td>
<td>ProEdgeDirGet()</td>
</tr>
<tr>
<td>prodb_contour_traversal()</td>
<td>ProContourTraversalGet()</td>
</tr>
<tr>
<td>prodb_containing_contour()</td>
<td>ProContainingContourFind()</td>
</tr>
<tr>
<td>prodb_vertex_data()</td>
<td>ProEdgeVertexdataGet()</td>
</tr>
<tr>
<td>prodb_get_solid_volumes()</td>
<td>ProSldsurfaceVolumesFind()</td>
</tr>
<tr>
<td></td>
<td><strong>Evaluating Geometry</strong></td>
</tr>
<tr>
<td>prodb_edge_tessellation()</td>
<td>ProEdgeTessellationGet()</td>
</tr>
<tr>
<td>pro_eval_xyz_edge()</td>
<td>ProEdgeXyzdataEval()</td>
</tr>
<tr>
<td>pro_eval_xyz_entity()</td>
<td>ProCurveXyzdataEval()</td>
</tr>
<tr>
<td>pro_eval_uv_edge()</td>
<td>ProEdgeUvdataEval()</td>
</tr>
<tr>
<td>pro_eval_xyz_face()</td>
<td>ProSurfaceXyzdataEval()</td>
</tr>
<tr>
<td>pro_get_edge_param()</td>
<td>ProEdgeParamEval()</td>
</tr>
<tr>
<td>pro_get_entity_param()</td>
<td>ProCurveParamEval()</td>
</tr>
<tr>
<td>pro_get_face_params()</td>
<td>ProSurfaceParamEval()</td>
</tr>
<tr>
<td>prodb_uv_in_face_domain()</td>
<td>ProSurfaceUvpntVerify()</td>
</tr>
<tr>
<td>pro_point_on_geom()</td>
<td>ProGeometryAtPointFind()</td>
</tr>
<tr>
<td>prodb_get_edge_uv_points()</td>
<td>ProEdgeTessellationGet()</td>
</tr>
<tr>
<td></td>
<td><strong>Geometry Equations</strong></td>
</tr>
<tr>
<td>prodb_get_curve_type()</td>
<td>ProCurveTypeGet()</td>
</tr>
<tr>
<td>pro_get_curve_type_geom()</td>
<td>ProCurvedataGet()</td>
</tr>
<tr>
<td></td>
<td>ProAxisDataGet()</td>
</tr>
<tr>
<td>prodb_get_edge_type()</td>
<td>ProEdgeTypeGet()</td>
</tr>
<tr>
<td>prodb_get_edge_curve()</td>
<td>ProEdgedataGet()</td>
</tr>
<tr>
<td>prodb_rls_edge_curve()</td>
<td>ProGeomitemdataFree()</td>
</tr>
<tr>
<td>prodb_get_face_type()</td>
<td>ProSurfaceTypeGet()</td>
</tr>
<tr>
<td>prodb_get_surface()</td>
<td>ProSurfacedataGet()</td>
</tr>
<tr>
<td>prodb_rls_surface()</td>
<td>ProGeomitemdataFree()</td>
</tr>
<tr>
<td>prodb_surface_to_nurbs()</td>
<td>ProSurfaceToNURBS()</td>
</tr>
<tr>
<td>prodb_edge_to_nurbs()</td>
<td>ProEdgeToNURBS()</td>
</tr>
<tr>
<td>prodb_entity_to_nurbs()</td>
<td>ProCurveToNURBS()</td>
</tr>
<tr>
<td>Pro/DEVELOP Function</td>
<td>Equivalent Pro/TOOLKIT Function</td>
</tr>
<tr>
<td>------------------------------</td>
<td>---------------------------------------------------------------------</td>
</tr>
<tr>
<td>pro_edge_length()</td>
<td>ProEdgeLengthEval()</td>
</tr>
<tr>
<td>pro_face_area()</td>
<td>ProSurfaceAreaEval()</td>
</tr>
<tr>
<td>prodb_measure()</td>
<td>ProSurfaceDiameterEval(),</td>
</tr>
<tr>
<td></td>
<td>ProGeomitemAngleEval(),</td>
</tr>
<tr>
<td></td>
<td>ProGeomitemDistanceEval()</td>
</tr>
<tr>
<td>proface_extremes()</td>
<td>ProSurfaceExtremesEval()</td>
</tr>
<tr>
<td>prodb_get_envelope()</td>
<td>ProSolidOutlineGet()</td>
</tr>
<tr>
<td>prodb_compute_outline()</td>
<td>ProSolidOutlineCompute()</td>
</tr>
<tr>
<td>Parameters and Dimensions</td>
<td></td>
</tr>
<tr>
<td>Parameters</td>
<td></td>
</tr>
<tr>
<td>prodb_get_parameters()</td>
<td>ProParameterVisit() and ProParameterValueGet()</td>
</tr>
<tr>
<td>prodb_set_parameters()</td>
<td>ProParameterValueSet()</td>
</tr>
<tr>
<td>prodb_add_parameters()</td>
<td>ProParameterCreate()</td>
</tr>
<tr>
<td>prodb_delete_parameters()</td>
<td>ProParameterDelete()</td>
</tr>
<tr>
<td>prodb_reset_parameters()</td>
<td>ProParameterValueReset()</td>
</tr>
<tr>
<td>prodb_designate_param()</td>
<td>ProParameterDesignationAdd()</td>
</tr>
<tr>
<td></td>
<td>ProParameterDesignationRemove()</td>
</tr>
<tr>
<td></td>
<td>ProParameterDesignationVerify()</td>
</tr>
<tr>
<td>Features</td>
<td></td>
</tr>
<tr>
<td>Listing Features</td>
<td></td>
</tr>
<tr>
<td>prodb_get_feature_ids()</td>
<td>ProSolidFeatVisit()</td>
</tr>
<tr>
<td>prodb_get_feat_parent_child()</td>
<td>ProFeatureParentsGet()</td>
</tr>
<tr>
<td></td>
<td>ProFeatureChildrenGet()</td>
</tr>
<tr>
<td>prodb_get_feat_type()</td>
<td>ProFeatureTypeGet()</td>
</tr>
<tr>
<td>Names of Features and Other Elements</td>
<td></td>
</tr>
<tr>
<td>prodb_get_element_name()</td>
<td>ProModelitemNameGet()</td>
</tr>
<tr>
<td>prodb_set_element_name()</td>
<td>ProModelitemNameSet()</td>
</tr>
<tr>
<td>Feature Geometry</td>
<td></td>
</tr>
<tr>
<td>prodb_get_feature_surfaces()</td>
<td>ProFeatureGeomitemVisit()</td>
</tr>
<tr>
<td>prodb_get_surface_feature()</td>
<td>ProGeomitemFeatureGet()</td>
</tr>
<tr>
<td>Manipulating Features</td>
<td></td>
</tr>
<tr>
<td>prodbSuppress_feature()</td>
<td>ProFeatureSuppress()</td>
</tr>
<tr>
<td></td>
<td>ProSolidFeatstatusSet()</td>
</tr>
<tr>
<td>Pro/DEVELOP Function</td>
<td>Equivalent Pro/TOOLKIT Function</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------------------------------------------------------</td>
</tr>
<tr>
<td>prodb_resume_feature()</td>
<td>ProFeatureResume()</td>
</tr>
<tr>
<td></td>
<td>ProSolidFeatstatusSet()</td>
</tr>
<tr>
<td>prodb_delete_feature()</td>
<td>ProFeatureDelete()</td>
</tr>
<tr>
<td><strong>Datum Planes</strong></td>
<td></td>
</tr>
<tr>
<td>prodb_first_datum()</td>
<td>ProSolidFeatVisit()</td>
</tr>
<tr>
<td></td>
<td>ProFeatureSurfaceVisit()</td>
</tr>
<tr>
<td>prodb_next_datum()</td>
<td>ProSolidFeatVisit()</td>
</tr>
<tr>
<td></td>
<td>ProFeatureSurfaceVisit()</td>
</tr>
<tr>
<td><strong>Datum Points</strong></td>
<td></td>
</tr>
<tr>
<td>prodb_get_feature_dtm_points()</td>
<td>ProSolidFeatVisit()</td>
</tr>
<tr>
<td></td>
<td>ProFeatureGeomitemVisit()</td>
</tr>
<tr>
<td>prodb_eval_xyz_dtm_point()</td>
<td>ProPointCoordGet()</td>
</tr>
<tr>
<td><strong>Axes</strong></td>
<td></td>
</tr>
<tr>
<td>prodb_get_first_axis()</td>
<td>ProSolidAxisVisit()</td>
</tr>
<tr>
<td></td>
<td>ProGeomitemFeatureGet()</td>
</tr>
<tr>
<td>prodb_get_next_axis()</td>
<td>ProSolidAxisVisit()</td>
</tr>
<tr>
<td></td>
<td>ProGeomitemFeatureGet()</td>
</tr>
<tr>
<td><strong>Coordinate Systems</strong></td>
<td></td>
</tr>
<tr>
<td>prodb_find_csys()</td>
<td>ProSolidCsystVisit()</td>
</tr>
<tr>
<td></td>
<td>ProGeomitemFeatureGet()</td>
</tr>
<tr>
<td>prodb_unpack_csys()</td>
<td>ProCsystdataGet()</td>
</tr>
<tr>
<td><strong>Surface Quilts</strong></td>
<td></td>
</tr>
<tr>
<td>prodb_get_surface_quilt()</td>
<td>ProSurfaceQuiltGet()</td>
</tr>
<tr>
<td><strong>Assemblies</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Finding Assembly Members</strong></td>
<td></td>
</tr>
<tr>
<td>prodb_first_member()</td>
<td>ProSolidFeatVisit()</td>
</tr>
<tr>
<td></td>
<td>ProFeatureTypeGet()</td>
</tr>
<tr>
<td>prodb_next_member()</td>
<td>ProSolidFeatVisit()</td>
</tr>
<tr>
<td></td>
<td>ProFeatureTypeGet()</td>
</tr>
<tr>
<td>prodb_member_to_object()</td>
<td>ProAsmcompMdlGet()</td>
</tr>
<tr>
<td>prodb_obj_from_assem_rel()</td>
<td>ProPostfixIdToSolid()</td>
</tr>
<tr>
<td><strong>Location of Assembly Members</strong></td>
<td></td>
</tr>
<tr>
<td>prodb_member_transform()</td>
<td>ProAsmcomppathTrfGet()</td>
</tr>
<tr>
<td>pro_vectors_to_transf()</td>
<td>ProMatrixInit()</td>
</tr>
<tr>
<td>Pro/DEVELOP Function</td>
<td>Equivalent Pro/TOOLKIT Function</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>--------------------------------------------------------</td>
</tr>
<tr>
<td>prodb_get_asm_transform()</td>
<td>ProAsmcomppathTrfGet()</td>
</tr>
<tr>
<td>prodb_set_member_transform()</td>
<td>ProAsmcomppathTrfSet()</td>
</tr>
<tr>
<td><strong>Assembling and Deleting Members</strong></td>
<td></td>
</tr>
<tr>
<td>prodb_assemble_by_transform()</td>
<td>ProFeatureCreate()</td>
</tr>
<tr>
<td>prodb_assemble_component()</td>
<td>ProFeatureCreate()</td>
</tr>
<tr>
<td>prodb_explode_assembly()</td>
<td>ProAssemblyExplode()</td>
</tr>
<tr>
<td></td>
<td>ProAssemblyUnexplode()</td>
</tr>
<tr>
<td>prodb_get_asm_constraints()</td>
<td>ProFeatureElemtreeCreate()</td>
</tr>
<tr>
<td>prodb_is_asm_exploded()</td>
<td>ProAssemblyIsExploded()</td>
</tr>
<tr>
<td><strong>Notify</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Basic Notify Functions</strong></td>
<td></td>
</tr>
<tr>
<td>prodev_notify()</td>
<td>ProNotificationSet()</td>
</tr>
<tr>
<td></td>
<td>ProNotificationUnset()</td>
</tr>
<tr>
<td>pro_on_regenerate_end()</td>
<td>ProNotificationSet()</td>
</tr>
<tr>
<td></td>
<td>ProNotificationUnset()</td>
</tr>
<tr>
<td><strong>Notify for File Management Operations</strong></td>
<td></td>
</tr>
<tr>
<td>user_menu_dbms_save_pre()</td>
<td>ProMdlSavePreAction</td>
</tr>
<tr>
<td>user_menu_dbms_save_post()</td>
<td>ProMdlSavePostAction</td>
</tr>
<tr>
<td>user_menu_dbms_copy_pre()</td>
<td>ProMdlCopyPreAction</td>
</tr>
<tr>
<td>user_menu_dbms_copy_post()</td>
<td>ProMdlCopyPostAction</td>
</tr>
<tr>
<td>user_menu_dbms_rename_pre()</td>
<td>ProMdlRenamePreAction</td>
</tr>
<tr>
<td>user_menu_dbms_rename_post()</td>
<td>ProMdlRenamePostAction</td>
</tr>
<tr>
<td>user_menu_dbms_erase_pre()</td>
<td>ProMdlErasePreAction</td>
</tr>
<tr>
<td>user_menu_dbms_erase_post()</td>
<td>ProMdlErasePostAction</td>
</tr>
<tr>
<td>user_menu_dbms_purge_pre()</td>
<td>ProMdlPurgePreAction</td>
</tr>
<tr>
<td>user_menu_dbms_purge_post()</td>
<td>ProMdlPurgePostAction</td>
</tr>
<tr>
<td>user_menu_dbms_delete_pre()</td>
<td>ProMdlDeletePreAction</td>
</tr>
<tr>
<td>user_menu_dbms_delete_post()</td>
<td>ProMdlDeletePostAction</td>
</tr>
<tr>
<td>user_menu_dbms_create_pre()</td>
<td>ProMdlCreatePreAction</td>
</tr>
<tr>
<td>user_menu_dbms_create_post()</td>
<td>ProMdlCreatePostAction</td>
</tr>
<tr>
<td>user_menu_dbms_retrieve_pre()</td>
<td>ProMdlRetrievePreAction</td>
</tr>
<tr>
<td>user_menu_dbms_retrieve_post()</td>
<td>ProMdlRetrievePostAction</td>
</tr>
<tr>
<td>user_dbms_save_post_all()</td>
<td>ProMdlSavePostAllAction</td>
</tr>
<tr>
<td>Pro/DEVELOP Function</td>
<td>Equivalent Pro/TOOLKIT Function</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>------------------------------------------------------</td>
</tr>
<tr>
<td>user_dbms_copy_post_all()</td>
<td>ProMdlCopyPostAllAction</td>
</tr>
<tr>
<td>user_dbms_erase_post_all()</td>
<td>ProMdlErasePostAllAction</td>
</tr>
<tr>
<td>user_dbms_delete_post_all()</td>
<td>ProMdlDeletePostAllAction</td>
</tr>
<tr>
<td>user_dbms_retrieve_post_all()</td>
<td>ProMdlRetrievePostAllAction</td>
</tr>
<tr>
<td><strong>Failure Notify</strong></td>
<td></td>
</tr>
<tr>
<td>user_dbms_failure_function()</td>
<td>ProMdlDbmsFailureAction</td>
</tr>
<tr>
<td><strong>Change Notify</strong></td>
<td></td>
</tr>
<tr>
<td>user_change_window_post()</td>
<td>ProWindowChangePostAction</td>
</tr>
<tr>
<td>user_change_directory()</td>
<td>ProDirectoryChangePostAction</td>
</tr>
<tr>
<td><strong>Graphics Notify</strong></td>
<td></td>
</tr>
<tr>
<td>user_graphics_object_output_pre()</td>
<td>ProMdlDisplayPreAction</td>
</tr>
<tr>
<td>user_graphics_object_output_post()</td>
<td>ProMdlDisplayPostAction</td>
</tr>
<tr>
<td><strong>Manufacturing Notify</strong></td>
<td></td>
</tr>
<tr>
<td>user_mfg_oper_cl_file_post()</td>
<td>ProMfgoperClPostAction</td>
</tr>
<tr>
<td>user_mfg_feat_cl_file_post()</td>
<td>ProNcseqClPostAction</td>
</tr>
<tr>
<td><strong>Manufacturing Operations</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Manufacturing Components</strong></td>
<td></td>
</tr>
<tr>
<td>prodb_get_comp_role()</td>
<td>ProMfgAssemGet(), ProMfgSolidGet(), ProMfgFeatureOwnerGet()</td>
</tr>
<tr>
<td><strong>Manufacturing Parameters</strong></td>
<td></td>
</tr>
<tr>
<td>promfg_get_tool_ids()</td>
<td>ProMfgToolVisit()</td>
</tr>
<tr>
<td>promfg_get_tool_parameters()</td>
<td>ProToolParamGet()</td>
</tr>
<tr>
<td>promfg_set_tool_parameters()</td>
<td>ProToolElemParamAdd()</td>
</tr>
<tr>
<td>promfg_get_nc_type()</td>
<td>ProNcseqTypeGet()</td>
</tr>
<tr>
<td><strong>Cabling Operations</strong></td>
<td></td>
</tr>
<tr>
<td>procbi_create_cables_from_logical()</td>
<td>ProCablesFromLogicalCreate()</td>
</tr>
<tr>
<td>procbi_get_cables_from_logical()</td>
<td>ProCablesFromLogicalGet()</td>
</tr>
<tr>
<td>procbi_get_connectors_from_logical()</td>
<td>ProConnectorsFromLogicalGet()</td>
</tr>
<tr>
<td>procbi_get_assy_spools()</td>
<td>ProAssemblySpoolsCollect()</td>
</tr>
<tr>
<td>procbi_create_spool()</td>
<td>ProSpoolCreate()</td>
</tr>
<tr>
<td>procbi_get_spool_params()</td>
<td>ProSpoolParametersCollect()</td>
</tr>
<tr>
<td><strong>Pro/DEVELOP Function</strong></td>
<td><strong>Equivalent Pro/TOOLKIT Function</strong></td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>procbl_get_spool_param()</td>
<td>ProSpoolParameterGet()</td>
</tr>
<tr>
<td>procbl_get_assy_connectors()</td>
<td>ProAssemblyConnectorsGet()</td>
</tr>
<tr>
<td>procbl_designate_connector()</td>
<td>ProConnectorDesignate()</td>
</tr>
<tr>
<td>procbl_unde designate_connector()</td>
<td>ProConnectorUndesignate()</td>
</tr>
<tr>
<td>procbl_get_connector_entry_ports()</td>
<td>ProConnectorEntryPortsGet()</td>
</tr>
<tr>
<td>procbl_set_spool_params()</td>
<td>ProSpoolParametersSet()</td>
</tr>
<tr>
<td>procbl_delete_spool_param()</td>
<td>ProSpoolParameterDelete()</td>
</tr>
<tr>
<td>procbl_get_spools_from_ logical()</td>
<td>ProSpoolsFromLogicalGet()</td>
</tr>
<tr>
<td>procbl_get_spools_from_ logical()</td>
<td>ProSpoolsFromLogicalGet()</td>
</tr>
<tr>
<td>procbl_get_connector()</td>
<td>ProConnectorParamGet()</td>
</tr>
<tr>
<td>procbl_delete_connector_param()</td>
<td>ProConnectorParamDelete()</td>
</tr>
<tr>
<td>procbl_get_connector_params()</td>
<td>ProConnectorParamsCollect()</td>
</tr>
<tr>
<td>procbl_set_connector_params()</td>
<td>ProConnectorParamsSet()</td>
</tr>
<tr>
<td>procbl_get_assy_harnesses()</td>
<td>ProAssemblyHarnessesCollect()</td>
</tr>
<tr>
<td>procbl_create_harness()</td>
<td>ProHarnessCreate()</td>
</tr>
<tr>
<td>procbl_get_harness_cables()</td>
<td>ProHarnessCablesCollect()</td>
</tr>
<tr>
<td>procbl_get_harness_id()</td>
<td>ProHarnessLocationsCollect()</td>
</tr>
<tr>
<td>procbl_get_wire_harnesses()</td>
<td>ProHarnessHarnessesGet()</td>
</tr>
<tr>
<td>procbl_get_wire_length()</td>
<td>ProCableLengthGet()</td>
</tr>
<tr>
<td>procbl_get_cable_id()</td>
<td>ProCableByParamGet()</td>
</tr>
<tr>
<td>procbl_get_cable_name()</td>
<td>ProCableNameGet()</td>
</tr>
<tr>
<td>procbl_get_cable_subtype()</td>
<td>ProCableTypeGet()</td>
</tr>
<tr>
<td>procbl_create_cable()</td>
<td>ProCableCreate()</td>
</tr>
<tr>
<td>procbl_create_bundle()</td>
<td>ProBundleCreate()</td>
</tr>
<tr>
<td>procbl_get_bundle_cables()</td>
<td>ProBundleCablesCollect()</td>
</tr>
<tr>
<td>procbl_get_cable_logical_ends()</td>
<td>ProCableLogicalEndsGet()</td>
</tr>
<tr>
<td>procbl_get_cable_geom()</td>
<td>ProCableIsComplete()</td>
</tr>
<tr>
<td>procbl_set_cable_params()</td>
<td>ProCableParametersSet()</td>
</tr>
<tr>
<td>procbl_delete_cable_param()</td>
<td>ProCableParameterDelete()</td>
</tr>
<tr>
<td>procbl_get_cable_param()</td>
<td>ProCableParameterGet()</td>
</tr>
<tr>
<td>procbl_get_cable_params()</td>
<td>ProCableParametersCollect()</td>
</tr>
<tr>
<td>Pro/DEVELOP Function</td>
<td>Equivalent Pro/TOOLKIT Function</td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>proclbl_cable_clearance()</td>
<td>ProCableClearanceCompute()</td>
</tr>
<tr>
<td>proclbl_get_cable_geom()</td>
<td>ProCablesegmentPointsGet()</td>
</tr>
<tr>
<td>proclbl_get_cable_geom()</td>
<td>ProCablesegmentIsInBundle()</td>
</tr>
<tr>
<td>proclbl_get_cable_geom()</td>
<td>ProCablesegmentIsNew()</td>
</tr>
<tr>
<td>proclbl_get_cable_locations()</td>
<td>ProCableLocationsCollect()</td>
</tr>
<tr>
<td>proclbl_get_location_cables()</td>
<td>ProCablelocationCablesGet()</td>
</tr>
<tr>
<td>proclbl_get_location_type()</td>
<td>ProCablelocationTypeGet()</td>
</tr>
<tr>
<td>proclbl_get_location_pnt()</td>
<td>ProCablelocationPointGet()</td>
</tr>
<tr>
<td>proclbl_routing_start()</td>
<td>ProCableRoutingStart()</td>
</tr>
<tr>
<td>proclbl_route_thru_location()</td>
<td>ProCableThruLocationRoute()</td>
</tr>
<tr>
<td>proclbl_routing_end()</td>
<td>ProCableRoutingEnd()</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>UDF Functions</strong></td>
<td></td>
</tr>
<tr>
<td>prodb_first_udf()</td>
<td>ProSolidGroupVisit() or ProSolidGroupsCollect()</td>
</tr>
<tr>
<td>prodb_next_udf()</td>
<td>ProSolidGroupVisit() or ProSolidGroupsCollect()</td>
</tr>
<tr>
<td>prodb_get_group()</td>
<td>ProFeatureGroupGet plus ProGroupFeatureVisit or ProGroupFeaturesCollect plus ProGroupIsTabledriven</td>
</tr>
<tr>
<td>prodb_get_udf_name()</td>
<td>ProUdfNameGet()</td>
</tr>
<tr>
<td>prodb_first_dim_udf()</td>
<td>ProUdfDimensionVisit() or ProUdfDimensionsCollect()</td>
</tr>
<tr>
<td>prodb_next_dim_udf()</td>
<td>ProUdfDimensionVisit() or ProUdfDimensionsCollect()</td>
</tr>
<tr>
<td>prodb_get_udf_dim_name()</td>
<td>ProUdfDimensionNameGet()</td>
</tr>
<tr>
<td>prodb_place_udf()</td>
<td>Replaced</td>
</tr>
<tr>
<td>prodb_create_group()</td>
<td>ProUdfCreate()</td>
</tr>
<tr>
<td>prodb_get_udf_instance_name()</td>
<td>ProUdfNameGet()</td>
</tr>
</tbody>
</table>
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 D - 30
- Ptc_surf Structure Definition on page D - 33
- Select3d Structure Definition on page D - 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_pnts;     /* 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

<table>
<thead>
<tr>
<th>Data Structure Constants</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTC_COLOR_DEFAULT</td>
<td>0</td>
<td>Use the default color.</td>
</tr>
<tr>
<td>PTC_COLOR_RGB</td>
<td>1</td>
<td>Specify the red, green, and blue color values.</td>
</tr>
<tr>
<td>PTC_COLOR_INDEX</td>
<td>2</td>
<td>Specify an index into the color table.</td>
</tr>
</tbody>
</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_splsrf      spl_srf;
    Ptc_b_splsrf    b_spl_srf;
    Ptc_cyl_splsrf  cyl_splsrf;
    Ptc_frgnsrf     frgnsrf;
} Ptc_srfshape;

typedef struct Ptc_plane     /* the local coordinate system */
{                             /* the local coordinate system */
    double   e1[3];          /* the local coordinate system */
    double   e2[3];          /* the local coordinate system */
    double   e3[3];          /* the local coordinate system */
    double   origin[3];      /* the local coordinate system */
} 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_pnts; /* 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_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_splsrf
{
    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_splsrf

typedef struct ptc_b_splsrf
{
    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_splsrf;

typedef struct ptc_gen_splsrf
{
    double       e1[3];
    double       e2[3];
    double       e3[3];
    double       origin[3];
    Ptc_splsrf   splsrf; /* in appropriate coordinates */
} Ptc_cyl_splsrf, Ptc_sph_splsrf;

typedef struct Ptc_frgnsrf
{
    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_frgnsrf;

typedef char * Prohandle;
## Data Structure Constants

<table>
<thead>
<tr>
<th>Data Structure Constant</th>
<th>Constant</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Surface types</td>
<td>PTC_OTHER_SRF</td>
<td>Another surface</td>
</tr>
<tr>
<td></td>
<td>PTC_PLANE</td>
<td>Plane</td>
</tr>
<tr>
<td></td>
<td>PTC_CYLINDER</td>
<td>Cylinder</td>
</tr>
<tr>
<td></td>
<td>PTC_CONE</td>
<td>Cone</td>
</tr>
<tr>
<td></td>
<td>PTC_TORUS</td>
<td>Torus</td>
</tr>
<tr>
<td></td>
<td>PTC_SRFREV</td>
<td>General surface of revolution</td>
</tr>
<tr>
<td></td>
<td>PTC_TABCYL</td>
<td>Tabulated cylinder</td>
</tr>
<tr>
<td></td>
<td>PTC_RULSRF</td>
<td>Ruled surface</td>
</tr>
<tr>
<td></td>
<td>PTC_COONS</td>
<td>Coons patch</td>
</tr>
<tr>
<td></td>
<td>PTC_FILSRF</td>
<td>Fillet surface</td>
</tr>
<tr>
<td></td>
<td>PTC_SPLSRF</td>
<td>Spline surface</td>
</tr>
<tr>
<td></td>
<td>PTC_B_SPLSRF</td>
<td>NURBS surface</td>
</tr>
<tr>
<td></td>
<td>PTC_FRGNSRF</td>
<td>Foreign surface</td>
</tr>
<tr>
<td></td>
<td>PTC_CYL_SPLSRF</td>
<td>Cylindrical spline surface</td>
</tr>
<tr>
<td>Line types</td>
<td>PRO_SOLIDFONT</td>
<td>Solid line</td>
</tr>
<tr>
<td></td>
<td>PRO_DOTFONT</td>
<td>Dotted line</td>
</tr>
<tr>
<td></td>
<td>PRO_CTRLFONT</td>
<td>Alternating long and short dashes</td>
</tr>
<tr>
<td></td>
<td>PRO_PHANTOMFONT</td>
<td>Alternating long dash and two dots</td>
</tr>
<tr>
<td>Entity types</td>
<td>PTC_POINT</td>
<td>Point</td>
</tr>
<tr>
<td></td>
<td>PTC_LINE</td>
<td>Line</td>
</tr>
<tr>
<td></td>
<td>PTC_ARC</td>
<td>Arc</td>
</tr>
<tr>
<td></td>
<td>PTC_SPLINE</td>
<td>Spline curve</td>
</tr>
<tr>
<td></td>
<td>PTC_B_SPLINE</td>
<td>B-spline curve</td>
</tr>
<tr>
<td></td>
<td>PTC_ELLIPSE</td>
<td>Ellipse</td>
</tr>
<tr>
<td></td>
<td>PTC_POLYGON</td>
<td>Polygon</td>
</tr>
</tbody>
</table>

### Select3d Structure Definition

Always initialize the Select3d structure using the `memset` function, as follows:

```c
Select3d  sel;
memset (&sel, '\0', sizeof (Select3d));
```

```c
sel.memb_id_tab[0] = -1;
mem_num = 0;
```
```c
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;    /* 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

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEL_3D_SRF</td>
<td>Surface</td>
</tr>
<tr>
<td>SEL_3D_EDG</td>
<td>Edge</td>
</tr>
<tr>
<td>SEL_3D_AXIS</td>
<td>Axis</td>
</tr>
<tr>
<td>SEL_3D_PNT</td>
<td>Point</td>
</tr>
<tr>
<td>SEL_3D_FEAT</td>
<td>Feature</td>
</tr>
<tr>
<td>SEL_3D_VERT</td>
<td>Vertex</td>
</tr>
</tbody>
</table>
The following table lists the possible options for the selection functions and the resultant data.

<table>
<thead>
<tr>
<th>option</th>
<th>sel_type</th>
<th>selected_ptr</th>
<th>selected_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>point</td>
<td>SEL_3D_PNT</td>
<td>point</td>
<td>feature</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The datum point ID is given by the Select3d field selElemId.</td>
</tr>
<tr>
<td>axis</td>
<td>SEL_3D_AXIS</td>
<td>axis</td>
<td>axis</td>
</tr>
<tr>
<td>datum</td>
<td>SEL_3D_SRF</td>
<td>surface</td>
<td>surface</td>
</tr>
<tr>
<td>csys</td>
<td>SEL_3D_CSYS</td>
<td>entity</td>
<td>entity</td>
</tr>
<tr>
<td>feature</td>
<td>SEL_3D_FEAT</td>
<td>feature</td>
<td>feature</td>
</tr>
<tr>
<td>edge</td>
<td>SEL_3D_EDG</td>
<td>edge</td>
<td>edge</td>
</tr>
<tr>
<td>edge_end</td>
<td>SEL_3D_VERT</td>
<td>edge</td>
<td>edge</td>
</tr>
<tr>
<td>curve</td>
<td>SEL_3D_CURVE</td>
<td>entity</td>
<td>entity</td>
</tr>
<tr>
<td>curve_end</td>
<td>SEL_CUVE_END</td>
<td>entity</td>
<td>entity</td>
</tr>
<tr>
<td>sldedge</td>
<td>SEL_3D_EDG</td>
<td>edge</td>
<td>edge</td>
</tr>
<tr>
<td>qltedge</td>
<td>SEL_3D_EDG</td>
<td>edge</td>
<td>edge</td>
</tr>
<tr>
<td>surface</td>
<td>SEL_3D_SRF</td>
<td>surface</td>
<td>surface</td>
</tr>
<tr>
<td>sldface</td>
<td>SEL_3D_SRF</td>
<td>surface</td>
<td>surface</td>
</tr>
<tr>
<td>qltface</td>
<td>SEL_3D_SRF</td>
<td>datum</td>
<td>datum</td>
</tr>
<tr>
<td>dtmqlt</td>
<td>SEL_3D_SRF_LIST</td>
<td>quilt</td>
<td>quilt</td>
</tr>
<tr>
<td>option</td>
<td>sel_type</td>
<td>selected_ptr</td>
<td>selected_id</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------------</td>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>dimension</td>
<td>DTL_DIM</td>
<td>N/A</td>
<td>dimension</td>
</tr>
<tr>
<td>ipar</td>
<td>SEL_3D_IPAR</td>
<td>N/A</td>
<td>parameter</td>
</tr>
<tr>
<td>gtol</td>
<td>DTL_GTOL</td>
<td>N/A</td>
<td>gtol</td>
</tr>
<tr>
<td>dtl_symbol</td>
<td>DTL_SYMBOL</td>
<td>N/A</td>
<td>symbol instance</td>
</tr>
<tr>
<td>dwg_table</td>
<td>SEL_DWG_TABLE</td>
<td>N/A</td>
<td>table</td>
</tr>
<tr>
<td>any_note</td>
<td>DTL_USER_NOTE</td>
<td>N/A</td>
<td>note</td>
</tr>
<tr>
<td>dwg_view</td>
<td>SEL_DWG_VIEW</td>
<td>view</td>
<td>view</td>
</tr>
<tr>
<td>dgm_obj</td>
<td>SEL_DGM_REF_OBJ</td>
<td>fixed connector, fixed component, or parametric connector</td>
<td>identifier for <code>selected_ptr</code></td>
</tr>
<tr>
<td>dgm_mv_obj</td>
<td>SEL_DGM_MV_REF_OBJ</td>
<td>multiview connector or multiview component</td>
<td>identifier for <code>selected_ptr</code></td>
</tr>
<tr>
<td>dgm_non_cable_wire</td>
<td>DTL_DGM_CONNECTION</td>
<td>wire is not in the cable</td>
<td>identifier for <code>selected_ptr</code></td>
</tr>
<tr>
<td>dgm_cable_conn</td>
<td>DTL_DGM_CONNECTION</td>
<td>cable conductor</td>
<td>identifier for <code>selected_ptr</code></td>
</tr>
<tr>
<td>dgm_cable</td>
<td>DTL_DGM_CONNECTION</td>
<td>cable</td>
<td>identifier for <code>selected_ptr</code></td>
</tr>
<tr>
<td>dgm_rail</td>
<td>SEL_DGM_RAIL</td>
<td>rail</td>
<td>identifier for <code>selected_ptr</code></td>
</tr>
<tr>
<td>dgm_hway</td>
<td>DTL_DGM_HIGHWAY</td>
<td>highway</td>
<td>identifier for <code>selected_ptr</code></td>
</tr>
<tr>
<td>draft_ent</td>
<td>DTL_DRAFT_ENT</td>
<td>N/A</td>
<td>entity</td>
</tr>
<tr>
<td>part</td>
<td>SEL_3D_PART</td>
<td>part</td>
<td>part</td>
</tr>
</tbody>
</table>
The following table lists the fields of the `Select3d` structure.

<table>
<thead>
<tr>
<th>option</th>
<th>sel_type</th>
<th>selected_ptr</th>
<th>selected_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>prt_or_asm</td>
<td>SEL_3D_PART</td>
<td>part, assembly</td>
<td>part, assembly</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Part and assembly IDs are not preserved over Pro/ENGINEER sessions.</td>
</tr>
<tr>
<td>table_cell</td>
<td>SET_TABLE CELL</td>
<td>table</td>
<td>Segment ID</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>part_ptr*</td>
<td>Part to which the <code>selected_ptr</code> belongs. In Assembly mode, this is really the component pointer (either a part or an assembly). NULL if irrelevant.</td>
</tr>
<tr>
<td>char</td>
<td>assembly_ptr</td>
<td>Root assembly; NULL if irrelevant.</td>
</tr>
<tr>
<td>int</td>
<td>memb_id_tab[]</td>
<td>Path to selected part through assembly members.</td>
</tr>
<tr>
<td>double</td>
<td>aux_depth</td>
<td>Not User Accessable</td>
</tr>
<tr>
<td>int</td>
<td>sel_elem_id</td>
<td>For tables, Segment ID</td>
</tr>
<tr>
<td>INT_FUNC</td>
<td>pre_filter</td>
<td>User-specified filter function</td>
</tr>
<tr>
<td>INT_FUNC</td>
<td>post_filter</td>
<td>User-specified filter function</td>
</tr>
<tr>
<td>char</td>
<td>*auxiliary</td>
<td>Use Not Recommended</td>
</tr>
<tr>
<td>char</td>
<td>*p_object</td>
<td>For tables, references the drawing model.</td>
</tr>
<tr>
<td>int</td>
<td>memb_num</td>
<td>Number of members in the assembly path.</td>
</tr>
<tr>
<td>int</td>
<td>sel_type</td>
<td>Type of the selected element. For tables, <code>SEL_TABLE CELL</code>.</td>
</tr>
<tr>
<td>Type</td>
<td>Field</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>int</td>
<td>selected_id</td>
<td>Identifier of the selected element.</td>
</tr>
<tr>
<td>char *</td>
<td>selected_ptr</td>
<td>Pointer to the selected element, for example, the table.</td>
</tr>
<tr>
<td>double</td>
<td>select_pnt[3]</td>
<td>xyz of the selected point in the member coordinate system.</td>
</tr>
<tr>
<td>double</td>
<td>sel_param[2]</td>
<td>uv parameters of selected point on surface; the t parameter for an edge (sel_param[0]). For tables, row and column number of a table cell, (sel_param[row,column]).</td>
</tr>
<tr>
<td>double</td>
<td>sel_depth</td>
<td>Selection depth, in model coordinates.</td>
</tr>
<tr>
<td>char *</td>
<td>view_ptr</td>
<td>View to which the selected_ptr belongs. This is used in Drawing mode.</td>
</tr>
<tr>
<td>int</td>
<td>window_id</td>
<td>Selected window ID</td>
</tr>
<tr>
<td>int</td>
<td>use_cache_node</td>
<td>PTC Internal</td>
</tr>
<tr>
<td>int</td>
<td>node_type</td>
<td>Type of node picked in tree tool, or 0</td>
</tr>
<tr>
<td>char</td>
<td>*cache_data</td>
<td>PTC Internal</td>
</tr>
<tr>
<td>char</td>
<td>*ptr_exp</td>
<td>PTC Internal</td>
</tr>
</tbody>
</table>
This appendix describes the Pro/TOOLKIT objects.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Pro/TOOLKIT Objects</td>
<td>E - 2</td>
</tr>
</tbody>
</table>
**List of Pro/TOOLKIT Objects**

The following table lists all the objects supported by Pro/TOOLKIT.

Items marked with an asterisk (*) are not actual Pro/TOOLKIT objects. However, they represent classes of functions that can be grouped under common concepts.

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProAnalysis</td>
<td>Analysis object</td>
</tr>
<tr>
<td>ProAnimation</td>
<td>Animation object</td>
</tr>
<tr>
<td>Pro/TOOLKIT functions for animation support two modes of animation—batch animation and single animation.</td>
<td></td>
</tr>
<tr>
<td>ProAnimFrame</td>
<td>Animation frame</td>
</tr>
<tr>
<td>ProAnimMovie</td>
<td>Animation movie</td>
</tr>
<tr>
<td>ProAnimObj</td>
<td>Object to be animated</td>
</tr>
<tr>
<td>Any object that is to be animated should be modeled as an animation object.</td>
<td></td>
</tr>
<tr>
<td>ProArray</td>
<td>Expandable array object</td>
</tr>
<tr>
<td>Expandable arrays are provided as a basic utility in Pro/TOOLKIT. These arrays enable applications to allocate an array of any object with no preset limit on its size. Many Pro/TOOLKIT functions use expandable arrays or return them.</td>
<td></td>
</tr>
<tr>
<td>ProAsm</td>
<td>Skeleton</td>
</tr>
<tr>
<td>ProAsmcomp</td>
<td>Assembly component feature</td>
</tr>
<tr>
<td>Components are building blocks for assemblies. They are parametric features. In application assemblies, a component can have a specific type. For example, in manufacturing assemblies, there is a workpiece component, a design part component, and so on.</td>
<td></td>
</tr>
<tr>
<td>ProAsmcomppath</td>
<td>Assembly component path object</td>
</tr>
<tr>
<td>In an assembly of arbitrary levels (an assembly that contains a subassembly and so forth), the assembly component path enables you to reference a particular component in the assembly.</td>
<td></td>
</tr>
<tr>
<td>ProAssembly</td>
<td>Assembly model</td>
</tr>
<tr>
<td>A Pro/ENGINEER assembly. It can be uniquely identified by its name and type.</td>
<td></td>
</tr>
<tr>
<td>Object</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------------------------</td>
</tr>
<tr>
<td>ProAsynchronous</td>
<td>Asynchronous object</td>
</tr>
<tr>
<td>ProAutodrill</td>
<td>Autodrilling sequence</td>
</tr>
<tr>
<td>ProAxis</td>
<td>Axis entity</td>
</tr>
<tr>
<td></td>
<td>An axis entity is a geometric entity that can belong to any feature. It has a unique identifier.</td>
</tr>
<tr>
<td>ProCavitylayout</td>
<td>Cavity layout</td>
</tr>
<tr>
<td>ProCavlayRule</td>
<td>Cavity layout rule</td>
</tr>
<tr>
<td>ProCl</td>
<td>Cutter location (CL) command</td>
</tr>
<tr>
<td></td>
<td>A single instruction within a CL file that can be generated in Manufacturing mode.</td>
</tr>
<tr>
<td>ProClCmd</td>
<td>CL Command elements</td>
</tr>
<tr>
<td></td>
<td>Enables you to add your own command (or action) to menu bar menus.</td>
</tr>
<tr>
<td>ProCollection</td>
<td>Collection</td>
</tr>
<tr>
<td>ProCollectioninstr</td>
<td>Collection instruction</td>
</tr>
<tr>
<td>ProCollectionref</td>
<td>Collection reference</td>
</tr>
<tr>
<td>ProColormap</td>
<td>Color map</td>
</tr>
<tr>
<td>ProCmd</td>
<td>Command</td>
</tr>
<tr>
<td>ProCompoundmenu</td>
<td>Compound menu</td>
</tr>
<tr>
<td>ProConfigopt</td>
<td>Configuration file option</td>
</tr>
<tr>
<td></td>
<td>The Pro/ENGINEER configuration file option.</td>
</tr>
<tr>
<td>ProContour</td>
<td>Contour of edges</td>
</tr>
<tr>
<td></td>
<td>A surface topologically consists of a set of contours. The contours in turn consist of a set of edges.</td>
</tr>
<tr>
<td>ProCsys</td>
<td>Coordinate system entity</td>
</tr>
<tr>
<td></td>
<td>A coordinate system entity is a geometric entity that can belong to any feature. It has a unique identifier.</td>
</tr>
<tr>
<td>ProCsysdata</td>
<td>Coordinate System Data</td>
</tr>
<tr>
<td>ProCurve</td>
<td>Curve entity</td>
</tr>
<tr>
<td></td>
<td>A curve entity is a geometric entity that can belong to any feature. A curve has a unique identifier. Curves can be of several types:</td>
</tr>
<tr>
<td>Line</td>
<td>-</td>
</tr>
<tr>
<td>Arc</td>
<td>-</td>
</tr>
<tr>
<td>Spline</td>
<td>-</td>
</tr>
<tr>
<td>Object</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Composite curve</td>
<td>Contains a set of curves</td>
</tr>
<tr>
<td>ProCurvedata</td>
<td>Curve data</td>
</tr>
<tr>
<td>ProDatum</td>
<td>Datum object</td>
</tr>
<tr>
<td>ProDatumdata</td>
<td>Datum data</td>
</tr>
<tr>
<td>ProDirectory</td>
<td>Working Directory</td>
</tr>
<tr>
<td>The working directory for Pro/ENGINEER.</td>
<td></td>
</tr>
<tr>
<td>ProDispdat</td>
<td>Display data for external object</td>
</tr>
<tr>
<td>The display data that describes how an external object should be displayed by Pro/ENGINEER.</td>
<td></td>
</tr>
<tr>
<td>ProDisplist</td>
<td>Display list</td>
</tr>
<tr>
<td>A display list captures the graphics information of Pro/ENGINEER.</td>
<td></td>
</tr>
<tr>
<td>ProDisplist2d</td>
<td>2-D display list</td>
</tr>
<tr>
<td>A 2-D display list. Used when the graphic being displayed is two-dimensional, such as drawings and sections.</td>
<td></td>
</tr>
<tr>
<td>ProDisplist3d</td>
<td>3-D display list</td>
</tr>
<tr>
<td>A 3-D display list. Used when the graphic being displayed is three-dimensional, such as parts and assemblies.</td>
<td></td>
</tr>
<tr>
<td>ProDtlattach</td>
<td>Detail attachment</td>
</tr>
<tr>
<td>ProDtllentity</td>
<td>Detail entity</td>
</tr>
<tr>
<td>ProDtllentitydata</td>
<td>Detail entity data</td>
</tr>
<tr>
<td>ProDtllgroup</td>
<td>Detail group</td>
</tr>
<tr>
<td>ProDtllgroupdata</td>
<td>Detail group data</td>
</tr>
<tr>
<td>*ProDtllitem</td>
<td>Detail item</td>
</tr>
<tr>
<td>In a drawing sheet, a detail item is everything except a view or a table (for example, draft entities, symbol instances, and notes).</td>
<td></td>
</tr>
<tr>
<td>ProDtlnote</td>
<td>Detail note</td>
</tr>
<tr>
<td>ProDtlnotedata</td>
<td>Detail note data</td>
</tr>
<tr>
<td>ProDtlnoteline</td>
<td>Detail note line</td>
</tr>
<tr>
<td>ProDtlnotetext</td>
<td>Detail note text</td>
</tr>
<tr>
<td>ProDtlsymdefattach</td>
<td>Detail symbol definition attachment</td>
</tr>
<tr>
<td>ProDtlsymdef</td>
<td>Detail symbol definition</td>
</tr>
<tr>
<td>Object</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>ProDtlsemdefdata</td>
<td>Detail symbol definition data</td>
</tr>
<tr>
<td>ProDtlseminst</td>
<td>Detail symbol instance</td>
</tr>
<tr>
<td>ProDtlseminstdata</td>
<td>Detail symbol instance data</td>
</tr>
<tr>
<td>ProDtlvartext</td>
<td>Detail variant text</td>
</tr>
<tr>
<td>ProDimension</td>
<td>Dimension</td>
</tr>
<tr>
<td>ProDtmpln</td>
<td>Datum Plane Feature</td>
</tr>
<tr>
<td>*ProDwg</td>
<td>Drawing</td>
</tr>
</tbody>
</table>

This is a Pro/ENGINEER drawing that is made up of multiple drawing sheets. This is the top-level type (the parallel of ProMdl) for Drawing mode.

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProDwgtable</td>
<td>A drawing table</td>
</tr>
<tr>
<td>ProDwgtabledata</td>
<td>Drawing table data</td>
</tr>
<tr>
<td>ProEdge</td>
<td>Edge</td>
</tr>
</tbody>
</table>

An edge of a Pro/ENGINEER part. A set of edges forms a contour, which in turn is part of a surface. Each edge has two surfaces adjacent to it. Edges define the extents of a surface.

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProEdgedata</td>
<td>Edge data</td>
</tr>
<tr>
<td>ProElement</td>
<td>Element object</td>
</tr>
</tbody>
</table>

An element object is an abstract representation for a specific database of Pro/ENGINEER. It can be used to represent the type of a feature, the cycle type in holemaking, or a section plane for a 3-D protrusion. Elements can have single or multiple values (ProValue object), be compound, or be an array type element. Elements form the building blocks for element trees (ProElemtree object).

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProElempath</td>
<td>Element path</td>
</tr>
</tbody>
</table>

This object is analogous to the assembly component path in that it enables a particular element in the element tree to be referenced.

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProElemtree</td>
<td>Element tree</td>
</tr>
</tbody>
</table>

An element tree is a compound element that has a set of child elements. The child elements may in turn branch out as subelement trees. Element trees are useful in representing complex data, while keeping the building blocks simple.

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProEngineer</td>
<td>Pro/ENGINEER</td>
</tr>
</tbody>
</table>

The Pro/ENGINEER executable.

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProExtdata</td>
<td>External data</td>
</tr>
</tbody>
</table>

External data associated with a Pro/ENGINEER model that can be defined by the Pro/TOOLKIT application. This data gets stored in the same file as the Pro/ENGINEER model.
<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProExtobj</td>
<td>External object</td>
</tr>
<tr>
<td></td>
<td>External objects associated with a Pro/ENGINEER model. The application can</td>
</tr>
<tr>
<td></td>
<td>define display, references, and other characteristics. Pro/ENGINEER manages</td>
</tr>
<tr>
<td></td>
<td>the storage, retrieval, display, selection, notifications, and so on. Each</td>
</tr>
<tr>
<td></td>
<td>external object has a unique integer identifier and type.</td>
</tr>
<tr>
<td>ProExtobjdata</td>
<td>External object data</td>
</tr>
<tr>
<td></td>
<td>The data of an external object, such as the display data or the selection</td>
</tr>
<tr>
<td></td>
<td>data.</td>
</tr>
<tr>
<td>ProExt</td>
<td>External reference</td>
</tr>
<tr>
<td>ProFaminstance</td>
<td>Family table instance (row)</td>
</tr>
<tr>
<td>ProFamtable</td>
<td>Family table</td>
</tr>
<tr>
<td>ProFeature</td>
<td>Feature</td>
</tr>
<tr>
<td></td>
<td>A feature in a Pro/ENGINEER part or assembly. Each feature has a unique</td>
</tr>
<tr>
<td></td>
<td>integer identifier.</td>
</tr>
<tr>
<td>ProFem</td>
<td>Finite Element Modeling (FEM)</td>
</tr>
<tr>
<td></td>
<td>Enables you to access the Pro/DEVELOP functions that access finite modeling</td>
</tr>
<tr>
<td></td>
<td>elements.</td>
</tr>
<tr>
<td>ProFemmesh</td>
<td>FEM Mesh</td>
</tr>
<tr>
<td>ProFile</td>
<td>File</td>
</tr>
<tr>
<td></td>
<td>A 40-character-long wide string that represents a file name, including its</td>
</tr>
<tr>
<td></td>
<td>extension and revision number.</td>
</tr>
<tr>
<td>ProFixture</td>
<td>Fixture Setup</td>
</tr>
<tr>
<td>ProFrgnCrv</td>
<td>Foreign curve</td>
</tr>
<tr>
<td></td>
<td>A datum curve whose evaluator function is provided within a Pro/TOOLKIT</td>
</tr>
<tr>
<td></td>
<td>application.</td>
</tr>
<tr>
<td>ProGeomitem</td>
<td>Geometrical item</td>
</tr>
<tr>
<td>ProGeomitemdata</td>
<td>Geometry data</td>
</tr>
<tr>
<td>ProGraphics</td>
<td>Graphic</td>
</tr>
<tr>
<td></td>
<td>Basic graphic items such as lines, arcs, and text can be drawn in a</td>
</tr>
<tr>
<td></td>
<td>Pro/ENGINEER window using the provided API.</td>
</tr>
<tr>
<td>ProGtol</td>
<td>Geometric tolerance</td>
</tr>
<tr>
<td>ProGtoldata</td>
<td>Geometric Tolerance Data</td>
</tr>
<tr>
<td>ProGtoldatumref</td>
<td>Geometric Tolerance Datum Ref</td>
</tr>
<tr>
<td>ProGtolleader</td>
<td>Geometric Tolerance Leade</td>
</tr>
<tr>
<td>ProGtolleaders</td>
<td>Geometric Tolerance Leaders</td>
</tr>
<tr>
<td>Object</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ProImportfeat</td>
<td>Enables you to manipulate an import feature.</td>
</tr>
<tr>
<td>ProIntf</td>
<td>Interface data</td>
</tr>
<tr>
<td>ProImm</td>
<td>Injection molding machine</td>
</tr>
<tr>
<td>ProLayer</td>
<td>Layer</td>
</tr>
<tr>
<td>ProMacro</td>
<td>Macro</td>
</tr>
<tr>
<td>ProMfg</td>
<td>Manufacturing model</td>
</tr>
<tr>
<td>ProMfgoper</td>
<td>Manufacturing operation</td>
</tr>
<tr>
<td>ProMaterial</td>
<td>Material</td>
</tr>
<tr>
<td>ProMdl</td>
<td>Pro/ENGINEER model</td>
</tr>
<tr>
<td>ProMenu</td>
<td>Menu</td>
</tr>
<tr>
<td>ProMenubar</td>
<td>Menu bar</td>
</tr>
<tr>
<td>ProMenubarMenu</td>
<td>Menu bar menu</td>
</tr>
<tr>
<td>ProMenubutton</td>
<td>Menu button</td>
</tr>
<tr>
<td>ProMessage</td>
<td>Message</td>
</tr>
<tr>
<td>ProMdl</td>
<td>Pro/ENGINEER model</td>
</tr>
<tr>
<td>ProMode</td>
<td>Pro/ENGINEER mode, such as Part or Sketcher mode.</td>
</tr>
</tbody>
</table>

A layer in a Pro/ENGINEER model. A layer is uniquely identified by its name.

A manufacturing model. This in turn references the manufacturing assembly. It is uniquely identified by a name and type.

A manufacturing operation. It is a type of a feature uniquely identified by name or an integer identifier.

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. A model is uniquely identified by its name and type.

A menu in Pro/ENGINEER. A menu is uniquely identified by its name.

The row of menus in the Pro/ENGINEER interface that includes, for example, the File, Info, and View menus.

A menu button in a menu. This is uniquely identified by the button name and the menu to which it belongs.

A message displayed the Pro/ENGINEER message window.
<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProModelitem</td>
<td>Model item</td>
</tr>
<tr>
<td></td>
<td>A model item is any item in a model. Each model item is a database object with</td>
</tr>
<tr>
<td></td>
<td>a unique identifier and type. A model item knows about its owner.</td>
</tr>
<tr>
<td>ProMoldbase</td>
<td>Moldbase Parameters</td>
</tr>
<tr>
<td>ProMouse</td>
<td>Mouse</td>
</tr>
<tr>
<td>Mouse-based input for Pro/ENGINEER.</td>
<td></td>
</tr>
<tr>
<td>ProNcseq</td>
<td>NC sequence</td>
</tr>
<tr>
<td></td>
<td>An NC sequence in a manufacturing assembly. It represents a class of features</td>
</tr>
<tr>
<td></td>
<td>used to create a toolpath. NC sequences are uniquely identified by name or by</td>
</tr>
<tr>
<td></td>
<td>an integer identifier.</td>
</tr>
<tr>
<td>ProNote</td>
<td>Note</td>
</tr>
<tr>
<td>Notes in Pro/ENGINEER models.</td>
<td></td>
</tr>
<tr>
<td>ProNotification</td>
<td>Notification upon object actions</td>
</tr>
<tr>
<td>Pro/ENGINEER can issue notifications before or after certain actions are</td>
<td></td>
</tr>
<tr>
<td>performed on Pro/ENGINEER objects. For example, you can set up a</td>
<td></td>
</tr>
<tr>
<td>notification to be issued by Pro/ENGINEER when a model it erased.</td>
<td></td>
</tr>
<tr>
<td>ProOption</td>
<td>Option</td>
</tr>
<tr>
<td>ProParameter</td>
<td>Parameter</td>
</tr>
<tr>
<td>Parameters that belong to models, features, and external objects can be</td>
<td></td>
</tr>
<tr>
<td>manipulated using the supplied functions.</td>
<td></td>
</tr>
<tr>
<td>ProParamvalue</td>
<td>Parameter value</td>
</tr>
<tr>
<td>The value of a Pro/ENGINEER parameter.</td>
<td></td>
</tr>
<tr>
<td>ProPart</td>
<td>Part model</td>
</tr>
<tr>
<td>A part model in Pro/ENGINEER.</td>
<td></td>
</tr>
<tr>
<td>ProPattern</td>
<td>Pattern</td>
</tr>
<tr>
<td>The method for creating patterns in Pro/TOOLKIT is similar to the method</td>
<td></td>
</tr>
<tr>
<td>for creating features.</td>
<td></td>
</tr>
<tr>
<td>ProPecktable</td>
<td>Peck table</td>
</tr>
<tr>
<td>A peck table in a manufacturing assembly.</td>
<td></td>
</tr>
<tr>
<td>ProPlotfile</td>
<td>Plot File</td>
</tr>
<tr>
<td>ProPoint</td>
<td>Point entity</td>
</tr>
<tr>
<td>A point entity is a geometric entity that can belong to any feature. Most</td>
<td></td>
</tr>
<tr>
<td>commonly, points can be generated by creating the datum point feature. A</td>
<td></td>
</tr>
<tr>
<td>point has a unique integer identifier.</td>
<td></td>
</tr>
<tr>
<td>Object</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ProPrint</td>
<td>Printing</td>
</tr>
<tr>
<td>ProProcstep</td>
<td>Process step</td>
</tr>
<tr>
<td>ProRef</td>
<td>External reference control</td>
</tr>
<tr>
<td>ProQuilt</td>
<td>Quilt</td>
</tr>
</tbody>
</table>

Quilts are the results of “weaving” surface patches together. Quilts are most commonly generated by creating a datum surface feature. Each quilt has a unique integer identifier.

| ProQuiltdata    | Quilt data                                                                  |
| ProRel          | Relation                                                                    |
| ProRelset       | Relation set                                                                |
| ProRmdt         | Mold layout                                                                 |
| ProRule         | Rule for gathering assembly components                                      |

A list of assembly components can be generated by specifying rules. This is used to include or exclude components when defining simplified representations.

| ProSecdim       | Section dimension                                                          |
| ProSecerror     | Section error                                                              |

Errors generated as a result of section regeneration.

| ProSection      | Section model                                                              |

The section model. This represents the common characteristics of 2-D and 3-D sections.

| ProSeldata      | Selection data for external object                                         |

The selection data for external objects. This needs to be specified in an external object in order to make it selectable.

| ProSelection    | Selection object                                                           |

A selection object is commonly generated as a result of selecting any Pro/ENGINEER object. The selection object contains not only the information of the object selected, but also other useful pieces of information, such as the location of this Pro/ENGINEER object in the assembly hierarchy, the coordinates of the selection, and so on.

| ProSimprep      | Simplified representation                                                  |

The simplified representation of a Pro/ENGINEER assembly.

| ProSimprepdata  | Simplified representation definition                                         |

The data used to define simplified representations.
<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProSimprepdataitem</td>
<td>Item in a simplified representation.</td>
</tr>
<tr>
<td>The items that belong to a simplified representation.</td>
<td></td>
</tr>
<tr>
<td>ProSmt</td>
<td>Sheet metal</td>
</tr>
<tr>
<td>ProSolid</td>
<td>Base class object for a part or an assembly</td>
</tr>
<tr>
<td>A solid is an abstract Pro/ENGINEER object that represents all the common characteristics of parts and assemblies, such as features, dimensions, and parameters.</td>
<td></td>
</tr>
<tr>
<td>ProSubstitution</td>
<td>Simplified reps substitution</td>
</tr>
<tr>
<td>ProSurface</td>
<td>Surface</td>
</tr>
<tr>
<td>A surface in Pro/ENGINEER. This is owned by a feature and has a unique integer identifier. Pro/ENGINEER supports a rich variety of surfaces, such as plane, cones, and Coons patches.</td>
<td></td>
</tr>
<tr>
<td>ProSurfacedata</td>
<td>Surface data</td>
</tr>
<tr>
<td>ProText</td>
<td>Text</td>
</tr>
<tr>
<td>The displayed text in a Pro/ENGINEER graphics window.</td>
<td></td>
</tr>
<tr>
<td>ProTool</td>
<td>Tool in a manufacturing assembly</td>
</tr>
<tr>
<td>A tool in a manufacturing assembly.</td>
<td></td>
</tr>
<tr>
<td>ProToolinput</td>
<td>Input for creating a tool</td>
</tr>
<tr>
<td>Tool input is used to define a tool for programmatic creation.</td>
<td></td>
</tr>
<tr>
<td>ProToolpath</td>
<td>Tool path</td>
</tr>
<tr>
<td>ProUI</td>
<td>UI dialog box components</td>
</tr>
<tr>
<td>ProValue</td>
<td>Value</td>
</tr>
<tr>
<td>ProVerstamp</td>
<td>Version stamp</td>
</tr>
<tr>
<td>Version stamps on Pro/ENGINEER objects that are used to detect and manage change in Pro/ENGINEER objects.</td>
<td></td>
</tr>
<tr>
<td>ProView</td>
<td>View</td>
</tr>
<tr>
<td>A view in Pro/ENGINEER. A Pro/ENGINEER window consists of a number of views. Also, a Pro/ENGINEER model can have number of named views associated with it.</td>
<td></td>
</tr>
<tr>
<td>ProWeld</td>
<td>Weld</td>
</tr>
<tr>
<td>ProWcell</td>
<td>Workcell feature of a manufacturing assembly</td>
</tr>
<tr>
<td>A workcell in a manufacturing assembly. This is a type of a feature that can be uniquely identified by name or by an integer identifier.</td>
<td></td>
</tr>
<tr>
<td>ProWindow</td>
<td>Window</td>
</tr>
</tbody>
</table>

E - 10

Pro/TOOLKIT User’s Guide
<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProWorkspace</td>
<td>Workspace</td>
</tr>
<tr>
<td>ProWstring</td>
<td>Wide String</td>
</tr>
<tr>
<td>ProXsec</td>
<td>Cross section</td>
</tr>
<tr>
<td>ProCircledata</td>
<td>-</td>
</tr>
<tr>
<td>ProArrowdata</td>
<td>-</td>
</tr>
<tr>
<td>ProPolygondata</td>
<td>-</td>
</tr>
<tr>
<td>ProTextdata</td>
<td>-</td>
</tr>
<tr>
<td>ProCsysdata</td>
<td>-</td>
</tr>
<tr>
<td>ProCsysdata</td>
<td>-</td>
</tr>
<tr>
<td>ProDr1group</td>
<td>-</td>
</tr>
<tr>
<td>ProDr1group</td>
<td>-</td>
</tr>
<tr>
<td>ProExtobjClass</td>
<td>-</td>
</tr>
<tr>
<td>ProExtobjCB</td>
<td>-</td>
</tr>
<tr>
<td>ProExtobjRef</td>
<td>-</td>
</tr>
<tr>
<td>ProFamtableItem</td>
<td>-</td>
</tr>
<tr>
<td>ProHle</td>
<td>-</td>
</tr>
<tr>
<td>ProSrf</td>
<td>-</td>
</tr>
<tr>
<td>ProDtmpln</td>
<td>-</td>
</tr>
<tr>
<td>ProDtmpnt</td>
<td>-</td>
</tr>
</tbody>
</table>
Geometry Traversal

This appendix illustrates the relationships between faces, contours, and edges.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>F - 2</td>
</tr>
<tr>
<td>Geometry Terms</td>
<td>F - 2</td>
</tr>
</tbody>
</table>
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**

![Diagram of a part with faces, contours, and edges labeled](image)

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

![Diagram of Face A with contours and edges]

Face A has 2 contours and 6 edges.

Example 3

![Diagram of extruded part with faces and protrusion]

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 ‘Geometry’ chapter.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain of Evaluation</td>
<td>G - 2</td>
</tr>
<tr>
<td>Surface Data Structures</td>
<td>G - 3</td>
</tr>
<tr>
<td>Edge and Curve Data Structures</td>
<td>G - 18</td>
</tr>
</tbody>
</table>
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.

![Diagram showing surfaces and edges with and without draft feature](image)
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
- General Surface of Revolution
- NURBS Surface
- Plane
- Ruled Surface
- Spline Surface
- Tabulated Cylinder
- Torus
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
- \( \text{origin}[3] \): Origin of the plane

Parameterization:

\[
(x, y, z) = u \cdot e1 + v \cdot e2 + \text{origin}
\]
Cylinder

Figure G-2: 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:

- $e_1[3]$ Unit vector, in the $u$ direction
- $e_2[3]$ Unit vector, in the $v$ direction
- $e_3[3]$ Normal to the plane
- radius Radius of the cylinder

Parameterization:

$$(x, y, z) = \text{radius} \times [\cos(u) \times e_1 + \sin(u) \times e_2] + v \times e_3 + \text{origin}$$

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 ($e_1$, $e_2$, and $e_3$) and an origin. The curve lies in the plane of $e_1$ and $e_3$, and is rotated in the direction from $e_1$ to $e_2$. The $u$ surface parameter determines the angle of rotation, and the $v$ parameter determines the position of the point on the generating curve.
The generating curve of a cone is a line at an angle alpha 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 \( v \times \tan(alpha) \).

Data Format:

- \( e_1[3] \)  Unit vector, in the \( u \) direction
- \( e_2[3] \)  Unit vector, in the \( v \) direction
- \( e_3[3] \)  Normal to the plane
- \( \text{origin}[3] \)  Origin of the cone
- \( \alpha \)  Angle between the axis of the cone and the generating line

Parameterization:

\[
(x, y, z) = v \times \tan(alpha) \times (\cos(u) \times e_1 + \sin(u) \times e_2) + v \times e_3 + \text{origin}
\]
The generating curve of a torus is an arc of radius $R_2$ with its center at a distance $R_1$ from the origin. The starting point of the generating arc is located at a distance $R_1 + R_2$ 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 $R_1 + R_2 \cos(v)$, and the height of the point along the axis of revolution is $R_2 \sin(v)$.

Data Format:

- $e_1[3]$: Unit vector, in the $u$ direction
- $e_2[3]$: Unit vector, in the $v$ direction
- $e_3[3]$: Normal to the plane
- $\text{origin}[3]$: Origin of the torus
- $\text{radius}_1$: Distance from the center of the generating arc to the axis of revolution
- $\text{radius}_2$: Radius of the generating arc

Parameterization:

$$(x, y, z) = (R_1 + R_2 \cos(v)) \cdot [\cos(u) \cdot e_1 + \sin(u) \cdot e_2] + R_2 \sin(v) \cdot e_3 + \text{origin}$$
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
- \(\text{origin}[3]\) Origin of the surface of revolution
- \(\text{curve}\) Generating curve

Parameterization:

- \(\text{curve}(v) = (c_1, c_2, c_3)\) is a point on the curve.

\[
(x, y, z) = [c_1 \cos(u) - c_2 \sin(u)] \cdot e_1 + [c_1 \sin(u) + c_2 \cos(u)] \cdot e_2 + c_3 \cdot e_3 + \text{origin}
\]
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.
- $(x', y', z') = (1 - v) \cdot C1(u) + v \cdot C2(u)$
- $(x, y, z) = x' \cdot e1 + y' \cdot e2 + z' \cdot e3 + origin$
Tabulated Cylinder

Figure G-7: 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:
- $e_1[3]$  Unit vector, in the $u$ direction
- $e_2[3]$  Unit vector, in the $v$ direction
- $e_3[3]$  Normal to the plane
- $\text{origin}[3]$  Origin of the tabulated cylinder curve
- Generating curve

Parameterization:

$(x', y', z')$ is the point in local coordinates.

$(x', y', z') = C(u) + (0, 0, v)$

$(x, y, z) = x' \cdot e_1 + y' \cdot e_2 + z' \cdot e_3 + \text{origin}$
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

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:

\[
R(v) = P(v) - C(v)
\]
\[
(x, y, z) = C(v) + R(v) \cdot \cos(u) + T(v) \times R(v) \cdot \sin(u)
\]
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:**

- $u_{\text{par}}_{\text{arr}}[]$ (Point parameters, in the $u$ direction, of size $Nu$)
- $v_{\text{par}}_{\text{arr}}[]$ (Point parameters, in the $v$ direction, of size $Nv$)
- $\text{point}_{\text{arr}}[3]$ (Array of interpolant points, of size $Nu \times Nv$)
- $\text{u}_{\text{tan}}_{\text{arr}}[3]$ (Array of $u$ tangent vectors at interpolant points, of size $Nu \times Nv$)
- $\text{v}_{\text{tan}}_{\text{arr}}[3]$ (Array of $v$ tangent vectors at interpolant points, of size $Nu \times Nv$)
- $\text{uvder}_{\text{arr}}[3]$ (Array of mixed derivatives at interpolant points, of size $Nu \times Nv$)

**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 $\text{point}_{\text{arr}}[3]$, you will find that the innermost variable representing $v(j)$ varies first.
NURBS Surface

The NURBS surface is defined by basis functions (in $u$ and $v$), expandable arrays of knots, weights, and control points.

Figure G-11: NURBS Surface

Cubic NURBS Surface

Data Format:

- `deg[2]`: Degree of the basis functions (in $u$ and $v$)
- `u_par_arr[]`: Array of knots on the parameter line $u$
- `v_par_arr[]`: Array of knots on the parameter line $v$
- `wghts[]`: Array of weights for rational NURBS, otherwise NULL
- `c_point_arr[] [3]`: 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 \((x,y,z) \times w_{i,j}\)

Engineering Notes:

The weights and \( c\_points\_arr \) arrays represent matrices of size \( wghts[N1+1][N2+1] \) 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 G-12: 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
- splsrf  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 \( Nu \times Nv \)

uvder_arr[] [3]  Array of mixed derivatives, in cylindrical coordinates, of size \( Nu \times Nv \)

Engineering Notes:

If the surface is represented in cylindrical coordinates \((r, \theta, z)\), 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 \((r_1 = r_2, r_3 = r_4, \text{ and } r_1 \neq r_3)\).

If a replacement cannot be done (such as for the surface \(S_0\) in the figure \((ra \neq rb \text{ or } rc \neq rd)\)), leave it as a cylindrical spline surface representation.
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 G - 20
- Line on page G - 19
- NURBS on page G - 22
- Spline on page G - 21
Line

Data Format:

end1[3]   Starting point of the line
end2[3]   Ending point of the line

Parameterization:

\[(x, y, z) = (1 - t) \cdot \text{end1} + t \cdot \text{end2}\]
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:

- \text{vector1[3]} \quad \text{First vector that defines the plane of the arc}
- \text{vector2[3]} \quad \text{Second vector that defines the plane of the arc}
- \text{origin[3]} \quad \text{Origin that defines the plane of the arc}
- \text{start\_angle} \quad \text{Angular parameter of the starting point}
- \text{end\_angle} \quad \text{Angular parameter of the ending point}
- \text{radius} \quad \text{Radius of the arc}

Parameterization:

- \( t' \) (the unnormalized parameter) is \( (1 - t) * \text{start\_angle} + t * \text{end\_angle} \)
- \((x, y, z) = \text{radius} * \left[ \cos(t') * \text{vector1} + \sin(t') * \text{vector2} \right] + \text{origin}\)
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 \( p_{\text{max}} \) be the parameter of the last spline point. Then, \( t' \), the unnormalized parameter, is \( t' \) * \( p_{\text{max}} \).

Locate the \( i \)th spline segment such that:

\[
\text{par}_\text{arr}[i] < t' < \text{par}_\text{arr}[i+1]
\]

(If \( t < 0 \) or \( t > +1 \), use the first or last segment.)

\[
t0 = \frac{t' - \text{par}_\text{arr}[i]}{\text{par}_\text{arr}[i+1] - \text{par}_\text{arr}[i]} \\
t1 = \frac{\text{par}_\text{arr}[i+1] - t'}{\text{par}_\text{arr}[i+1] - \text{par}_\text{arr}[i]}
\]
The NURBS (nonuniform rational B-spline) curve is defined by expandable arrays of knots, weights, and control points.

**Figure G-13: 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:

\[
R(t) = \sum_{i=0}^{N} \frac{C_i \times B_{i,k}(t)}{\sum_{i=0}^{N} w_i \times B_{i,k}(t)}
\]

- \(k\) = 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 \(N+1\).

References:

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.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>H - 2</td>
</tr>
<tr>
<td>ptk_revtool Utility Features</td>
<td>H - 2</td>
</tr>
<tr>
<td>Running ptk_revtool</td>
<td>H - 2</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-#</td>
<td>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.</td>
</tr>
<tr>
<td>input_file</td>
<td>File name of the Pro/TOOLKIT application that you want to update.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Output Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>updated file filename.c</td>
<td>The specified file was successfully updated.</td>
</tr>
<tr>
<td>no symbols to change</td>
<td>The -# option wasn’t provided. No changes made.</td>
</tr>
<tr>
<td>no symbols changed in file filename</td>
<td>The file is already up-to-date. No changes were made to the file.</td>
</tr>
<tr>
<td>ptk_revtool: can’t open file</td>
<td>File not found. Check the file name and path. You must enter full file name, including extension.</td>
</tr>
<tr>
<td>bad arg ’-#’; version must be &gt;= 19</td>
<td>You entered a version number less than 19. The program works only on version 19 or greater.</td>
</tr>
<tr>
<td>no symbols for version</td>
<td>-</td>
</tr>
<tr>
<td>Warning: too-long, MY_NAME written to</td>
<td>Output file name greater than 32 characters.</td>
</tr>
</tbody>
</table>
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 ‘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.
Numerics

2-D
display list 8-18
sections
  adding dimensions to 23-5
  adding entities 23-4
  allocating 23-3
3-D
display list 8-18

A

Access
  material data 7-11
  menu buttons 4-42
  parameters 14-4
to explode states 13-15
Accuracy
  of solids 7-7
Action functions
  for visits 1-56
Actions
  adding 4-5
Activate
  explode state 13-15
  window 8-5
Add
  action to Pro/ENGINEER 4-5
  animation frames to a movie 46-5
  animation objects to frames 46-4
dimensions to a 2-D section 23-5
external object data 44-11
family table items 34-6
items to layers 6-13
leaders 7-17
menu buttons 4-28
menu to menu bar 4-15
pattern table instances 12-16
section entities 23-4
add or update 1-23
Allocate
  2-D sections 23-3
display data 44-7
external object references 44-13
note attachments 7-17
selection data for external objects 44-10
simplified representations 38-6
text style structures 7-17
version stamps 6-10
Analyze
  manufacturing model 30-3
Angles
  example 9-21
Animation 46-2
  batch 46-5
  creating 46-3
dialog box 46-2
  frame
description 46-3
movies 46-5
object
description 46-3
refresh 8-3
single 46-5
APIWizard
  browsing
description 1-10
functions 1-10
objects 1-8
user's guide 1-10
code examples 1-7
defined 1-2
display frame 1-7

Index - i
documentation online 1-2
PDF format 1-2
find (search mechanism) 1-11
frames in web browser 1-7
function descriptions 1-7
interface defined 1-7
Internet Explorer
Java2 plug in 1-3, 1-5
Java2 plugin 1-6
requirements 1-3, 1-5
troubleshooting 1-5
navigating the tree 1-8
Netscape Navigator
Java2 plugin not recommended 1-6
requirements 1-3
object description 1-7
searching for a string 1-11
supported web browsers 1-3
Swing
class path-NT 1-5
class path-UNIX 1-4
download JFC archive 1-4
Java Foundation Class 1-3
topic/object selection frame 1-7
tree updating
defined 1-6
Java2 plugin requirements 1-6
Applications
compiling and linking 1-30
core 1-40
debugging 1-36
program structure D-7
registering 1-31
stopping and restarting 1-35
structure of D-7
unlocking 1-38
using with Pro/TOOLKIT D-3
Arcs
drawing 8-15
edges
extracting the diameter 9-19
representation G-20
Argument
Management 53-2
Arrays
element
description 16-5
expandable 1-50
allocating 1-51
code example 1-53
freeing 1-51
Assemblies
active explode state 13-15
automatic interchange 13-16
components 13-2
assembling 13-13
assembling by feature creation 22-13
deleting 13-14
locating 13-11
traversing 13-5
visiting 13-5
conversion paths D-11
coordinate systems 10-3
cut out 13-15
explode states 13-15
exploded 13-14
hierarchy 13-4
interference checking 9-28
merging 13-15
process step 33-2
structure of 13-2
Asynchronous mode
definition Glossary-1
full 52-8, 52-10
non-interactive 52-9
simple 52-7
Attachment
points and leaders 39-55
location 7-17
Attributes
text 8-17
user
definition Glossary-7
Automatic dimensioning 23-10
Automatic interchange 13-16
Auxiliary Applications
dialog box 1-35
Auxiliary tools
parameters 30-11
Axes
datums
visiting 9-10
gallery of 9-22
Backup
model 6-6
Bar elements 43-11
Base window
identifier 8-1
Batch animation 46-5
code example 46-6
Batch mode
element 1-43
Batch sessions 1-42
Bind
evaluation functions 27-7
Browsing
Pro/TOOLKIT functions with APIWizard 1-10
Pro/TOOLKIT objects with APIWizard 1-8
Pro/TOOLKIT user's guide with APIWizard 1-10

Button
placing 4-22

Buttons
accessibility 4-42
adding 4-28
example 4-30
adding to menu bar 4-8
menu bar 4-2
mode-specific 4-25
position 8-20
setting 4-41

C

Cabling
cable geometry 42-18
cable identifiers 42-14
cable types 42-14
connectivity 42-15
harness clearance 42-18
parameters 42-15
routing locations 42-17
routing procedure 42-19

Callbacks
external Analysis 29-5
external analysis feature 29-5
for external objects 44-14
for notification 36-6

Chamfers 19-2
dimension schemes 19-3

Check out 47-2
Checks 12-5
Child
definition Glossary-1

Circles
code example 8-16
drawing 8-15
CL commands 32-3
customizing 32-5

Classes 35-2
external objects 44-3
notification 36-2
of foreign datum surfaces 27-2

Classification of messages
critical 3-8
error 3-8
info 3-8
prompt 3-7
warning 3-8

Clear
single animation 46-5

window 8-3

Clearance
harness 42-18

Close
windows 8-4

Collect functions 1-53

Color
coloring 8-13
external objects 44-9
graphics 8-10
map modifying 8-13

Command
adding 4-20
Commands
designating 4-20
Pro/ENGINEER
entering 4-44
preempting 4-37

Compatibility
Pro/TOOLKIT and Pro/ENGINEER versions 1-33

Compiling 1-30

Complement mode
definition Glossary-1

Component assembly
code example 22-15
constraints 22-15
element tree 22-14

Components
manufacturing
traversing 30-5
simplified representation
gathering 38-14

Compound element 16-5

Compound menus 4-36

Cone G-6

Configuration file
definition Glossary-1
options
getting and setting 49-2
PROTKDAT option 1-31
toolkit_registry_file option 1-31

Confirmation
using menu buttons 4-38

Connectivity
cable 42-15
diagram 42-5

Connectors
finding 42-9
parameters 42-10
file 42-10

Constraints
case names 43-5
component assembly 22-15
FEM 43-8
visiting 9-12
definition Glossary-2
evaluating 9-15
foreign datum 27-3
Customized plot driver 41-1
Cut out 13-15
Cylinders G-5
  spline surfaces G-16
tabulated G-10

D

Data
  external object 44-6
  handles
    description 1-48
    material 7-11
types D-7
Data menus 4-40
Data types
  Pro/DEVELOP versus Pro/TOOLKIT D-7
Database
  search 31-4
Database items
  conversion paths D-8
Datum
  axes
    visiting 9-10
  coordinate systems
    visiting 9-11
curves
  geometry 9-23
  parametric equations 9-18
    visiting 9-12
  planes
    creating
      code example 16-17
    geometry 9-22
      visiting 9-11
  points
    geometry 9-23
      visiting 9-13
  surfaces
    foreign 27-2
    geometry 9-22
Datum axis
  creating 17-57
  feature element tree 17-50
  normal planes 17-56
  point on surface 17-51
  tangent 17-52
  through edge or surface 17-53
  two planes 17-54
  two points 17-55
Datum coordinate system
  creating 17-69
  feature element tree 17-62
  orienting by selecting csys axes 17-87
  orienting by selecting references 17-66
  using 3Planes or 2 edges and axes 17-64
  using a csys 17-67
  using a vertex or a datum point 17-66
  using curve, edges, or plane and axis 17-65
Datum plane
  creating 17-7
  feature element tree 17-3
Datum point
  at an offset 17-43
  at center of curve or surface 17-47
  at intersection of 3 surfaces 17-45
  at intersection of a curve and a surface 17-46
  feature element tree 17-12
  field 17-19
  general 17-30
  offset csys 17-23
  on a vertex 17-43
  on curve 17-48
  on/offset from a Surface 17-45
  sketched 17-12
Debugging
  applications 1-36
    already running C-2
    DLL C-5
    invoking as a separate process C-3
  on Windows and NT systems C-4
Default values 3-9
  constant 3-10
  in text box 3-10
  variable 3-10
Delete
  animation frames 46-4
  animation objects 46-3
  cross sections 40-5
  display lists 8-18
  external object classes 44-3
  external objects 44-3
  features 12-6, 13-14
  layers 6-13
  material data 7-11
  menus 4-30
  models 6-6
  pattern 12-12
  pattern tables 12-16
  relation sets 11-10
  section dimensions 23-5
  section entities 23-4
  simplified representations 38-6
  windows 8-4
Deleting Cable Sections Cable
  Delete 42-20
E

Edge and curve data structures  G-18
Edge chamfers  19-2
Edges
  definition Glossary-2
  evaluating  9-15
  getting the description of  9-18
  parametric equations  9-18
  traversing F-2
  types  9-18
Edit
  files  49-4
Element Tree
  extruded features  25-2
  revolved features  25-58
Element tree
  first features  25-90
Element trees
  auxiliary NC sequences  32-2
  custom NC sequences  32-3
  description  16-3
  patterns  21-2
Elements
  definition Glossary-2
  in an element tree  16-3
  paths  16-11
  roles  16-4
  types  16-4
  types of  16-4
  values  16-5
Entities
  adding to 2-D sections  23-4
  definition Glossary-3
Epsilon
  specifying  23-9
Equations
  geometry  9-16
  parametric  9-18
  of surfaces  9-20
Erase
  family table instances  34-5
  family tables  34-2
  models  6-6
Errors
  section  23-12
Evaluation
  definition Glossary-3
  domain of G-2
  functions
    for foreign datum curves  27-4
  inverse  9-16
  of faces, edges, and curves  9-15
  of geometry  9-15
  of relations  11-10
Events
  trapping  36-1
EXample
  weld callback notification  37-4
Example
  creating a datum axis  17-57
  creating a datum coordinate system  17-69
  creating a datum plane  17-7
  creating a field datum point  17-20
  creating a sketched datum point  17-13
  creating an offset csys datum point  17-25
  creating general datum point  17-35
  offset coordinate system datum  29-11
example  52-7, 52-10
Examples
  adding a menu button  4-30
  adding help text  4-27
  adding items to a layer  6-14, 6-18
  adding surfaces to an element  30-33
  adding to menu bar menus  4-16
  asking for confirmation on Quit Window
    4-38
  assembling a part into an assembly  22-15
  batch mode  1-43
  calculating the mass properties of a cross
    section  40-6
  computing the outline of a solid  7-6
  converting strings  1-63
  copying an assembly from the Workspace
    47-3
  creating a 2-axis lathe workcell  30-24
  creating a batch animation  46-6
  creating a conventional milling sequences
    30-35
  creating a datum plane  16-17
  creating a menu that selects a value  4-35
  creating a note  7-18
  creating a parameter tree  30-18
  creating a parameter-driven tool  30-15
  creating a pattern  21-9
  creating a section  23-14
  creating a sinusoidal foreign datum curve
    27-7
  creating a sweep  26-5
  creating a sweep protrusion feature by
    conventional approach  26-13
  creating a tool from a solid model  30-13
  creating an edge chamfer feature  19-5
  creating an external object  44-18
  creating an extruded feature  25-8
  creating an operation  30-28
  creating drawing views  39-16
  defining a new menu that closes itself  4-32
  defining a new menu that the user must
    close  4-33
  designating a command  4-24
  displaying a solid  7-3

Index - vii
displaying lines and circles  8-16
displaying messages  3-10
drawing a rubber-band line  8-21
extracting the diameter of an arc edge  9-19
finding the handle to a model  6-5
finding the position of a component  13-12
finding the surface penetrated by a hole  9-8
getting the angle of a conical surface  9-21
identifying workcell features of a Pro/NC model  30-6
inserting a feature  12-7
interference checking for assemblies and parts  9-28
labeling a feature with a string parameter  14-5
listing cross sections  40-3
listing the holes in a model  1-58
listing the members of an assembly  13-8
listing views  39-11
loading and displaying a solid  7-3
menu file  4-27
modifying colors  8-13
reading material properties from a file  7-12
renaming a selected surface  6-9
reordering features  12-9
retrieving keyboard input  3-10
saving views  8-7
transforming screen to drawing coordinates  10-7
transforming solid coordinates  10-6
using a new menu  4-34
using drawing sheets  39-8
using expandable arrays  1-53
using the FEM functions  43-14
using the model item functions  6-9
using the plot driver functions  41-3
visiting the items in a simplified representation  38-9
writing a family table to a file  34-6
Exit actions  defining  4-32
Expandable arrays  1-50
allocating  1-51
code example  1-53
freeing  1-51
Explode states  access  13-15
activating  13-15
visiting  13-15
Exploded assemblies  13-14
Export  FEA mesh  43-3
files  50-2
External analysis attributes  get and set  29-29
PROANALYSIS_COMPUTE_OFF  29-29
callbacks  29-5
defined  29-2
do not use global variables with  29-8
entity shape union  29-10
entity type enum  29-9
feature parameter structure  29-9
graph item structure  29-9
interactive creation  29-3
use without Pro/ENGINEER  29-30
External analysis feature  defined  29-2
interactive creation  29-3
storage as feature dimensions or geometry references  29-4
storage of in Pro/ENGINEER  29-4
use without Pro/ENGINEER  29-30
when stored as external data  29-4
External data  35-1
classes  35-2
retrieving  35-5
slots  35-2
storing  35-4
External databases
query  31-1
External objects
callbacks  44-14
classes  44-3
color  44-9
creating entities  44-7
data  44-6
manipulating  44-11
display data  44-6
allocating  44-7
properties  44-8
displaying  44-7
identifiers  44-5
information for  44-2
line styles  44-9
parameters  44-5
reusing identifiers  44-4
references  44-12
creating  44-13
types  44-13
visiting  44-13
scale  44-10
selection data  44-10
summary  44-2
transformation  44-8
types  44-5
visiting  44-6
warning mechanism  44-17
External references
  defined 45-4

F

Face milling
  required parameters 30-32
Faces
  definition Glossary-3
  evaluating 9-15
  traversal F-2
Family tables
  editing 34-2
  instances
    creating 34-5
    erasing 34-5
    generic 34-5
    locks 34-5
    operations 34-4
    retrieving 34-5
  items
    from model items 34-6
    from parameters 34-6
    operations on 34-6
  objects 34-2
  showing 34-2
  utilities 34-2
  visiting 34-3
  writing to a file 34-6
FEA mesh 43-3
Feature
  extruded
    creating 25-8
Feature element tree
  datum axis 17-50
  datum coordinate system 17-62
  datum plane 17-2
  datum point 17-12
  field datum point 17-20
  general datum point 17-32
  offset csys datum point 17-24
  sketched datum point 17-12
Features
  chamfers 19-2
  children 12-5
  creating 16-1
  assembly components by 22-13
  code example 16-17
  datums 16-17
  steps 16-2
  datum axis 17-50
  datum coordinate system 17-62
  datum plane 17-2
  datum point 17-12
  deleting 12-6, 13-14
  dimensions 12-10
  element table 16-5
  element tree 16-5
  element values 16-5
  elements
    creating 16-13
    manipulating 16-13
    paths 16-11
    values 16-9
  geometry 12-6
  visiting 9-5
  header files for 16-6
  incomplete 16-16
  description 12-4
  inquiry 12-3
  detailed explanation 16-22
  manipulating 12-6
  manufacturing
    analyzing 30-38
    creating 30-21
  objects 12-2
  parents 12-5
  patterns 12-10
    creating 21-2
  process steps 33-2
  redefining 12-6
  reordering 12-8
  selecting 12-5
  status 12-4
  suppressing 12-6
  sweeps 26-1
  tweak surface replacement 28-1
  types 12-3
  user-defined 12-18
  visiting 12-2
  workcell
    identifying 30-6
FEM
  exporting an FEA mesh 43-3
  getting bar elements 43-11
  getting constraint case names 43-5
  getting constraints 43-8
  getting contacts 43-11
  getting mass elements 43-12
  getting mesh controls 43-5
  getting runners 43-11
  getting shell pairs 43-13
  overview of functionality 43-2
Files
  checking out 47-2
  connector parameters 42-10
  displaying 49-4
  editing 49-4

Index - ix
include 1-39
management 6-6
material 7-12
maximum length 4-26
menu 4-25
message 3-5
naming restrictions 3-5
opening 49-5
parsing 49-5
trail 49-3
writing a family table to 34-6
Fillet surfaces G-12
Filter functions
for visits 1-56
Find
APIWizard search feature (Find) 1-11
Fixtures
creating 30-21
Fonts 8-18
Foreign datum curves 27-3
continuity 27-6
evaluation function for binding 27-7
evaluation functions for 27-4 example 27-7
Foreign datum surfaces 27-2
importing 27-2
Foreign programs
running B-4
multiple B-5
Frames
display frame in APIWizard 1-7
in the APIWizard online browser 1-7
topic/object selection 1-7
Free
attachments 7-17
external object data 44-11
external object references 44-13
text style structure 7-17
version stamp 6-11
Full asynchronous mode 52-8, 52-10
Function
MenuPane 5-52
Functions
action 1-56
browsing with APIWizard 1-10
collect 1-53
comparing toolkits D-4
error statuses 1-19
filter 1-56
list of equivalent D-16
new with Release Wildfire A-8
prototyping 1-19
superseded A-20
using expandable arrays 1-52
visit 1-56

G

Gather
by distance 38-16
by model name 38-15
by parameters 38-15
by rule 38-14
by simplified representation 38-17
by size 38-16
by zone 38-16
General process steps 33-6
General surface of revolution G-8
Generic instances 34-5
Geometry
cables 42-18
checks 12-5
coordinate system datums 9-22
cross-sectional 40-2
datum axes 9-22
datum curves 9-23
datum planes 9-22
datum points 9-23
datum surfaces 9-22
equations 9-16
evaluating 9-15
feature visiting 9-5
measurement of 9-25
NURBS 9-27
objects 9-2
visiting 9-3
of solid edges 9-18
of surfaces 9-20
representations G-1
solid visiting 9-7
terms F-2
tolerance reference by gtol 15-2
traversal F-1
Graphics
color and line styles 8-10
color map 8-13
displaying 8-14
line styles setting 8-14
surviving a repaint 8-19
Groups 12-18
drawing symbol 39-50
identifying symbol definition 39-51
Identifying symbol instance 39-51
local 12-17
manipulating symbols 39-52
read access 12-18
gtol
  creation 15-7
defined 15-2
deleting 15-14
determining if datum is gtol 15-3
inght 15-13
  backup made by ProGtolEditInit 15-14
cancellation 15-14
live in Pro/ENGINEER session 15-13
save changed data with
  ProGtolEditCommit 15-14
handle
  return 15-2
type 15-2
placement
set 15-8
ProGtolleader defined 15-8
reading 15-4
reference
clear 15-3
creation 15-2
set 15-8
reference to internal data structure of 15-2
required function sequence 15-7, 15-14
return type 15-4
set functions return ProGtolstatus
type 15-8
visiting 15-3
gtoldata
defined 15-2
handle type 15-2

H

handle
  return for gtol 15-2
  return for gtoldata 15-2
type for gtol 15-2
type for gtoldata 15-2
Handles
data 1-48
description 1-46
object 1-46
Pro/DEVELOP versus Pro/TOOLKIT D-7
workspace 1-49
Hardware type
setting 1-61
Harnesses
clearance 42-18
connectors 42-9
Height
text 8-17
Help 54-2
  creating help files 54-3
  environment variables for 54-2
error message 54-3
helpfiles.html 54-2
smhelp.h 54-2
Highlight 2-9
definition Glossary-3
menu buttons 4-41
Highlighting
functions described 2-9
Holemaking
required parameters 30-32
tool parameters 30-12
Holes
code example 1-58
Hot spots 44-10

I

Identifiers
cable 42-14
external object 44-5
postfix 7-9
recycling 44-4
Identifying
symbol groups
definition 39-51
instance 39-51
Identity matrix
used with external objects 44-8
Import
files 50-19
foreign datum surfaces 27-2
Include files 1-39
Incomplete features
description 12-4
Information window
definition Glossary-3
display function 49-4
Initialize
assembly component paths 13-5
family tables 34-2
process step features 33-2
single animation 46-5
tool 30-13
Input
default values 3-9
keyboard 3-9
mouse 8-20
Input files 50-2
Inquiry
features 12-3
Insert mode 12-7
Install
Pro/TOOLKIT 1-20
See Pro/TOOLKIT Installing 1-20
Installation
See Pro/TOOLKIT Installing 1-20

Index - xi
Instances
identifying symbol groups 39-51
Interactive selection 2-5
Interchange domain 13-16
Interface
tools 41-1
Interference 9-28
Internal buffer
writing a message to 3-9
Internet Explorer
Java2 plugin in 1-3, 1-5
requirements 1-3, 1-5
Inverse evaluation 9-16
Invert
matrixes 7-21

J
Java2 plugin
not recommended for APIWizard with
Netscape Navigator 1-6
required for APIWizard with Internet
Explorer 1-3, 1-5 to 1-6

K
Keyboard
input 4-48
default values 3-9
getting 3-9
macros
execution rules 4-45

L
LANG variable 54-2
Launch
Synchronous J-Link Applications 53-9
Toolkit DLL Functions 53-8
Layers 6-12
creating 6-13
deleting 6-13
getting 6-12
items 6-13
view dependency 6-14
visiting 6-14
Layouts 6-17
Leaders
definition in glossary Glossary-3
note placement 7-16
patterns 12-12
Length
units 7-8
License data 49-3
Light sources 8-25
Line
styles
of external objects 44-9
Lines
code example 8-16
drawing 8-15
representation G-19
styles 8-10
setting 8-14
types 8-14
Linking 1-30
Load directory
definition Glossary-3
Local groups 12-17
Locations
of assembly components 13-11
routing 42-17
Locks 34-5

M
Macro keys
definition Glossary-3
main()
user-supplied 1-45
Makefiles
description of 1-30
Management
Argument 53-2
Memory 53-4
Manipulating symbol groups 39-52
Manufacturing
analyzing a model 30-3
components
roles 30-5
traversing 30-5
features
analyzing 30-38
creating 30-21
fixtures 30-21
models
create 30-3
types 30-2
NC sequences 30-30
objects
creating 30-9
operations
creating 30-27
parameters 30-16
roles 30-5
storage solids 30-3
identifying 30-6
tools
creating 30-9
types 30-8
visiting 30-8
workcells
creating 30-22
Mass elements
  getting 43-12
Mass properties 7-8
  definition Glossary-4
  example 40-6
  of cross sections 40-6
Mass units 7-8
Materials
  data
    accessing 7-11
  files 7-12
  properties 7-10
  example 7-12
  removal volumes
    creating 30-34
Matrix
  definition Glossary-4
  inverting 7-21
  pan and zoom 8-6
  view 8-6
  window 8-6
Maximum length
  of files 4-26
Measurement 9-25
Memory
  Management
    Task Library Functions 53-4
    maximum allocated by ProArrayAlloc 1-51
Menu Bars
  pushbutton
    adding 4-4
Menu bars
  adding a menu 4-15
  adding buttons 4-8
  buttons 4-2
  code example 4-16
  menus 4-2
Menus
  adding to menu bar 4-15
  buttons
    accessibility 4-42
    adding 4-28
    example 4-30
    locations 4-40
    setting 4-41
  compound
    creating 4-36
  creating
    exit actions 4-31
    for selecting a single value 4-34
    preempting existing commands 4-37
  data menus 4-40
  definition Glossary-4
  exit actions 4-31
  files
    definition Glossary-4
    names and contents 4-26
    purpose 4-25
    sample 4-27
    submenus 4-39
    syntax 4-26
    toolkit D-7
  main
    definition Glossary-3
  manipulating 4-39
    accessibility of buttons 4-42
    data menus 4-40
    setting buttons 4-41
  mode-specific 4-25
  new
    creating 4-30
    defining 4-31
    using 4-33
    pushing and popping 4-42
    run-time 4-43
    submenus 4-39
    definition Glossary-6
Merge 13-15
Mesh controls
  getting 43-5
Message
  classification of 3-7
Message file
  definition Glossary-4
  restrictions 3-5
Messages
  classification of
    critical 3-8
    error 3-8
    info 3-8
    prompt 3-7
    warning 3-8
  file 3-5
  files
    toolkit D-7
Milling tools
  parameters 30-11
Model coordinates
  definition Glossary-4
Model items
  definition Glossary-4
  description 6-7
  from family table items 34-6
  names 6-9
Model units 7-7
ModelCHECK 49-7
  running 49-7
Models 6-2
definition Glossary-4
drawing 39-14
file management operations 6-6
finding the handle
  example 6-5
identifying 6-4
in session 6-5
names
  gathering by 38-15
  orientation 8-6
section 23-2
Modes 1-28
  complement
    definition Glossary-1
    description 6-2
    set
    definition Glossary-6
Mode-specific menus 4-25
Modify
colors
  example 8-13
cross sections 40-5
  simplified representations 38-13
Mouse
  input 8-20
  positions
    example 8-21
Movies
  animation 46-5
Multiprocess mode
  overview 1-28
  registry file for 1-33
Multivalued element 16-5

N

Names
  of foreign datum surfaces 27-2
  simplified representations 38-9
Naming conventions 1-17
NC sequences
  auxiliary and custom 32-2
  creating 30-30
  determining 12-4
Netscape Navigator
  requirements 1-3
  Swing required 1-3
Notes 7-15
  attachments 7-16
  code example 7-18
  detail items 39-20
  displaying 7-16
  leaders 7-16
  owners 7-16
  placement 7-16
  placement information 7-17
text 7-15
text styles 7-17 to 7-18
URLs 7-17
visiting 7-16
Notify 36-1
  classes 36-2
  definition Glossary-5
NURBS 9-27
  representation G-22
  surface G-14

O

Object
  ProDtlitem 39-21
Object handle
  ProDtlentity 39-21
Object handles
  ProDtlentity 39-21
  ProDtlgroup 39-21
  ProDtlnote 39-21
  ProDtlvmdel 39-21
  ProDtslninst 39-21
ObjectName
  ProPatternParam 11-2
Objects
  browsing with APIWizard 1-8
  definition Glossary-5
  displaying 8-10
  external 44-2
  handles 1-46
  list of E-2
  naming conventions 1-17
  ProSelection
    defined 2-2
    selecting 2-2
  OHandles
    converting to DHandles 1-49
    description 1-47
  ProView 39-10
Opaque pointers 1-47
Operations
  creating 30-27
  example 30-28
Options
  configuration file
    getting and setting 49-2
Orientation
  model 8-6
Outlines
  code example 7-6
  of solids 7-5
Output files 50-2
Overlay views
  definition Glossary-5
Owners
include files  1-39
Installation  1-23
installing  1-20
add or update  1-23
loadpoint directories  1-20
test of  1-24
version compatibility  1-33
models  6-2
new functions for Release 20.0  A-2
objects  1-17
registry file
  compared to Pro/DEVELOP  D-7
terminology
  contrasted with Pro/DEVELOP  D-4
using a batch session  1-42
utility functions  49-1
Pro/Toolkit DLL Functions
  Launch  53-8
pro_change_dir() function
  equivalent  D-21
pro_clear_window() function
  equivalent  D-18
pro_close_object_window() function
  equivalent  D-18
pro_compute_clearance() function  9-28
assemble members  13-15
  used in a code example  9-28
pro_compute_global_interference() function
  description  9-28
  used in a code example  9-31
pro_compute_interference() function
  description  9-28
pro_compute_volume() function
  used in a code example  9-28
pro_create_2d_disp_list() function
  equivalent  D-19
pro_create_3d_disp_list() function
  description  D-19
pro_delete_2d_disp_list() function
  description  D-19
pro_delete_3d_disp_list() function
  equivalent  D-19
pro_display_2d_disp_list() function
  description  D-19
pro_display_3d_disp_list() function
  description  D-19
pro_display_interf_volume() function
  used in a code example  9-28
pro_dist_manifolds() function
  description  9-28
  used in a code example  9-29
PRO_DRAWING_WELD_GROUPIDS_GET
  weld symbol notification type  37-3
PRO_DRAWING_WELD_SYMPATH_GET
  weld symbol notification type  37-3
PRO_DRAWING_WELD_SYMTEXT_GET
  weld symbol notification type  37-3
PRO_E_CURVE_CONTINUITY element  27-6
pro_edge_length() function
  equivalent  D-24
pro_element_length() function
  equivalent  D-21
pro_eval_uv_edge() function
  equivalent  D-23
pro_eval_xyz_edge() function
  equivalent  D-23
pro_eval_xyz_edge() function
  equivalent  D-23
pro_eval_xyz_face() function
  equivalent  D-23
pro_eval_xyz_face() function
  description  43-3
pro_export_fea_mesh() function
  description  43-3
pro_export_file_from_pro() function
  equivalent  D-22
pro_export_plot_file() function
  equivalent  D-22
pro_face_area() function
  equivalent  D-24
pro_face_extremes() function
  equivalent  D-24
PRO_FEAT_ASSEM_CUT  13-2
PRO_FEAT_COMPONENT  13-2
Pro_fem_bar_element structure  43-11
Pro_fem_con_value structure  43-10
Pro_fem_constraint structure  43-8
Pro_fem_contact structure  43-11
Pro_fem_mass_element structure  43-12
Pro_fem_mass_props structure  43-13
Pro_fem_region structure  43-7
Pro_fem_runner structure  43-11
Pro_fem_shell_pair structure  43-13
pro_get_config() function
  equivalent  D-22
pro_get_cur_window_matrix() function
  equivalent  D-18
pro_get_current_directory() function
  equivalent  D-21
pro_get_current_mode() function
  equivalent  D-20
pro_get_current_objects() function
  equivalent  D-20
pro_get_current_window() function
  equivalent  D-18
pro_get_curve_type_geom() function
  equivalent  D-23
pro_get_edge_params() function
  equivalent  D-23
pro_get_entity_params() function
  equivalent  D-23
pro_get_face_params() function
  equivalent  D-23

Index - xvi
<table>
<thead>
<tr>
<th>Function/Structure/Enum</th>
<th>Reference</th>
<th>Function/Structure/Enum</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>pro_get_light_sources() function</td>
<td>D-18</td>
<td>pro_rotate_view() function</td>
<td>D-18</td>
</tr>
<tr>
<td>description</td>
<td>8-25</td>
<td>pro_sample_xy() function</td>
<td>D-18</td>
</tr>
<tr>
<td>pro_get_prodevdat_info() function</td>
<td>D-21</td>
<td>pro_select() function</td>
<td>D-19</td>
</tr>
<tr>
<td>equivalent</td>
<td>D-21</td>
<td>guidelines</td>
<td>D-15</td>
</tr>
<tr>
<td>pro_get_selection() function</td>
<td>D-19</td>
<td>options</td>
<td>D-14</td>
</tr>
<tr>
<td>guidelines</td>
<td>D-15</td>
<td>pro_select_strings() function</td>
<td>D-17</td>
</tr>
<tr>
<td>pro_get_text_attributes() function</td>
<td>D-18</td>
<td>equivalent</td>
<td>D-21</td>
</tr>
<tr>
<td>equivalent</td>
<td>D-18</td>
<td>guidelines</td>
<td>D-15</td>
</tr>
<tr>
<td>pro_get_view_matrix() function</td>
<td>D-18</td>
<td>pro_set_and_get_selection() function</td>
<td>D-21</td>
</tr>
<tr>
<td>equivalent</td>
<td>D-18</td>
<td>equivalent</td>
<td>D-21</td>
</tr>
<tr>
<td>pro_get_view_names() function</td>
<td>D-18</td>
<td>guidelines</td>
<td>D-15</td>
</tr>
<tr>
<td>equivalent</td>
<td>D-21</td>
<td>pro_set_config() function</td>
<td>D-22</td>
</tr>
<tr>
<td>pro_getbox() function</td>
<td>D-19</td>
<td>pro_set_current_window() function</td>
<td>D-18</td>
</tr>
<tr>
<td>equivalent</td>
<td>D-21</td>
<td>pro_set_light_sources() function</td>
<td>D-18</td>
</tr>
<tr>
<td>pro_getenvironment() function</td>
<td>D-21</td>
<td>pro_set_view_matrix() function</td>
<td>D-18</td>
</tr>
<tr>
<td>pro_interference_volume_release() function</td>
<td>9-28</td>
<td>pro_show_file() function</td>
<td>D-21</td>
</tr>
<tr>
<td>used in a code example</td>
<td>9-30</td>
<td>pro_show_info_window() function</td>
<td>D-21</td>
</tr>
<tr>
<td>pro_is_option_ordered() function</td>
<td>D-21</td>
<td>pro_show_select() function</td>
<td>D-18</td>
</tr>
<tr>
<td>equivalent</td>
<td>D-21</td>
<td>guidelines</td>
<td>D-15</td>
</tr>
<tr>
<td>Pro_light structure</td>
<td>8-25</td>
<td>pro_solid_id_from_ptr() function</td>
<td>D-20</td>
</tr>
<tr>
<td>Pro_mesh_data structure</td>
<td>43-3</td>
<td>pro_solid_ptr_from_id() function</td>
<td>D-20</td>
</tr>
<tr>
<td>pro_on_regenerate_end() function</td>
<td>D-26</td>
<td>pro_store_view() function</td>
<td>D-21</td>
</tr>
<tr>
<td>equivalent</td>
<td>D-18</td>
<td>pro_str_to_wstr() function</td>
<td>D-21</td>
</tr>
<tr>
<td>pro_open_object_window() function</td>
<td>D-18</td>
<td>Pro2dExport function</td>
<td>D-16</td>
</tr>
<tr>
<td>equivalent</td>
<td>D-18</td>
<td>Pro2dImportAppend function</td>
<td>D-21</td>
</tr>
<tr>
<td>Pro_Pattern structure</td>
<td>12-13</td>
<td>Pro2dImportCreate function</td>
<td>D-21</td>
</tr>
<tr>
<td>declaration</td>
<td>12-15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pro_pattern_dim structure</td>
<td>12-15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>declaration</td>
<td>12-14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pro_pattern_dir structure</td>
<td>12-13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>declaration</td>
<td>12-13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pro_pattern_type enum</td>
<td>12-13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>declaration</td>
<td>12-13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pro_point_on_geom() function</td>
<td>D-23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>equivalent</td>
<td>D-23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pro_ray_x_model() function</td>
<td>D-21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>equivalent</td>
<td>D-21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pro_read_file_to_pro() function</td>
<td>D-22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>equivalent</td>
<td>D-22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pro_refresh_window() function</td>
<td>D-18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>equivalent</td>
<td>D-18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pro_regenerate() function</td>
<td>D-21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>equivalent</td>
<td>D-21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pro_regenerate_object() function</td>
<td>D-21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>and pattern tables</td>
<td>12-17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>equivalent</td>
<td>D-21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pro_release_memb_id_tab() function</td>
<td>42-9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>description</td>
<td>42-9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pro_reset_view() function</td>
<td>D-18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>equivalent</td>
<td>D-18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pro_retrieve_view() function</td>
<td>D-18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pro_str_to_wstr() function</td>
<td>D-21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>equivalent</td>
<td>D-21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pro2dExport function</td>
<td>D-21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>defined 50-4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pro2dImportAppend function</td>
<td>defined 50-21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>defined 50-21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pro2dImportCreate function</td>
<td>defined 50-21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>defined 50-21</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Index - xvii
ProAnalysisAttrIsSet() function
defined 29-29
ProAnalysisAttrSet() function
defined 29-29
ProAnalysisInfoGet function
defined 29-8
ProAnalysisInfoSet function
defined 29-8
ProAnalysisNameGet function
defined 29-29
ProAnalysisTypeGet function
defined 29-8
ProAnalysisTypeRegister function
call from user initialize 29-5
defined 29-5
ProAnimframeCreate() function
description 46-4
used in a code example 46-8
ProAnimframeDelete() function
description 46-4
ProAnimframeObjAdd() function
description 46-4
used in a code example 46-8
ProAnimframeObjRemove() function
description 46-4
ProAnimframeObjVisit() function
description 46-4
ProAnimmovieCreate() function
description 46-5
used in a code example 46-8
ProAnimmovieDelete() function
description 46-5
ProAnimmovieFrameAdd() function
description 46-5
used in a code example 46-8
ProAnimmovieFrameRemove() function
description 46-5
ProAnimmovieFrameVisit() function
description 46-6
ProAnimobjectCreate() function
description 46-3
used in a code example 46-8
ProAnimobjectDelete() function
description 46-3
ProAppData data type
declaration 1-57
proappdata_create_data() function
equivalent D-22
proappdata_delete_data() function
equivalent D-22
proappdata_list_classes() function
equivalent D-22
proappdata_list_data_in_class() function
equivalent D-22
proappdata_read_data() function
equivalent D-22
proappdata_register_class() function
equivalent D-22
proappdata_write_data() function
equivalent D-22
ProArcdataInit() function
description 51-10
ProArgument
Argument Management 53-2
description 53-2
ProArgumentByLabelGet() function
description 53-2
ProArgumentProarrayFree() function
description 53-3
ProArrayAlloc
maximum allocated memory 1-51
ProArrayAlloc() function
description 1-51
with element paths 16-12
ProArrayFree() function
description 1-51
ProArrayObjectAdd() function
description 1-52
ProArrayObjectRemove() function
description 1-52
ProArraySizeGet() function
description 1-51
ProArraySizeSet() function
description 1-52
ProAsmcomp object
description 13-2
ProAsmcompAssemble() function
description 22-2
ProAsmcompconstraintArrayAllocate() function
description 22-3
ProAsmcompconstraintArrayFree() function
description 22-6
ProAsmcompconstraintAsmreferenceGet() function
description 22-5
ProAsmcompconstraintAsmreferenceSet() function
description 22-5
ProAsmcompconstraintAttributesGet() function
description 22-6
ProAsmcompconstraintAttributesSet() function
description 22-6
ProAsmcompconstraintCompreferenceGet() function
description 22-5
ProAsmcompconstraintCompreferenceSet() function
description 22-5
ProAsmcompconstraintOffsetGet() function
description 22-6
Pro/TOOLKIT User’s Guide
ProCableSectionsDelete() function description 42-20
ProCablesFromLogicalCreate() function description 42-14
procb_cable_clearance() function description 42-18
procb_designate_connector() function description 42-9
procb_get_assy_connectors() function description 42-12
procb_get_cable_geom() function description 42-18
procb_get_cable_id() function description 42-14
procb_get_cable_logical_ends() function description 42-15
procb_get_cable_subtype() function description 42-14
procb_get_connector_entry_ports() function description 42-10
procb_get_connector_parameters() function description 42-10
procb_get_harness_locations() function description 42-17
procb_get_location_cables() function description 42-17
procb_get_location_pnt() function description 42-17
procb_get_wire_length() function description 42-18
procb_route_thru_location() function description 42-19
procb_routing_end() function description 42-19
procb_undesignate_connector() function description 42-12

Process steps
access 33-2
creating 33-3
feature elements 33-4
optional elements 33-6
types of 33-5
visiting 33-2

ProClcmdElemAdd() function description 32-4
ProClcmdElemCreate() function description 32-4
ProClcmdElemGet() function description 32-4
ProClcmdElemGetByIndex() function description 32-4
ProClcmdElemSet() function description 32-4

ProClcmdElemRemove() function description 32-4
ProClcmdElemRemoveByIndex() function description 32-4
ProClcmdElemSet() function description 32-4

ProClCommandExpandAction callback 36-7 to 36-8
ProClCommandGetLocAction callback 36-7

ProCmdActionAdd() function description 4-5, 4-20
used in a code example 4-17
ProCmdDesignate() function description 4-21
ProCmdIconSet() function description 4-21
ProCmdOptionAdd() function description 4-20
ProColorByTypeGet 8-12
ProColormapGet() function description 8-13
used in a code example 8-13
ProColormapSet() function description 8-13
used in a code example 8-14
ProColorTextModify() function description 8-11
ProCompoundmenuModify() function description 4-36

ProConfigoptArrayCreate() function description 49-2
ProConfigoptSet() function description 49-2
ProContourdataAlloc() function description 51-7
ProContourdataEdgeIdArraySet() function description 51-8
ProContourdataInit() function description 51-7
ProContourdataInit() function description 51-7
ProContourEdgeVisit() function description 9-7
ProContourTraversalGet() function description 9-7
ProContourUvpntVerify() function description 9-16
ProCsyst object declaration 1-47
ProCsystCreate() function description 9-22, 10-8
ProCsystSet() function description 9-11
ProCsystGet() function description 9-11
ProCurrentWorkspaceGet() function description 47-2

Index - xx
used in a code example 47-3

ProCurve object
declaration 1-47
description 9-12

ProCurvedata objects
used for external objects 44-7

ProCurvedataAlloc() function
defined 39-24
description 51-9

ProCurvedataFree function
description 51-9

ProCurveDataGet() function
description 9-23

ProCurveIdGet() function
description 9-12

ProCurveInit() function
description 9-13

ProCurveLengthEval() function
description 9-26

ProCurveLengthT1T2Eval() function
description 9-26

ProCurveParamByLengthEval() function
description 9-26

ProCurveParamEval() function
description 9-16

ProCurveTessellationGet function
description 9-14

ProCurveToNURBS() function
description 9-27

ProCurveTypeGet() function
description 9-23

ProDatumdataAlloc() function
description 51-12

ProDatumdataInit() function
description 9-16

ProDatumdataxyzdataEval() function
description 9-16

prodb_add_parameters() function
equivalent D-24
prodb_assemble_by_transform() function
equivalent D-26
prodb_assemble_component() function
equivalent D-26
prodb_auto_interchange() function
description 13-16
prodb_backup_object() function
equivalent D-20
prodb_compute_outline() function
equivalent D-24
prodb_containing_contour() function
equivalent D-23
prodb_contour_traversal() function
equivalent D-23
prodb_copy_object() function
equivalent D-20
prodb_create_obj() function
equivalent D-20
prodb_create_object() function
equivalent D-20
prodb_create_parallel_xsec() function
used in a code example 40-5
prodb_declare_layout() function 6-17
prodb_delete_feature() function
equivalent D-25
prodb_delete_parameters() function
equivalent D-24
prodb_delete_xsec() function
description 40-6
used in a code example 40-9
prodb_designate_param() function
equivalent D-24
prodb_edge_data() function
equivalent D-23
prodb_edge_direction() function
equivalent D-23
prodb_edge_tessellation() function
equivalent D-23
prodb_edge_to_nurbs() function
equivalent D-23
prodb_edit_file() function
equivalent D-21
prodb_entity_to_nurbs() function
equivalent D-23
prodb_erase_object() function
equivalent D-20
prodb_eval_xyz_dtm_point() function
equivalent D-25
prodb_explode_assembly() function
equivalent D-26
prodb_find_csys() function
equivalent D-25
prodb_find_declared_objects() function
equivalent D-20
prodb_find_nobject_depend() function
equivalent D-20
prodb_first_cntr_edge() function
equivalent D-22
prodb_first_datum() function
equivalent D-22
prodb_first_face_contour() function
equivalent D-22
prodb_first_member() function
equivalent D-25
prodb_first_name_in_list() function
equivalent D-20
prodb_first_part_face() function
equivalent D-22
prodb_first_xsec() function
description 40-2
used in a code example 40-3
prodb_get_asm_constraints() function
equivalent D-26

Index - xxi
prodb_get_asm_transform() function  
  equivalent D-26
prodb_get_comp_role() function  
  equivalent D-27
prodb_get_curve_type() function  
  equivalent D-23
prodb_get_datum_curves() function  
  equivalent D-22
prodb_get_datum_surfaces() function  
  equivalent D-22
prodb_get_dwg_fmt_name() function  
  39-7
prodb_get_edge_curve() function  
  equivalent D-23
prodb_get_edge_type() function  
  equivalent D-23
prodb_get_edge_uv_points() function  
  equivalent D-23
prodb_get_element_name() function  
  equivalent D-24
prodb_get_envelope() function  
  equivalent D-24
prodb_get_face_type() function  
  equivalent D-23
prodb_get_feat_parent_child() function  
  equivalent D-24
prodb_get_feature_dtm_points() function  
  equivalent D-25
prodb_get_feature_ids() function  
  equivalent D-24
prodb_get_feature_surfaces() function  
  equivalent D-24
prodb_get_first_axis() function  
  equivalent D-25
prodb_get_material_name() function  
  equivalent D-22
prodb_get_material_props() function  
  equivalent D-22
prodb_get_model_accuracy() function  
  equivalent D-21
prodb_get_model_units() function  
  description 7-7
prodb_get_next_axis() function  
  equivalent D-25
prodb_get_object_info() function  
  used in a cross section example 40-5
prodb_get_object_info() function  
  equivalent D-20
prodb_get_parameters() function  
  equivalent D-20
prodb_get_simplfd_rep_info() function  
  equivalent D-20
prodb_get_simplfd_rep_list() function  
  equivalent D-20
prodb_get_solid_surfaces() function  
  equivalent D-22
prodb_get_solid_volumes() function  
  equivalent D-23
prodb_get_surface() function  
  equivalent D-23
prodb_get_surface_feature() function  
  equivalent D-24
prodb_get_surface_quilt() function  
  equivalent D-25
prodb_interchange_domain() function  
  description 13-16
prodb_is_asm_exploded() function  
  equivalent D-26
prodb_mass_prop() function  
  equivalent D-21
prodb_mass_prop_xsec() function  
  used in a code example 40-9
prodb_measure() function  
  equivalent D-24
prodb_member_to_object() function  
  equivalent D-25
prodb_merge_members() function  
  description 13-15
prodb_next_cntr_edge() function  
  equivalent D-22
prodb_next_datum() function  
  equivalent D-25
prodb_next_face_contour() function  
  equivalent D-22
prodb_next_member() function  
  equivalent D-25
prodb_next_name_in_list() function  
  equivalent D-20
prodb_next_part_face() function  
  equivalent D-22
prodb_next_xsec() function  
  description 40-2
  used in a code example 40-3
prodb_obj_from_assem_rel() function  
  equivalent D-25
prodb_regen_xsec() function  
  to modify cross sections 40-6
  used in a code example 40-4
prodb_rename_object() function  
  equivalent D-24
prodb_resume_feature() function  
  equivalent D-25
prodb_resume_feature() function  
  equivalent D-25
Index - xxiii

prodb_retrieve_object() function
  equivalent D-20
prodb_retrieve_simplfd_rep() function
  equivalent D-20
prodb_rls_edge_curve() function
  equivalent D-23
prodb_rls_surface() function
  equivalent D-23
prodb_save_object() function
  equivalent D-20
prodb_set_dwg_fmt() function
  39-7
prodb_set_element_name() function
  equivalent D-24
  used with diagrams 42-2
prodb_set_material_name() function
  equivalent D-22
prodb_set_member_transform() function
  equivalent D-26
prodb_set_model_accuracy() function
  equivalent D-21
prodb_set_model_units() function
  description 7-7
prodb_set_parameters() function
  equivalent D-24
prodb_set_surface_props() function
  description 8-22
prodb_surface_tessellation() function
  description 9-14
  surface properties 8-24, 9-14
prodb_surface_to_nurbs() function
  equivalent D-23
prodb_undeclare_layout() function
  6-17
prodb_unset_surface_props() function
  description 8-22
prodb_uv_in_face_domain() function
  equivalent D-23
prodb_vertex_data() function
  equivalent D-23
prodb_was_object_modified() function
  equivalent D-20
prodb_xsec_component() function
  used in a code example 40-4
prodb_xsec_name() function
  description 40-2
  used in a code example 40-4
prodev_get_proe_status() function
  description 40-6
prodev_set_interrupt_func() function
  equivalent D-16
prodev_set_proe_term_func() function
  equivalent D-16
prodev_start_proengineer() function
  equivalent D-16
prodgm_get_ids() function
  description 42-2
prodgm_get_node_connections() function
  used in a code example 42-6
prodgm_get_nodes() function
  description 42-3
prodgm_get_parameters() function
  description 42-2
prodgm_get_wire_nodes() function
  description 42-6
prodgm_get_wire_spool() function
  description 42-5
Prodgm_item structure 42-3
Prodgm_node_ref structure 42-6
prodgm_set_parameters() function
  description 42-4
prodgm_set_wire_spool() function
  description 42-5
prodim_drawing_dim_view_id() function
  39-13
prodim_set_dim() function
  used with cross-sections 40-6
ProDimModifyValuePreAction callback 36-6
ProDirectoryChange() function
  description 49-4
ProDirectoryChangePostAction callback 36-7
ProDirectoryChoose() function
  description 49-5
ProDispdatAlloc() function
  description 44-7
  used in a code example 44-21
ProDispdatColorGet() function
  description 44-9
ProDispdatColorSet() function
  description 44-9
  used in a code example 44-21
ProDispdatEntsGet() function
  description 44-7
ProDispdatEntsSet() function
  description 44-9
  used in a code example 44-21
ProDispdatGet() function
  description 44-7
ProDispdatLinestyleGet() function
  description 44-9
ProDispdatLinestyleSet() function
  description 44-9
  used in a code example 44-22
ProDispdatPropsGet() function
  description 44-9
  used in a code example 44-21
ProDispdatPropsSet() function
  description 44-9
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProDispdatPropsSet()</td>
<td>44-8</td>
</tr>
<tr>
<td>ProDispdatScaleGet()</td>
<td>44-10</td>
</tr>
<tr>
<td>ProDispdatScaleSet()</td>
<td>44-10</td>
</tr>
<tr>
<td>ProDispdatTrfGet()</td>
<td>44-8</td>
</tr>
<tr>
<td>ProDispdatTrfSet()</td>
<td>44-8</td>
</tr>
<tr>
<td>ProDispList2DCreate()</td>
<td>8-18</td>
</tr>
<tr>
<td>ProDispList2DElete()</td>
<td>8-18</td>
</tr>
<tr>
<td>ProDispList2DDisplay()</td>
<td>8-18</td>
</tr>
<tr>
<td>ProDispList3DCreate()</td>
<td>8-18</td>
</tr>
<tr>
<td>ProDispList3DElete()</td>
<td>8-18</td>
</tr>
<tr>
<td>ProDispList3DDisplay()</td>
<td>8-18</td>
</tr>
<tr>
<td>ProDispListInvalidate()</td>
<td>49-2</td>
</tr>
<tr>
<td>ProDltsymgroupLevelIsExclusive()</td>
<td>39-52</td>
</tr>
<tr>
<td>ProDrawingAsmsimprepAdd()</td>
<td>39-14</td>
</tr>
<tr>
<td>ProDrawingAsmsimprepDelete()</td>
<td>39-14</td>
</tr>
<tr>
<td>ProDrawingAsmsimprepDisplay()</td>
<td>39-14</td>
</tr>
<tr>
<td>ProDrawingCurrentSheetGet()</td>
<td>39-8</td>
</tr>
<tr>
<td>ProDrawingCurrentSheetSet()</td>
<td>39-8</td>
</tr>
<tr>
<td>ProDrawingDimCreate()</td>
<td>39-75</td>
</tr>
<tr>
<td>ProDrawingDtlentitiesCollect()</td>
<td>39-21</td>
</tr>
<tr>
<td>ProDrawingDtlentityVisit()</td>
<td>39-21</td>
</tr>
<tr>
<td>ProDrawingDtlgroupsCollect()</td>
<td>39-21</td>
</tr>
<tr>
<td>ProDrawingDtlgroupVisit()</td>
<td>39-21</td>
</tr>
<tr>
<td>ProDrawingDtlnotesCollect()</td>
<td>39-21</td>
</tr>
<tr>
<td>ProDrawingDtlnoteVisit()</td>
<td>39-21</td>
</tr>
<tr>
<td>ProDrawingDtlNameGet()</td>
<td>39-21</td>
</tr>
<tr>
<td>ProDrawingDtlNameSet()</td>
<td>39-21</td>
</tr>
<tr>
<td>ProDrawingDtlSymdefRetrieve()</td>
<td>39-46</td>
</tr>
<tr>
<td>ProDrawingDtlSymdefsCollect()</td>
<td>39-46</td>
</tr>
<tr>
<td>ProDrawingDtlSymdefVisit()</td>
<td>39-21</td>
</tr>
<tr>
<td>ProDrawingDtlSyminstCollect()</td>
<td>39-21</td>
</tr>
<tr>
<td>ProDrawingDtlSyminstVisit()</td>
<td>39-21</td>
</tr>
<tr>
<td>ProDrawingFormatAdd()</td>
<td>39-8</td>
</tr>
<tr>
<td>ProDrawingFormatGet()</td>
<td>39-8</td>
</tr>
<tr>
<td>ProDrawingPosToLocgrid()</td>
<td>39-8</td>
</tr>
<tr>
<td>ProDrawingScaleGet()</td>
<td>39-6</td>
</tr>
<tr>
<td>ProDrawingScaleSet()</td>
<td>39-11</td>
</tr>
<tr>
<td>ProDrawingSheetCreate()</td>
<td>10-7</td>
</tr>
<tr>
<td>ProDrawingSheetDelete()</td>
<td>39-8</td>
</tr>
<tr>
<td>ProDrawingSheetsCount()</td>
<td>39-7</td>
</tr>
<tr>
<td>ProDrawingSheetsReorder()</td>
<td>39-8</td>
</tr>
<tr>
<td>ProDrawingSheetTrfGet()</td>
<td>10-7, 39-7</td>
</tr>
<tr>
<td>ProDrawingSheetTrfSet()</td>
<td>39-8</td>
</tr>
<tr>
<td>ProDrawingSimPrepsCollect()</td>
<td>39-14</td>
</tr>
<tr>
<td>ProDrawingSolidAdd()</td>
<td>39-14</td>
</tr>
<tr>
<td>ProDrawingSolidDelete()</td>
<td>39-14</td>
</tr>
<tr>
<td>ProDrawingSolidReplace()</td>
<td>39-14</td>
</tr>
<tr>
<td>ProDrawingSolidsCollect()</td>
<td>39-14</td>
</tr>
<tr>
<td>ProDrawingSolidsVisit()</td>
<td>39-14</td>
</tr>
<tr>
<td>ProDrawingSystemDtlSymdefRetrieve()</td>
<td>39-46</td>
</tr>
<tr>
<td>ProDrawingViewCollect()</td>
<td>39-10</td>
</tr>
<tr>
<td>ProDrawingViewDelete()</td>
<td>39-13</td>
</tr>
<tr>
<td>ProDrawingViewDisplayGet()</td>
<td>39-11</td>
</tr>
<tr>
<td>ProDrawingViewIsOverlay()</td>
<td>39-19</td>
</tr>
<tr>
<td>ProDrawingViewMove()</td>
<td>39-13</td>
</tr>
<tr>
<td>ProDrawingViewNameGet()</td>
<td>39-11</td>
</tr>
<tr>
<td>ProDrawingViewOutlineGet()</td>
<td>39-10</td>
</tr>
<tr>
<td>ProDrawingViewScaleGet()</td>
<td>39-11</td>
</tr>
<tr>
<td>ProDrawingViewScaleIsUserDefined()</td>
<td>39-11</td>
</tr>
<tr>
<td>ProDrawingViewScaleSet()</td>
<td>39-13</td>
</tr>
</tbody>
</table>
ProDrawingViewSheetGet() function
description 39-10
ProDrawingViewSolidGet() function
description 39-11
ProDrawingViewTransformGet() function
description 39-11
ProDrawingViewVisit() function
description 39-10
ProDrawingViewZclippingGet function
description 39-14
prodrw_add_sheet() function
description 39-7
prodrw_delete_sheet() function
description 39-7
prodrw_get_current_sheet() function 39-7
prodrw_get_overlay_view_id() function 39-19
prodrw_set_current_sheet() function 39-7

ProDtl_attach structure 39-55
prodtl_create() function
detail items 39-20
prodtl_delete() function 39-20
prodtl_display() function 39-20
prodtl_get_ids() function 39-20
prodtl_get_item() function 39-20

ProDtlentityDataGet() function
description 39-23
ProDtlentitydataHiddenSet() function
description 39-28
ProDtlentitydataIsConstruction() function
description 39-28
ProDtlentitydataIsHidden() function
description 39-28
ProDtlentitydataViewSet() function
defined 39-27
ProDtlentitydataZclippingGet() function
defined 39-27

prodtl_modify() function 39-20
prodtl_dataColorGet() function 39-27
prodtl_dataColorSet() function 39-27
prodtl_dataConstructionSet() function
description 39-28
prodtl_dataCurveGet() function 39-27
prodtl_dataCurveSet() function 39-27
prodtl_dataFontGet() function 39-27
prodtl_dataFontSet() function 39-27
prodtl_dataWidthGet() function 39-27
prodtl_dataWidthSet() function 39-27

ProDtlgroup
object handle 39-21

ProDtlgroupAttach() function
description 39-33
ProDtlgroupAttachFree() function
description 39-33
ProDtlgroupAttachGet() function
description 39-33
ProDtlgroupAttachModify() function
description 39-33
ProDtlgroupAttachCreate() function
description 39-33

ProDtlgroupdataDisplayedSet() function
description 39-53
ProDtlgroupdataGet() function
description 39-33
ProDtlgroupdataIsDisplayed() function
description 39-53
ProDtlgroupdataCanDelete() function
description 39-23
ProDtlgroupdataModify() function
description 39-23
ProDtlgroupdataCreate() function
description 39-23

ProDtlitem
object 39-21

ProDtlnote
object handle 39-21

ProDtlnotedataAlloc() function 39-34
ProDtlnotedataAngleGet() function 39-36
ProDtlnotedataAngleSet() function 39-36
ProDtlnotedataAttachmentSet() function
defined 39-35
ProDtlnotedataColorGet() function
defined 39-35
ProDtlnotedataColorSet() function
defined 39-35
ProDtlnotedataDisplayedSet() function  
  description  39-36

ProDtlnotedataElbowlengthGet() function  
  defined  39-35

ProDtlnotedataElbowlengthSet() function  
  defined  39-35

ProDtlnotedataFree() function  
  defined  39-34

ProDtlnotedataGet() function  
  defined  39-24
  description  39-23

ProDtlnotedataIdGet() function  
  defined  39-35

ProDtlnotedataIsDisplayed() function  
  description  39-36

ProDtlnotedataIsMirrored() function  
  defined  39-35

ProDtlnotedataJustifGet() function  
  defined  39-36

ProDtlnotedataJustifSet() function  
  defined  39-36

ProDtlnotedataLeaderAdd() function  
  defined  39-35

ProDtlnotedataLeadersCollect() function  
  defined  39-35

ProDtlnotedataLinesCollect() function  
  defined  39-35

ProDtlnotedataLinesSet() function  
  defined  39-35

ProDtlnotedataLineAdd() function  
  defined  39-35

ProDtlnotedataLinesCollect() function  
  defined  39-35

ProDtlnotedataLinesSet() function  
  defined  39-35

ProDtlnotedataMirrorSet() function  
  defined  39-35

ProDtlnoteDelete() function  
  description  39-23

ProDtlnoteDraw() function  
  description  39-22

ProDtlnoteErase() function  
  description  39-22

ProDtlnotedataLinesSet() function  
  defined  39-35

ProDtlnotedataLineEnvelopeGet() function  
  described  39-23

ProDtlnotedataLinesSet() function  
  described  39-31

ProDtlnotedataFree() function  
  described  39-31

ProDtlnoteFree() function  
  described  39-31

ProDtlnotedataLineFree() function  
  described  39-31

ProDtlnoteFree() function  
  described  39-31

ProDtlnoteLineAlloc() function  
  described  39-31

ProDtlnoteLineEnvelopeGet() function  
  described  39-23

ProDtlnoteLineFree() function  
  described  39-31

ProDtlnoteLineFree() function  
  described  39-31

ProDtlnoteLineTextAdd() function  
  described  39-31

ProDtlnoteLineTextsCollect() function  
  described  39-31

ProDtlnoteLineTextsSet() function  
  described  39-31

ProDtlnoteModify() function  
  described  39-31

ProDtlnotetextAlloc() function  
  defined  39-30

ProDtlnotetextFontGet() function  
  defined  39-31

ProDtlnotetextFontSet() function  
  defined  39-31

ProDtlnotetextFree() function  
  defined  39-30

ProDtlnotetextHeightGet() function  
  defined  39-30

ProDtlnotetextHeightSet() function  
  defined  39-30

ProDtlnotetextSlantGet() function  
  defined  39-30

ProDtlnotetextSlantSet() function  
  defined  39-30

ProDtlnotetextStringGet() function  
  defined  39-31

ProDtlnotetextStringSet() function  
  defined  39-31

ProDtlnotetextThicknessGet() function  
  defined  39-31

ProDtlnotetextThicknessSet() function  
  defined  39-31

ProDtlnotetextUlineGet() function  
  defined  39-31

ProDtlnotetextUlineSet() function  
  defined  39-31

ProDtlnotetextWidthGet() function  
  defined  39-30

ProDtlnotetextWidthSet() function  
  defined  39-30

ProDtlnotetextUlineGet() function  
  defined  39-30

ProDtlnotetextWidthGet() function  
  defined  39-30

ProDtlnotetextWidthSet() function  
  defined  39-30

ProDtlnotetextUlineSet() function  
  defined  39-30

ProDtlnotetextUlineSet() function  
  defined  39-30

ProDtlsymdef  
  object handle  39-21

ProDtlsymdefCreate() function  
  description  39-23

ProDtlsymdefDataGet() function  
  description  39-23

ProDtlsymdefDelete() function  
  description  39-23

ProDtlsymdefCreate() function  
  description  39-23

ProDtlsymdefModify() function  
  description  39-23

ProDtlsymdefCreate() function  
  description  39-23

ProDtlsymgroupdataAlloc() function  
  description  39-52

ProDtlsymgroupdataFree() function  
  description  39-52

ProDtlsymgroupDataGet() function  
  description  39-51

ProDtlsymgroupdataItemAdd() function  
  description  39-52

ProDtlsymgroupdataItemsCollect() function  
  description  39-52
ProElementIsArray() function  description  16-13
ProElementIsCompound() function  description  16-14
ProElementIsMultival() function  description  16-14
ProElementValueGet() function  description  16-9
ProElementValueSet() function  description  16-9
ProElementValuesGet() function  description  16-9
ProElementValuesSet() function  description  16-9
ProElemId object  description  16-4
ProElempath object  description  16-11
ProElempathAlloc() function  description  16-12
ProElempathCompare() function  description  16-12
ProElempathCopy() function  description  16-12
ProElempathDataGet() function  description  16-12
ProElempathDataSet() function  description  16-12
ProElempathFree() function  description  16-12
ProElempathItem structure  declaration  16-11
ProElempathSizeGet() function  description  16-12
ProElemtreeElementAdd() function  adding manufacturing elements  30-17
ProElemtreeElementGet() function  description  16-14
ProElemtreeElementRemove() function  description  16-14
ProElemtreeElementVisit() function  description  16-14
ProEngineerConnect() function  description  52-5
ProEngineerConnectIdExtract() function  description  52-5
ProEngineerConnectionStart() function  description  52-4
ProEngineerDisconnect() function  description  52-5
ProEngineerEnd() function  description  52-5
ProEngineerStart() function  description  52-4
ProEngineerStatusGet() function  description  52-8
ProErritemType enum  declaration  16-17
ProError return type  description  1-19
ProEventListener declaration  16-16
ProEventProcess() function  description  52-8
ProExpldstate structure  declaration  13-14
ProExpldstateActivate() function  description  13-15
ProExpldstateActiveGet() function  description  13-15
ProExpldstateSelect() function  description  13-15
ProExportVRML function  defined  50-11
ProExtdataClass structure  description  35-3
ProExtdataCopy() function  description  35-4
ProExtdataClassNameList() function  description  35-6
ProExtdataClassRegister() function  description  35-4
ProExtdataClassUnregister() function  description  35-4
ProExtdataFree() function  description  35-6
ProExtdataInit() function  description  35-6
ProExtdataLoadAll() function  description  35-5
ProExtdataSlot structure  description  35-3
ProExtdataSlotCreate() function  description  35-4
ProExtdataSlotDelete() function  description  35-5
ProExtdataSlotIdsList() function  description  35-6
ProExtdataSlotList() function  description  35-6
ProExtdataSlotNamesList() function  description  35-6
ProExtdataSlotRead() function  description  35-6
ProExtdataSlotWrite() function  description  35-4
ProExtdataTerm() function  description  35-5
ProExtobj object  44-2
ProExtobjCallbacks structure  declaration  44-15
ProExtobjCallbacksSet() function  description  52-5
description 44-14
ProExtobjCBAct() function
description 44-14
ProExtobjCBEnable() function
description 44-14
used in a code example 44-23
ProExtobjClass object
description 44-3
ProExtobjClassCreate() function
description 44-3
used in a code example 44-20
ProExtobjClassDelete() function
description 44-3
ProExtobjClassGet() function
description 44-3
ProExtobjClassWarningDisable() function
description 44-17
ProExtobjClassWarningEnable() function
description 44-17
used in a code example 44-20
ProExtobjCreate() function
description 44-3
used in a code example 44-21
ProExtobjdataAdd() function
description 44-11
ProExtobjdataFree() function
description 44-11
ProExtobjdataGet() function
description 44-11
ProExtobjdataRemove() function
description 44-11
ProExtobjdataSet() function
description 44-11
ProExtobjdataType enum
declaration 44-12
ProExtobjDelete() function
description 44-3
ProExtobjDispprops object
description 44-9
ProExtobjExtidGet() function
description 44-5
ProExtobjExtidSet() function
description 44-5
ProExtobjExttypeGet() function
description 44-5
ProExtobjExttypeSet() function
description 44-5
ProExtobjOwnerobjGet() function
description 44-4
ProExtobjOwnerobjSet() function
description 44-4
ProExtobjRefAdd() function
description 44-13
used in a code example 44-23
ProExtobjRefAlloc() function
used in a code example 44-23
ProExtobjRefRemove() function
description 44-13
ProExtobjRefselectionGet() function
description 44-13
ProExtobjRefselectionSet() function
used in a code example 44-23
ProExtobjRefetypeGet() function
description 44-13
ProExtobjRefetypeSet() function
description 44-13
ProExtobjRefVisit() function
description 44-14
ProExtobjReusableClear() function
description 44-4
ProExtobjVisit() function
description 44-8
ProFaminstance object
description 34-2
ProFaminstanceAdd() function
description 34-4
ProFaminstanceCheck() function
description 34-5
ProFaminstanceCreate() function
description 34-5
ProFaminstanceErase() function
description 34-5
ProFaminstanceGenericGet() function
description 34-5
ProFaminstanceIsExtLocked() function
description 34-5
ProFaminstanceIsFlatState() function
description 34-5
ProFaminstanceIsVerified() function
description 34-5
ProFaminstanceMdGet() function
description 34-5
ProFaminstanceRemove() function

ProFaminstanceRetrieve() function
description 34-5
ProFaminstanceSelect() function
description 34-5
ProFaminstanceValueGet() function
description 34-4
used in a code example 34-8
ProFaminstanceValueSet() function
description 34-4
ProFamtable object
description 34-2
ProFamtableCheck() function
description 34-2
used in a code example 34-7
ProFamtableEdit() function
description 34-2
ProFamtableErase() function
description 34-2
ProFamtableInit() function
description 34-2
used in a code example 34-9
ProFamtableInstanceVisit() function
description 34-3
used in a code example 34-9
ProFamtableItem object
description 34-2
ProFamtableItemAdd() function
description 34-6
ProFamtableItemRemove() function
description 34-6
ProFamtableItemToModelitem() function
description 34-6
ProFamtableItemToParameter() function
description 34-6
ProFamtableVisit() function
description 34-3
used in a code example 34-7
ProFamtableShow() function
description 34-2
ProFeature object
description 12-2
ProFeatureChildrenGet() function
description 12-5
ProFeatureCopyPostAction callback 36-7
ProFeatureCreate() function
and ProSelection 2-4
calling 16-15
description 12-4
manufacturing objects 30-9
used to assemble components 22-13
ProFeatureCreateOptions enum
description 16-15
ProFeatureDeletePostAction callback 36-6
ProFeatureCreatePreAction callback 36-6
ProFeatureDelete() function
deleting cross sections 40-6
description 12-6, 13-14
ProFeatureDeletePostAction callback 36-7
ProFeatureDeletePreAction callback 36-7
ProFeatureElemDimensionIdGet() function
description 16-4
ProFeatureElemIsIncomplete() function
description 16-23
ProFeatureElemIsVisible() function
description 16-22
ProFeatureElemtreeCreate() function
derived 22-13
description 16-22
with NC sequences 30-32
ProFeatureElemValueGet() function
description 16-22
ProFeatureElemValuesGet() function
description 16-22
ProFeatureGeomitemVisit() function 12-6
description 9-5
to create an operation 30-28
to create NC sequences 30-31
used with axis datums 9-10
used with composite datum curves 9-13
used with coordinate system datums 9-11
used with datum curves 9-12
used with datum planes 9-11
used with datum points 9-13
ProFeatureGrppatternStatusGet() function
description 12-11
ProFeatureHasGeomChecks() function
description 12-5
ProFeatureInit() function
description 12-2
ProFeatureInsertModeActivate() function
description 12-7
used in a code example 12-7
ProFeatureInsertModeIsActive() function
description 12-5
ProFeatureIsIncomplete() function
description 12-4
ProFeatureIsNcseq() function
description 12-4
ProFeatureIsReadonly() function
description 12-5
ProFeatureModeCancel() function 12-5
ProFeatureNumberOfGet() function
description 12-8
used in a code example 12-10
ProFeatureNumberOfGet() function
description 23-14
ProFeatureParentsGet() function
description 12-5
ProFeaturePatternGet() function
description 12-12
ProFeaturePatternStatusGet() function
description 12-11
ProFeatureReadonlySet() function
description 12-8
ProFeatureReadonlyUnset() function
description 12-8
ProFeatureRedefine() function
defined 22-13
description 12-6
ProFeatureRedefinePostAction callback 36-7
ProFeatureRedefinePreAction callback 36-7
ProFeatureRegenFailureAction callback 36-7
ProFeatureRegenPostAction callback 36-7
ProFeatureRegenPreAction callback 36-7
ProFeatureReorder() function
description 12-8
used in a code example 12-10
ProFeatureReplacePostAction callback 36-7
ProFeatureReplacePreAction callback 36-7
ProFeatureReroutePostAction callback 36-7
ProFeatureReroutePreAction callback 36-7
ProFeatureSectionCopy() function
description 23-14
ProFeatureSelectionGet() function
description 12-5
ProFeatureSolidGet() function
description 12-4
ProFeatureStatusGet() function
description 9-5
possible status values 12-4
ProFeatureSubtypeGet() function
defined 12-4
ProFeatureSuppress() function
description 12-6
ProFeatureSuppressPostAction callback 36-7
ProFeatureSuppressPreAction callback 36-7
ProFeatureTypeGet() function
description 12-3
to visit geometry 9-5
used in a code example 1-59
used with assemblies 13-5
ProFeatureTypenameGet() function
description 12-3
ProFeatureVisibilityGet() function
description 12-2
visiting geometry 9-5
profem_get_bar_elements() function
description 43-11
used in a code example 43-20
profem_get_case_names() function
description 43-5
used in a code example 43-19
profem_get_constraints() function
description 43-8
used in a code example 43-19
profem_get_contacts() function
description 43-11
used in a code example 43-20
profem_get_mass_elements() function
description 43-12
used in a code example 43-19
profem_get_mesh_controls() function
description 43-5
used in a code example 43-19
profem_get_runners() function
description 43-11
used in a code example 43-20
profem_get_shell_pairs() function
description 43-13
ProFileEdit() function
description 49-5
ProFilenameParse() function
description 49-5
ProFileOpen() function
description 49-4
ProFileOpenOKAction callback 36-6
ProFilesList() function
description 49-4
ProForeignCrvCont enum
description 27-6
ProForeignCurveClassEvalSet() function
description 27-7
ProForeignCurveEvalFunction() typedef
description 27-5
ProGeomitem
declaration 1-48
description 6-8
hierarchy 9-2
list of types 9-2
ProGeomitemAngleEval() function
description 9-26
ProGeomitemdata structure
declaration 9-17
ProGeomitemdataFree() function
description 9-17
ProGeomitemdataGet() function
description 9-17
ProGeomitemDiameterEval() function
description 9-26
ProGeomitemDistanceEval() function
and ProSelection 2-4
description 9-26
ProGeomitemFeatureGet() function 12-6
description 9-10
used in a code example 1-59
used with selection D-14
ProGeomitemGtolrefClear function
description 15-3
ProGeomitemGtolrefSet function
description 15-2
ProGeomitemIsGtolref function
description 15-3
ProGeomitemIsInactive() function
description 9-6
used with axis datums 9-10
used with coordinate system datums 9-11
used with datum curves 9-12
used with datum planes 9-12
used with datum points 9-13
ProGeomSimprepRetrieve() function
description 38-6
ProGlobalInterfCalcPostAction callback 36-7
progr_color() function
equivalent D-18
progr_display_object() function
equivalent D-18
progr_draw_3d() function
equivalent D-19
progr_draw_polygon_2d() function
equivalent D-19
progr_get_color_map() function
equivalent D-18
progr_get_default_font_id() function
equivalent D-19
progr_get_font_id() function
progr_get_font_name() function
progr_get_line_style_def() function
equivalent D-19
progr_get_font_id() function
equivalent D-19
progr_get_font_name() function
equivalent D-19
progr_get_line_style_def() function
equivalent D-19
progr_invalidate_display_list() function
equivalent D-22
progr_move_3d() function
equivalent D-19
progr_put_arc() function
progr_put_circle() function
equivalent D-19
progr_put_multi_polylines() function
equivalent D-19
progr_put_polyline() function
equivalent D-19
progr_put_text() function
progr_set_color_map() function
equivalent D-18
progr_set_draw_mode() function
progr_set_line_style() function
progr_text_color() function
ProGraphicsArcDraw() function
description 8-15
ProGraphicsCircleDraw() function
description 8-15
used in a code example 8-16
ProGraphicsColorModify() function
description 8-10 to 8-11
ProGraphicsColorSet() function
displaying graphics 8-15
used in a code example 8-16
ProGraphicsLineDraw() function
description 8-15
used in a code example 8-16
ProGraphicsModeSet() function
used in a code example 8-21
ProGraphicsMultipolylinesDraw() function
description 8-15
ProGraphicsPenPosition() function
description 8-15
used in a code example 8-16
ProGraphicsPolygonDraw() function 8-15
ProGraphicsPolylineDraw() function
description 8-15
ProGraphicsSimprepRetrieve() function
description 38-6
ProGraphicsTextDisplay() function
description 8-17
ProGraphicWindowSizeGet() function
description 8-2
ProGtol*Set function
description 15-7
ProGtolCreate function
description 15-7
ProGtoldataAllAroundGet function
description 15-6
ProGtoldataAlloc function
description 15-7
ProGtoldataCompositeGet function
description 15-5 to 15-6
ProGtoldataDiameterGet function
description 15-6
ProGtoldataFree function
description 15-14
ProGtoldataFreeStateGet function
description 15-6
ProGtolDataGet function
description 15-4
ProGtolDataDatumrefGet function
description 15-5
ProGtolDataDatumrefSet function
description 15-8
ProGtolDataMatCondGet function
description 15-6
ProGtolDataModelGet function
description 15-5
ProGtolDataOwnerGet function
description 15-5
ProGtolDataPerUnitGet function
description 15-6
ProGtolDataPlacementGet function
description 15-5
Index - xxxiii

ProGtoldataPlacementSet function description 15-8
ProGtoldataProBoundaryGet function description 15-6
ProGtoldataProjZoneGet function description 15-6
ProGtoldataReferenceGet function description 15-5
ProGtoldataStatTolGet function description 15-6
ProGtoldataStatus defined 15-8
ProGtoldataTangentPlaneGet function description 15-6
ProGtoldataTypeGet function description 15-4
ProGtoldataValueGet function description 15-6
ProGtoldataDatumrefAlloc function description 15-8
ProGtoldataDatumrefFree function description 15-8
ProGtoldataDatumrefGet function description 15-8
ProGtolleader allocated and freed 15-8 defined 15-8
ProGtolleaderAlloc function description 15-8
ProGtolleaderFree function description 15-8
ProGtolleaderGet function description 15-5
ProGtolleadersFree function description 15-5
ProImportfeatCreate() function description 51-17
ProImportfeatRedefine() function redefined 51-15
ProImportfeatRedefSourceInit() function description 51-15
ProImportfeatUserIdToItemId() function description 51-16
ProInfoWindowDisplay() function description 49-5
ProInputFileRead() function description 50-19
for cable parameters 42-15
for the drawing setup file 39-6
used with cabling 42-10
ProInsertModeCancel() function description 12-7
used in a code example 12-8
ProInsertModelsActive() function description 12-5
used in a code example 12-8
prointerface_2d() function description 41-3
used in a code example 41-4
prointerface_create() function description 41-2
used in a code example 41-3
prointerface_load_function() function description 41-2
used in a code example 41-4
prointerface_object_set() function description 41-2
used in a code example 41-3
ProIntf3DFileWrite function defined 50-8
ProIntfDataAccuracyGet() function description 51-18
ProIntfDataAccuracytypeGet() function description 51-18
ProIntfDataAlloc() function description 51-4
ProIntfDataDatumAppend() function description 51-13
ProIntfDataEdgeAppend() function description 51-11
ProIntfDataSourceInit() function description 51-2, 51-13
ProIntfDataSurfaceAppend() function description 51-8
ProIntfImportLayerFilter() function described 50-23
ProIntfimportModelCreate() function
described 50-23
ProIntfimportSourceTypeGet() function
description 50-23
ProIntfSliceFileExport() function
description 50-10
ProItemerror structure
declaration 16-16
ProJlinkApplicationStart() function
description 53-9
ProJlinkApplicationStop() function
description 53-10
ProJlinkTaskExecute() function
description 53-10
prolayer_add_item() function
equivalent D-19
prolayer_create_layer() function
equivalent D-20
prolayer_delete_layer() function
equivalent D-20
prolayer_display() function
equivalent D-20
prolayer_get_display() function
equivalent D-20
prolayer_get_items() function
equivalent D-20
prolayer_get_names() function
equivalent D-19
prolayer_remove_item() function
equivalent D-20
ProLayerCreate() function
description 6-13
ProLayerDefLayerGet() function
description 6-14
ProLayerDefLayerSet() function
description 6-14
ProLayerDelete() function
description 6-13
ProLayerDisplay enum
description 6-13
ProLayerDisplaystatusGet() function
description 6-13
ProLayerDisplaystatusSave() function
description 6-14
ProLayerDisplaystatusSet() function
description 6-13
ProLayerfilterdataActionSet() function
described 50-24
ProLayerfilterdataCountsGet() function
described 50-24
ProLayerfilterdataNameGet() function
described 50-24
ProLayerfilterdataNameSet() function
described 50-24
ProLayerItemAdd() function
description 6-13
ProLayerItemInit() function
description 6-13
ProLayeritemLayersGet() function
description 6-13
ProLayerItemRemove() function
description 6-13
ProLayerItemsGet() function
description 6-13
ProLayerViewDependencyGet() function
description 6-14
ProLayerViewDependencySet() function
description 6-14
ProLinedataInit function
description 51-10
ProLinedataInit() function
defined 39-24
ProLineStyleDefGet() function
description 8-14
ProLineStyleSet() function
description 8-14
displaying graphics 8-15
proload_cmd_sequence() function
equivalent D-17
ProLocalGroupCreate() function
description 12-17
ProMacroExecute() function
description 4-45
ProMAssProperty structure
description 7-9
ProMaterial object
description 7-10
ProMaterialCreate() function
description 7-11
used in a code example 7-14
ProMaterialCurrentGet() function
description 7-11
used in a code example 7-14
ProMaterialCurrentSet() function
description 7-11
used in a code example 7-14
ProMaterialdata object
description 7-10
ProMaterialDataGet() function
description 7-11
used in a code example 7-14
ProMaterialDataSet() function
description 7-11
used in a code example 7-14
ProMaterialDelete() function
description 7-12
used in a code example 7-13
ProMaterialfileRead() function
description 7-12
used in a code example 7-14
ProMaterialfileWrite() function
description 7-12
used in a code example 7-14
ProMatrixInit() function
description 10-8
ProMdl object
declaration 1-47
description 6-2
ProMdlBackup() function 6-6
ProMdlCopy() function 6-6
ProMdlCopyPostAction callback 36-6
ProMdlCopyPostAllAction callback 36-6
ProMdlCopyPreAction callback 36-6
ProMdlCreateCancelAction callback 36-6
ProMdlCreatePostAction callback 36-6
ProMdlCreatePreAction callback 36-6
ProMdlCurrentGet() function
description 6-5
ProMdlDataGet() function
description 6-4
ProMdlDbmsFailureAction callback 36-6
ProMdlDeclaredList() function
description 6-6
ProMdlDelete() function 6-6
ProMdlDeletePostAction callback 36-6
ProMdlDeletePostAllAction callback 36-6
ProMdlDeletePreAction callback 36-6
ProMdlDependenciesList() function
description 6-6
ProMdlDisplayPostAction callback 36-7
ProMdlDisplayPreAction callback 36-7
ProMdlErase() function 6-6
ProMdlEraseAll() function
defined 6-7
ProMdlErasePostAction callback 36-6
ProMdlErasePostAllAction callback 36-6
ProMdlErasePreAction callback 36-6
ProMdlErasePostAction() function
description 6-6
ProMdlGetVisit function
description 15-3
ProMdlIdGet() function
description 6-4
used in a code example 38-11
ProMdlInit() function
description 6-4
ProMdlLayerGet() function
description 6-12
ProMdlLayerVisit() function
description 6-14
ProMdlModificationVerify() function
description 6-6
ProMdlMultipleRetrieve() function
defined 6-7
ProMdlNameGet() function
description 6-4
ProMdlNoteVisit() function
description 7-16
ProMdlPurgePostAction callback 36-6
ProMdlPurgePreAction callback 36-6
ProMdlRename() function 6-6
ProMdlRenamePostAction callback 36-6
ProMdlRenamePreAction callback 36-6
ProMdlRetrieve() function
defined 6-7
description 6-6
used in a code example 7-3
ProMdlRetrievePostAction callback 36-6
ProMdlRetrievePostAllAction callback 36-6
ProMdlRetrievePreAction callback 36-6
ProMdlSave() function 6-6
ProMdlSavePostAction callback 36-6
ProMdlSavePostAllAction callback 36-6
ProMdlSavePreAction callback 36-6
ProMdlStartAction callback 36-6
ProMdlSubtypeGet() function
defined 6-5
ProMdlToModelitem() function
description 6-5
used with parameters 14-4
ProMdlTypeGet() function
description 6-4
used in a code example 38-11
used with assemblies 13-6
ProMdlVerstampGet() function
description 6-11
promenu_action() function
equivalent D-16
promenu_create() function
equivalent D-16
promenu_exit_action_up() function
equivalent D-16
promenu_exit_up() function
equivalent D-16
promenu_expand() function
equivalent D-16
promenu_get_pick() function
equivalent D-19
promenu_is_up() function
equivalent D-17
promenu_load_action() function
equivalent D-16
promenu_load_post_func() function
equivalent D-17
promenu_load_pre_func() function
equivalent D-17
promenu_make() function
equivalent D-16
promenu_make_compound() function
equivalent D-16
promenu_make_item_accessible() function
equivalent D-17
promenu_make_item_inaccessible() function
equivalent D-17
promenu_no_exit() function
Index - xxxv
promenu_on_button() function  equivalent D-16
promenu_pop() function  equivalent D-17
promenu_push() function  equivalent D-17
promenu_push_command() function  equivalent D-17
promenu_remove_item() function  equivalent D-17
promenu_reset_item() function  equivalent D-17
promenu_set_data_mode() function  equivalent D-17
promenu_set_item() function  equivalent D-17
promenu_set_item_location() function  equivalent D-17
promenu_set_item_visible() function  equivalent D-17
promenu_set_mode() function  equivalent D-17
ProMenuAuxfileRegister() function  description 4-28
ProMenubarMenuAdd() function  used in a code example 4-17
ProMenubarMenuAddMenu() function  description 4-15
used in a code example 4-17
ProMenubarMenuPushButtonAdd() function  syntax 4-8
used in a code example 4-17
ProMenubarMenuRadioGrpValueGet() function  description 4-13
ProMenubarMenuRadioGrpValueSet() function  description 4-13
ProMenubuttonActionSet() function  description 4-29
using the final arguments 4-34
ProMenubuttonActivate() function  description 4-42
ProMenubuttonDeactivate() function  description 4-42
ProMenubuttonGenActionSet() function 4-28
ProMenubuttonHighlight() function  description 4-41
ProMenubuttonLocActionSet() function  description 4-39
ProMenubuttonPostActionSet() function  description 4-37
ProMenubuttonPreActionSet() function  description 4-37
used in a code example 4-38
ProMenubuttonUnhighlight() function  description 4-41
ProMenuCommandPush() function  description 4-44
ProMenuCreate() function  description 4-33
to create submenus 4-39
ProMenuDatamodeSet() function  description 4-40
ProMenuDelete() function  description 4-32
used with ProMenuProcess() 4-35
ProMenuDeleteWithStatus() function  description 4-35
ProMenuFileRegister() function  calling 4-28
used in a code example 4-38
ProMenuPromStringsRegister() function  description 4-43
ProMenuHold() function  description 4-32
ProMenuModeSet() function  description 4-40
ProMenuPop() function  description 4-42
ProMenuProcess() function  description 4-33
returning a value from 4-35
ProMenuPush() function  description 4-42
ProMenuStringsSelect() function  description 4-43, 5-42
ProMenuVisibilityGet() function  description 4-42
ProMessageClear() function  description 3-5
ProMessageDisplay() function  description 2-5, 3-4, 3-9
used in a code example 4-38
ProMessageDoubleRead() function  description 3-9
ProMessageIntegerRead() function  description 3-9
ProMessagePasswordRead() function  description 3-9
ProMessagePasswordRead() function  description 3-9
used in a code example 4-38
ProMessageToBuffer() function  description 3-9
Promfg_get_nc_type() function  equivalent D-27
Promfg_get_tool_ids() function  equivalent D-27
Promfg_get_tool_parameters() function  equivalent D-27
promfg_set_tool_parameters() function
equivalent D-27
ProMfgAssemGet() function
description 30-2
ProMfgCreate() function
description 30-2
ProMfgdbDataAlloc() function
description 31-3
ProMfgdbDataDbnameAdd() function
description 31-4
ProMfgdbLoginAction() function
description 31-2
ProMfgdbLogoffAction() function
description 31-2
ProMfgdbNameCreate() function
definition 31-4
ProMfgdbRegister() function
description 31-3
ProMfgdbSearchAction() function
description 31-2
ProMfgdbSearchoptAllowedValueAdd() function
description 31-4
ProMfgdbSearchoptApplicDataAdd() function
description 31-4
ProMfgFeatureOwnerGet() function
description 30-6
ProMfgSolidGet() function
description 30-6
ProMfgTypeGet() function
description 30-8
ProMode object
description 30-2
ProModeCurrentGet() function
description 6-2
used in a code example 8-8
PromodelcheckExecute() function
description 49-7
ProModelitem object
declaration 1-48
description 6-8
ProModelitemByNameInit() function
description 6-8
PromodelitemDefaultnameGet() function
description 6-9
ProModelitemInit() function
description 6-9
ProCurve to a ProGeomitem 9-12
ProPoint to a ProGeomitem 9-13
ProSurface to a ProGeomitem 9-12
used in a code example 1-59
used in visits 9-10
used with parameters 14-4
ProModelitemMdGet() function
description 6-9
ProModelitemNameCanChange() function
description 6-9
ProModelitemNameGet() function
description 6-9
used in a code example 1-59
used with ProGeomitem 9-3
ProModelitemNameSet() function
description 6-9
used with ProGeomitem 9-3
ProModelitemToFamtableItem() function
description 34-6
ProModelitemToRelset() function
description 11-9
ProModelitemUsernameDelete() function
description 6-9
ProModeNameCurrentDelete() function
description 6-2
Promouseboxinput() function 8-20
PromousepickGet() function
description 8-20
used in a code example 8-16
PromouseTrack() function
description 8-20
used in a code example 8-21
promag_clear() function
equivalent D-17
promag_getdouble() function
equivalent D-18
promag_getint() function
equivalent D-17
promag_getpassword() function
equivalent D-18
promag_getstring() function
equivalent D-17
promag_print() function
promag_sprint() function
ProNcseqClPostAction callback 36-7
ProNcseqCutTimeGet() function
description 30-38
ProNcseqElemHolesetAdd() function
description 30-34
ProNcseqElemHolesetAxisAdd() function
description 30-34
ProNcseqElemHolesetDepthBySet() function
description 30-34
ProNcseqElemHolesetDepthSet() function
description 30-34
ProNcseqElemHolesetDepthTypeSet()
function description 30-34
ProNcseqElemHolesetDirectionSet() function description 30-34
ProNcseqElemHolesetEndSet() function description 30-34
ProNcseqElemHolesetStartSet() function description 30-34
ProNcseqElemMillsurfSet() function description 30-33
ProNcseqElemSurfaceAdd() function description 30-33
ProNcseqElemSurfaceFlipSet() function description 30-33
ProNcseqNumGet() function description 30-38
ProNcseqRemovedVolGet() function description 30-38
ProNcseqToolpathDisplay() function description 30-38
ProNoteAttach object description 7-17
ProNoteAttachAddend() function description 7-17
used in a code example 7-20
ProNoteAttachAlloc() function description 7-17
used in a code example 7-20
ProNoteAttachFreeGet() function description 7-17
ProNoteAttachFreeSet() function description 7-17
used in a code example 7-21
ProNoteAttachRelease() function description 7-17
used in a code example 7-21
ProNoteCreate() function description 7-15
used in a code example 7-20
ProNoteDelete() function description 7-15
used in a code example 7-21
ProNoteDisplay() function description 7-15
used in a code example 7-21
ProNoteLineListGet() function description 7-15
ProNoteLineListSet() function description 7-15
ProNoteOwnerGet() function description 7-16
ProNotePlacementGet() function description 7-17
used in a code example 7-21
ProNoteTextStyleGet() function description 7-18
ProNoteTextStyleSet() function description 7-18
ProNoteURLGet() function description 7-17
ProNoteURLSet() function description 7-17
ProNotificationSet() function description 36-5
ProNotificationUnset() function description 36-8
ProNoteType enum description 36-6
ProObjectWindowCreate() function description 8-4
used in a code example 7-4
ProObjModelsCheckOut() function used in a code example 47-3
ProObjModelsCheckout() function description 47-2
ProOptionOrderedVerify() function description 49-3
ProOutputAssemblyConfigurationIsSupporte d function defined 50-9
ProOutputFileWrite() function description 50-2
for cable parameters 42-15
for cabling 42-10, 50-3
for the drawing setup file 39-6
used with cabling 42-10
ProOutputInclusionAlloc function defined 50-9
ProOutputInclusionFlagsSet function
ProOutputInclusionFree function defined 50-10
ProOutputLayerOptionsAlloc function defined 50-10
ProOutputLayerOptionsAutoidSet function defined 50-10
ProOutputLayerOptionsSetupfileSet function defined 50-10
ProParameter object description 14-2
ProParameter structure declaration 14-2
ProParameterCreate() function description 14-5
ProParameterCreatePreAction callback 36-7
ProParameterDelete() function description 14-5
ProParameterDeletePreAction callback 36-7
ProParameterDesignationAdd() function description 14-7
ProParameterDesignationRemove() function
ProPatternClass object
   description  21-2
ProPatternCreate() function
   description  21-8
   used in a code example  21-13
ProPatternDelete() function
   description  12-12
ProPatternElementCreate() function
   description  21-13
ProPatternLeaderGet() function
   description  12-12
Properties
   material  7-10
   of external object display  44-8
   surface  8-22
   text style  7-18
ProPtTrfEval() function
   description  10-5
   used in a code example  10-6
   used with assemblies  13-11
ProPoint object
   declaration  1-47
ProPoint3d typedef
   declaration  10-4
ProPointCoordGet() function
   description  9-23
ProPointIdGet() function
   description  9-13
ProPointInit() function
   description  9-13
ProPostfixIdToSolid() function
   description  7-9
ProPrintExecute() function
   description  50-5
ProPrintMdlOptionsGet() function
   description  50-5
ProPrintPCFOptionsGet() function
   description  50-5
ProPrintPlacementOptionsGet() function
   description  50-5
ProPrintPrinterOptionsGet() function
   description  50-5
ProProcstep structure
   declaration  33-2
ProProcstepActiveGet() function
   description  33-2
ProProcstepActiveSet() function
   description  33-2
ProProcstepInit() function
   description  33-2
ProProcstepNumberGet() function
   description  33-3
ProProcstepVisit() function
   description  33-2
ProProductviewExport() function
   description  50-4, 50-11

description  14-7
ProParameterDesigntationVerify() function
   description  14-7
ProParameterInit() function
   description  14-4
ProParameterIsModified() function
   description  14-4
ProParameterModifyPreAction callback  36-7
ProParameterSelect() function
   description  14-5
   used with family table items  34-6
ProParameterToFamtableItem() function
   description  34-6
ProParameterValueGet() function
   description  14-4
ProParameterValueReset() function
   description  14-4
ProParameterValueSet() function
   description  14-4
ProParameterVisit() function
   description  14-5
ProParamfrom enum
   declaration  14-2
ProParamowner structure
   declaration  14-2
ProParamvalue structure
   declaration  14-3
ProParamvalueSet() function
   description  14-3
ProParamvalueType structure
   declaration  14-3
ProParamvalueGetType() function
   description  14-3
ProParamvalueValue Get structure
   declaration  14-3
ProPartDensityGet() function
   description  7-10
ProPartDensitySet() function
   description  7-10
ProPartMaterialsGet() function
   description  7-11
ProPartSimprefRetrieve() function
   description  38-6
ProPartTessellate 9-14
ProPartTessellate() function description 9-14
ProPartTessellationFree() function
   description  9-14
ProPartToProIntfData() function
   description  51-18
ProPathCreate() function
   description  49-5
ProPattern object
   description  21-2
proptn_get_leader() function
description 12-12
using with pattern tables 12-16
proptnbl_add_inst_to_table() function 12-16
proptnbl_delete_table() function 12-16
proptnbl_get_active_table() function
description 12-17
proptnbl_get_all_tbl_names() function
description 12-17
proptnbl_get_inst_dim_value() function
description 12-17
proptnbl_get_inst_indices() function
description 12-17
proptnbl_get_lead_pat_dims() function
description 12-17
proptnbl_remove_instance() function 12-16
proptnbl_rename_table() function 12-16
proptnbl_set_active_table() function
description 12-17
proptnbl_set_dimval_driven() function
description 12-17
proptnbl_set_inst_dim_value() function
description 12-17
ProQuiltdataAlloc() function
description 51-11
ProQuiltdataGet() function
defined 9-22
ProQuiltdataInit() function
description 51-11
ProQuiltdataSurfArrayGet() function
defined 9-22
ProQuiltdataSurfArraySet() function
defined 9-22
description 51-11
ProRasterFileWrite() function
defined 50-10
ProRelationEval() function
description 11-10
ProRelset object
description 11-9
ProRelsetCreate() function
description 11-10
ProRelsetDelete() function
description 11-10
ProRelsetRegenerate() function
description 11-10
ProRelsetRelationsGet() function
description 11-10
ProRelsetRelationsSet() function
description 11-10
ProRelsetToModelitem() function
description 11-10
ProRmdtBoundBoxPreAction callback
36-7 to 36-8
ProRmdtCavLayoutPreAction callback
36-7 to 36-8
ProRmdtCreateImmPreAction callback
36-7 to 36-8
ProRmdtCreateWpPreAction callback
36-7 to 36-8
ProRmdtMaterialPreAction callback 36-8
ProRmdtMBaseSelectPreAction callback 36-8
ProRuleEval() function
description 38-15
ProRuleInitDist() function
description 38-16
ProRuleInitExpr() function
description 38-15
ProRuleInitName() function
description 38-15
ProRuleInitRep() function
description 38-17
ProRuleInitSize() function
description 38-16
ProRuleInitZone() function
description 38-16
ProSecdimCreate() function
description 23-5
ProSecdimDelete() function 23-5
ProSecdimDiamClear
function definition 23-7
ProSecdimDiamInquire
function definition 23-7
ProSecdimDiamSet
function definition 23-7
ProSecdimValueGet() function
description 23-9
ProSecdimValueSet() function
description 23-9
ProSecerrorAdd() function
description 23-12
ProSecerrorAlloc() function
description 23-12
ProSecerrorCount() function
description 23-12
used in a code example 23-13
ProSecerrorFree() function
used in a code example 23-13
ProSecerrorGet() function
description 23-12
used in a code example 23-13
ProSecerrorItemGet() function
description 23-12
used in a code example 23-13
ProSecerrorMsgGet() function
description 23-12
used in a code example 23-13
ProSection2DAlloc() function
description 23-3
used in a code example 23-3
ProSectionAutodim() function
description 23-10
ProSectionConstraintDeny() function
description 23-10
ProSectionConstraintsIdsGet() function
description 23-10
ProSectionEntityAdd() function
description 23-4
used in a code example 23-4
ProSectionEntityDelete() function 23-4
description 23-4
ProSectionEntityReplace() function
description 23-9
ProSectionEpsilonGet() function
description 23-9
ProSectionEpsilonSet() function
description 23-9
ProSectionFree() function
description 23-3
ProSectionLocationGet() function
description 10-8
ProSectionNameGet() function 23-3
description 23-3
used in a code example 23-3
ProSectionRegenerate() function
description 23-8
ProSectionSolve() function
description 23-8
ProSectionSolveRigid() function
description 23-9
ProSelbox object
description 44-10
ProSeldatAlloc() function
description 44-10
ProSeldatSelboxesGet() function
description 44-10
ProSeldatSelboxesSet() function
description 44-10
ProSelect() function
and interactive selection 2-5
arguments 2-7
description 2-4
guidelines D-15
ProSelection 2-2
compared to Select3d D-15
information contained in 2-3
object 2-2
ProSelectionAlloc() function D-13
description 2-4
used in a code example 1-59
ProSelectionAsmcompathGet() function 2-2
ProSelectionCopy() function
description 2-5
ProSelectionDepthGet() function
description 9-24
ProSelectionDisplay() function
description 2-9
ProSelectionDwgtbclcellGet() function
description 39-55
ProSelectionFree() function
description 2-5
used in a code example 1-59
ProSelectionHighlight() function
and ProSelection 2-4
description 2-9
guidelines D-15
used in a code example 1-59
ProSelectionModelitemGet
to return gtol handle 15-2
ProSelectionModelitemGet() function 2-2
used with family table items 34-6
ProSelectionPoint3dGet() function 2-2
ProSelectionSet() function D-13
description 2-4
ProSelectionUnhighlight() function
and ProSelection 2-4
description 2-9
ProSelectionUvParamSet() function D-13
description 2-2
ProSelectionVerify() function D-13
description 2-4
ProSessionMdList() function
description 6-5
ProShrinkwrapoptionsAdditionalcomponents
Set function
defined 50-17
ProShrinkwrapoptionsAdditionalsurfacesSet
function
defined 50-16
ProShrinkwrapoptionsAlloc function
defined 50-15
ProShrinkwrapoptionsAssignmasspropsSet
function
defined 50-16
ProShrinkwrapoptionsAutoholefillingSet
function
defined 50-15
ProShrinkwrapoptionsDatumrefsSet function
defined 50-16
ProShrinkwrapoptionsFacetedformatSet
function
defined 50-16
ProShrinkwrapoptionsFramesFileSet
function
defined 50-16
ProShrinkwrapoptionsIgnorequiltsSet
function
defined 50-15
ProShrinkwrapoptionsIgnoreskeletonSet
function
defined 50-15
ProShrinkwrapoptionsFramesFileSet
function
defined 50-16
ProShrinkwrapoptionsFree function
defined 50-15
ProShrinkwrapoptionsIgnorequiltsSet
function
defined 50-15
ProShrinkwrapoptionsIgnoreskeletonSet
function
defined 50-15
ProShrinkwrapoptionsIgnoresmallsurfsSet function
defined 50-16
ProSimprep object
description 38-2
ProSimprepActionInit() function
description 38-13
used in a code example 38-8
ProSimprepActivate() function
description 38-4
used in a code example 38-8
ProSimprepActiveGet() function
description 38-4
ProSimprepCreate() function
used in a code example 38-8
ProSimprepdata structure
description 38-2
ProSimprepdataAlloc() function
used in a code example 38-8
ProSimprepdataDefltGet() function
description 38-9
used in a code example 38-10
ProSimprepdataDefltSet() function
description 38-13
ProSimprepdataGet() function
description 38-9
ProSimpredataitemAdd() function
description 38-13
used in a code example 38-8
ProSimpredataitemDelete() function
description 38-13
used in a code example 38-8
ProSimpredataitemInit() function
description 38-13
used in a code example 38-8
ProSimpredataitemsVisit() function
description 38-9
used in a code example 38-11
ProSimprepdataNameGet() function
description 38-9
used in a code example 38-10
ProSimprepdataNameSet() function
description 38-13
ProSimprepdataSet() function
description 38-13
used in a code example 38-8
ProSimprepdataTmpvalGet() function
description 38-9
used in a code example 38-10
ProSimprepdataTmpvalSet() function
description 38-13
ProSimprepDelete() function
description 38-6
ProSimprepInit() function
description 38-4
ProSimprepdata object
description 38-2
ProSimprepSelect() function
description 38-4
ProSingleAnimationClear() function
description 38-4
ProSingleAnimationInit() function
description 46-5
ProSingleAnimationPlay() function
description 46-5
ProSmtBendedgeChildGet() function
defined 9-36
ProSmtBendedgeParentGet() function
defined 9-35
ProSmtBendsrfChildGet() function
defined 9-35
ProSmtBendsrfParentGet() function
defined 9-35
ProSmtDevldimDrivenSet() function
defined 9-42
ProSmtDevldimIsDrivenSet() function
defined 9-42
ProSmtedgeContourGet() function
defined 9-34
ProSmtFeatureBendtableApply() function
defined 9-41
ProSmtFeatureBendtableRemove() function
defined 9-41
ProSmtDevldimsGet() feature
defined 9-42
ProSmtMidIsFlatStateInstance() function
description 9-36
ProSmtOppedgeGet() function
defined 9-35
ProSmtOppsurfGet() function
defined 9-34
ProSmtPartBendtableApply() function
defined 9-41
ProSmtPartBendtableRemove() function
defined 9-41
ProSmtPartThicknessGet() function
defined 9-34
ProSmtSurfaceTypeGet() function
description 9-36
ProSolid object
declaration 1-47
ProSolidAccuracyGet() function
description 7-7
ProSolidAccuracySet() function
description 7-7
ProSolidAnalysisVisit function
defined 29-29
ProSolidAxisVisit() function
description 9-10
overview 7-2
ProUICheckbuttonActivateActionSet() function description 5-22
ProUICheckbuttonDisable() function description 5-29
ProUICheckbuttonEnable() function description 5-29
ProUICheckbuttonGetState() function description 5-29
ProUICheckbuttonHelpTextSet() function description 5-29
ProUICheckbuttonHide() function description 5-29
ProUICheckbuttonIsEnabled() function description 5-29
ProUICheckbuttonIsVisible() function description 5-29
ProUICheckbuttonSet() function description 5-29
ProUICheckbuttonShow() function description 5-29
ProUICheckbuttonUnset() function description 5-29
ProUIDialogActivate() function description 5-22
ProUIDialogActivateActionSet() function description 5-23
ProUIDialogCloseActionSet() function description 5-23
ProUIDialogCreate() function description 5-22
ProUIDialogDefaultbuttonSet() function description 5-23
ProUIDialogDestroy() function description 5-23
ProUIDialogExit() function description 5-23
ProUIDialogExit() function description 5-22
ProUIDialogTitleSet() function description 5-22
ProUIDialogActivateActionSet() function description 5-23
ProUIDialogCloseActionSet() function description 5-23
ProUIDialogCreate() function description 5-23
ProUIDialogDefaultbuttonSet() function description 5-23
ProUIDialogDestroy() function description 5-23
ProUIDialogExit() function description 5-23
ProUIDialogExit() function description 5-22
ProUIDialogTitleSet() function description 5-22
ProUIDialogActivateActionSet() function description 5-23
ProUIDialogCloseActionSet() function description 5-23
ProUIDialogCreate() function description 5-23
ProUIDialogDefaultbuttonSet() function description 5-23
ProUIDialogDestroy() function description 5-23
ProUIDialogExit() function description 5-23
ProUIDialogExit() function description 5-22
ProUIDialogTitleSet() function description 5-22
ProUIDialogActivateActionSet() function description 5-23
ProUIDialogCloseActionSet() function description 5-23
ProUIDialogCreate() function description 5-23
ProUIDialogDefaultbuttonSet() function description 5-23
ProUIDialogDestroy() function description 5-23
ProUIDialogExit() function description 5-23
ProUIDialogExit() function description 5-22
ProUIDialogTitleSet() function description 5-22
ProUIDialogActivateActionSet() function description 5-23
ProUIDialogCloseActionSet() function description 5-23
ProUIDialogCreate() function description 5-23
ProUIDialogDefaultbuttonSet() function description 5-23
ProUIDialogDestroy() function description 5-23
ProUIDialogExit() function description 5-23
ProUIDialogExit() function description 5-22
ProUIDialogTitleSet() function description 5-22
ProUIDialogActivateActionSet() function description 5-23
ProUIDialogCloseActionSet() function description 5-23
ProUIDialogCreate() function description 5-23
ProUIDialogDefaultbuttonSet() function description 5-23
ProUIDialogDestroy() function description 5-23
ProUIDialogExit() function description 5-23
ProUIDialogExit() function description 5-22
ProUIDialogTitleSet() function description 5-22
ProUIDialogActivateActionSet() function description 5-23
ProUIDialogCloseActionSet() function description 5-23
ProUIDialogCreate() function description 5-23
ProUIDialogDefaultbuttonSet() function description 5-23
ProUIDialogDestroy() function description 5-23
ProUIDialogExit() function description 5-23
ProUIDialogExit() function description 5-22
ProUIDialogTitleSet() function description 5-22
ProUIDialogActivateActionSet() function description 5-23
ProUIDialogCloseActionSet() function description 5-23
ProUIDialogCreate() function description 5-23
ProUIDialogDefaultbuttonSet() function description 5-23
ProUIDialogDestroy() function description 5-23
ProUIDialogExit() function description 5-23
ProUIDialogExit() function description 5-22
ProUIDialogTitleSet() function description 5-22
ProUIDialogActivateActionSet() function description 5-23
ProUIDialogCloseActionSet() function description 5-23
ProUIDialogCreate() function description 5-23
ProUIDialogDefaultbuttonSet() function description 5-23
ProUIDialogDestroy() function description 5-23
ProUIDialogExit() function description 5-23
ProUIDialogExit() function description 5-22
ProUIDialogTitleSet() function description 5-22
description 5-55
ProUIOptionmenuHide() function
description 5-56

ProUIOptionmenuInputActionSet() function
description 5-55

ProUIOptionmenuIsVisible() function
description 5-56

ProUIOptionmenuIItemhelptextSet() function
description 5-55

ProUIOptionmenuLabelsSet() function
description 5-55

ProUIOptionmenuLastentereditemGet() function
description 5-55

ProUIOptionmenuNamesSet() function
description 5-55

ProUIOptionmenuItemhelptextSet() function
description 5-55

ProUIOptionmenuLabelsGet() function
description 5-55

ProUIOptionmenuLabelsSet() function
description 5-55

ProUIOptionmenuSelectActionSet() function
description 5-55

ProUIOptionmenuSelectednamesGet() function
description 5-55

ProUIOptionmenuSelectednamesSet() function
description 5-55

ProUIOptionmenuShow() function
description 5-55

ProUIOptionmenuTriggerhighlightActionSet( ) function
description 5-55

ProUIOptionmenuValueGet() function
description 5-55

ProUIOptionmenuValueSet() function
description 5-55

ProUIProgressbarHide() function
description 5-59

ProUIProgressbarVisiblerowsSet() function
description 5-55

ProUIProgressbarShow() function
description 5-59

ProUIProgressbarIItemhelptextSet() function
description 5-61

ProUIProgressbarIsVisible() function
description 5-59

ProUIProgressbarValueGet() function
description 5-59

ProUIProgressbarValueSet() function
description 5-59

ProUIRadiogroupDisable() function
description 5-67

ProUIRadiogroupEnable() function
description 5-67

ProUIRadiogroupHelptextSet() function
description 5-66

ProUIRadiogroupHide() function
description 5-67

ProUIRadiogroupIsEnabled() function
description 5-67

ProUIRadiogroupIsVisible() function
description 5-67

ProUIRadiogroupLastentereditemGet() function
description 5-67

ProUIRadiogroupNamesGet() function
description 5-66

ProUIRadiogroupNamesSet() function
description 5-66

ProUIRadiogroupSelectActionSet() function
description 5-66

ProUIRadiogroupSelectednamesGet() function
description 5-66 to 5-66

ProUIRadiogroupSelectednamesSet() function
description 5-66 to 5-66

ProUIRadiogroupShow() function
description 5-67

ProUIRadiogroupLabelsGet() function
description 5-66

ProUIRadiogroupLabelsSet() function
description 5-66

ProUIRadiogroupNamesGet() function
description 5-66

ProUIRadiogroupNamesSet() function
description 5-66

ProUIRadiogroupSelectActionSet() function
description 5-66

ProUIRadiogroupSelectednamesGet() function
description 5-66 to 5-66

ProUIRadiogroupSelectednamesSet() function
description 5-66 to 5-66

ProUIPushbuttonActivateActionSet() function
description 5-69

ProUIPushbuttonHelptextSet() function
description 5-61

ProUIPushbuttonHide() function
description 5-61

ProUIPushbuttonIsEnabled() function
description 5-61

ProUIPushbuttonShow() function
description 5-61

ProUIPushbuttonUpdateActionSet() function
description 5-61

ProUISliderDisable() function
description 5-69

ProUISliderEnable() function
description 5-69

ProUISliderHelptextSet() function
description 5-65 to 5-66

ProUISliderIsEnabled() function
description 5-69

ProUISliderIsVisible() function
description 5-69

ProUISliderReadonly() function
description 5-69

ProUISliderUpdateActionSet() function
description 5-69

ProUISpinboxActivateActionSet() function
description 5-79

ProUISpinboxDisable() function
description 5-79

ProUISpinboxEditable() function
description 5-79

ProUISpinboxEnable() function
description 5-79

ProUISpinboxHelptextSet() function
description 5-65 to 5-66

ProUISpinboxIsEnabled() function
description 5-79

ProUISpinboxIsVisible() function
description 5-79

ProUISpinboxValueGet() function
description 5-79

ProUISpinboxValueSet() function
description 5-79

ProUITabDisable() function
description 5-80

ProUITabEnable() function
description 5-80

ProUITabHelptextSet() function
description 5-65 to 5-66

ProUITabIsEnabled() function
description 5-80

ProUITabIsVisible() function
description 5-80

Index - xlvi
ProUITabHelpTextSet() function
  description  5-80
ProUITabHide() function
  description  5-80
ProUITabIsEnabled() function
  description  5-80
ProUITabIsVisible() function
  description  5-80
ProUITabItemHelpTextSet() function
  description  5-80
ProUITabLabelsGet() function
  description  5-80
ProUITabLabelsSet() function
  description  5-80
ProUITableActivateActionSet() function
  description  5-100
ProUITableAnchorCellGet() function
  description  5-98
ProUITableAnchorCellSet() function
  description  5-98
ProUITableArmActionSet() function
  description  5-100
ProUITableAutohighlightDisable() function
  description  5-89
ProUITableAutohighlightEnable() function
  description  5-89
ProUITableCellComponentCopy() function
  description  5-94
ProUITableCellComponentDelete() function
  description  5-94
ProUITableCellComponentNameGet() function
  description  5-94
ProUITableCellComponentNameSet() function
  description  5-94
ProUITableCellDisable() function
  description  5-94
ProUITableCellEnable() function
  description  5-94
ProUITableCellHelpTextSet() function
  description  5-94
ProUITableCellLabelGet() function
  description  5-94
ProUITableCellLabelSet() function
  description  5-94
ProUITableColumnIndexGet() function
  description  5-94
ProUITableColumnLabelGet() function
  description  5-94
ProUITableColumnLabelSet() function
  description  5-94
ProUITableColumnLabelsGet() function
  description  5-94
ProUITableColumnLabelsSet() function
  description  5-94
ProUITableColumnNamesGet() function
  description  5-92
ProUITableColumnnamesSet() function
  description  5-92
ProUITableColumnRename() function
  description  5-93
ProUITableColumnResizingFactorGet() function
  description  5-93
ProUITableColumnResizingFactorSet() function
  description  5-93
ProUITableColumnsDelete() function
  description  5-93
ProUITableColumnselectActionSet() function
  description  5-100
ProUITableColumnSelectionPolicySet() function
  description  5-97
ProUITableColumnsInsert() function
  description  5-93
ProUITableColumnsSet() function
  description  5-88
ProUITableColumnWidthGet() function
  description  5-93
ProUITableColumnWidthSet() function
  description  5-93
ProUITableColumnwidthsSet() function
  description  5-93
ProUITableComponentCopy() function
  description  5-89
ProUITableComponentDelete() function
  description  5-89
ProUITableDisallow() function
  description  5-89
ProUITableDisarmActionSet() function
  description  5-100
ProUITableEnable() function
  description  5-89
ProUITableFocusCellGet() function
  description  5-98
ProUITableFocusCellSet() function
  description  5-98
ProUITableFocusinActionSet() function
  description  5-101
ProUITableFocusoutActionSet() function
  description  5-101
ProUITableHelpTextSet() function
  description  5-89
ProUITableHide() function
  description  5-89
ProUITableIsAutohighlightEnabled() function
  description  5-87
ProUITableIsCellSensitive() function
  description  5-87
ProValueArrayToWstringArray() function
  description  16-10
ProValueData structure
description  16-9
ProValueDataGet() function
description  16-10
ProValueDataSet() function
description  16-10
ProValueDataStringSet() function
description  16-10, 53-3
ProValueDataTransformGet() function
description  16-10, 53-3
ProValueDataTransformSet() function
description  16-10, 53-3
ProValueDataType enum
declaration  16-10
ProValueFree() function
description  16-10
ProVector typedef
declaration  10-4
ProVectorTrfEval() function
description  10-5
  used with assemblies  13-11
ProVerstampAlloc() function  6-10
ProVerstampEqual() function
description  6-11
ProVerstampFree() function
description  6-11
ProVerstampGetStringFree() function  6-10
ProVerstampGetStringGet() function
description  6-10
ProView
drawing views and models  39-10
ProViewMatrixGet() function
description  10-6
  fundamentals  8-6
  used in a code example  10-6
ProViewMatrixSet() function
description  8-6, 10-6
  used in a code example  8-9
ProViewNamesGet() function
description  8-7
ProViewReset() function  8-6
ProViewRetrieve() function
description  8-7
ProViewRotate() function  8-6
ProViewStore() function
description  8-7
  used in a code example  8-8
ProWcharSizeVerify() function
description  1-60
ProWeldCompoundGet
genration 37-2
ProWeldFilletDataGet
genration 37-2
Index - li

ProWorkspaceRegister() function
  description 47-2
  used in a code example 47-3

ProWstringArrayToValueArray() function
  description 16-10

ProWstringFree() function
  description 49-6

ProWstringproarrayFree() function
  description 49-6

ProWstringToString() function
  description 1-60
  ensuring portability 1-62
  used in a code example 1-59

ProXsecGeometryRetrieve function
  description 40-3

PTC_DOCS variable 54-2

PTC_HELP variable 54-3

Pushbutton
  adding to menus 4-4

Pushing and popping menus 4-42

R

Query
  external databases 31-1
  features 12-3

Rays 9-24
Read access
  to weld features 37-2
Read status 12-5
Recycle
  object identifiers 44-4
  redefine features 12-6
Reference-driven patterns 21-5
References
  external
    defined 45-4
  external object 44-12
Refresh
  window 8-3
Regenerate
  2-D sections 23-8
  assembly components 13-11
  relation sets 11-10
  solid objects 7-4
Regeneration types 21-5
Register
  applications 1-31
  external object classes 44-3
  Workspace 47-2
Registry files 1-25

examples B-4
fields of B-2
functions 49-3
toolkit D-7

Relations 11-8

Release
  attachments 7-17
  Release 20.0
    new functions A-2

Release Wildfire
  technical changes A-21

Remove
  animation frames from a movie 46-6
  animation objects from frames 46-4
  external object data 44-11
  external object references 44-13
  family table items 34-6
  highlighting 2-9
  pattern table instance 12-16
  windows 8-4

Rename
  models 6-6
  surfaces
    code example 6-9

Reorder 12-8
Repaint
  windows 8-3

Replace
  section entities 23-4
  tweak surface features 28-1

Reporting errors
  for sections 23-12

Reposition process steps 33-6
Reset
  view 8-6

Resolution
  specifying 23-9

Restart
  applications 1-35

Retrieve
  2-D sections 23-13
  external data 35-5
  family table instances 34-5
  geometry of a simplified representation 38-6
  graphics of a simplified representation 38-6
  models 6-6
  simplified representations 38-5
  views 8-7

Reusable identifiers 44-4

roDlNotedataAttachmentGet() function
  defined 39-35

roDl laysminstCreate() function
  description 39-23

Roles
of manufacturing components 30-5
Rotate views 8-6
Rotation angle text 8-17
Routing 42-19
locations 42-17
Ruled surfaces G-9
Rules simplified representation 38-14
Runners 43-11
Run-time menus 4-43

S
Save 2-D sections 23-13
models 6-6
Scale external objects 44-10
view 39-10
Scope control defined 45-4
Screen coordinate system 10-2
Scroll messages 3-4
Search
APIWizard search feature (Find) 1-11
Sections 2-D
adding entities 23-4
saving 23-13
allocating 23-3
automatic dimensioning 23-10
constraints 23-10
creating 2-D 23-2
example 23-14
models 23-2
definition 23-1
dimensions 23-5
entities 23-4
errors 23-12
example 23-14
regenerating 23-8
saving 23-13
solving 23-8
Select3d
compared to ProSelection D-15
converting D-13
mapping sel_type D-14
Selection 2-2
boxes 44-10
size of 44-11
data
for external objects 44-10
explode states 13-15
family table instances 34-5
feature 12-5
interactive 2-5
Session simplified representations 38-3
Set mode
definition Glossary-6
Set up drawings 39-6
SGI platform requirements 39-6
Swing 1-3
Sheets drawing 39-7
Shell pairs 43-13
Simple asynchronous mode 52-7
Simplified representations adding items 38-13
creating 38-6
deleting 38-6
items 38-13
extracting information from 38-9
modifying 38-13
retrieving geometry 38-6
graphics 38-6
rules 38-14
session 38-3
zones 38-16
Single animation 46-5
Single-valued element 16-4
Size gathering by 38-16
Skeleton Model Functions 45-5
Slant angle text 8-17
Slots 35-2
smbelp.h file 54-2
Solids accuracy 7-7
contents of 7-2
coordinate system 10-2
creating 7-2
displaying 7-3
geometry visiting 9-7
mass properties 7-8
orientation 8-6
outline 7-5
regenerating 7-4
transforming coordinates 10-6
units 7-7
Solve sections 23-8

Index - lii Pro/TOOLKIT User’s Guide
Index - liii

Sources
  light 8-25
Splines
  cylindrical spline surface  G-16
  representation  G-21
  surface G-13
Spools 42-5
Start
  batch animation 46-5
Status
  feature 12-4
Statuses
  of a Pro/ENGINEER process 52-8
  read 12-5
Stop
  applications 1-35
Storage solids 30-3
  identifying 30-6
Store
  external data 35-4
  view 8-7
Stream
  type of slot 35-2
Strings
  converting 1-63
  wide 1-60
  functions for 49-5
Structure
  of applications D-7
Submenus 4-39
  definition Glossary-6
Superseded functions A-20
Support
  third-party tool manager 31-2
Suppress
  feature 12-6
Surfaces
  cylindrical spline  G-16
  data structures G-3
  definition Glossary-6
  fillet G-12
  general surface of revolution  G-8
  manufacturing code example 30-33
  NURBS G-14
  parametric equations 9-20
  properties of 8-22
  renaming
    code examples 6-9
  replacement features 28-1
  ruled G-9
  spline G-13
  traversing F-2
  types 9-20
Sweeps
  code example 26-5, 26-13
  creating 26-1
  element tree 26-3
Swing
  class path
    NT 1-5
    UNIX 1-4
  download Java foundation class archive
    1-4
  in APIWizard 1-3
  required for APIWizard with Netscape 1-3
  required for APIWizards on SGI platforms
    1-3
  required Java Foundation Class for
    APIWizard 1-4
Symbol definitions 39-20
Symbol instances 39-20
Symbols
  manipulating groups 39-52
  Synchronous J-Link Applications 53-9
Synchronous mode
  definition Glossary-7

T
Table-driven patterns 12-16
  creating 21-6
Tables
  drawing 39-55
  family
    code example 34-6
    objects 34-2
    operations on instances 34-4
    operations on items 34-6
    utilities 34-2
  visiting 34-3
Tabulated cylinders G-10
Task Library Functions
  Memory Management 53-4
Tessellation 9-14
  definition Glossary-7
  surface 8-22
Text
  attributes 8-17
  displaying 8-17
  fonts 8-18
  message files 3-5
  message window
    identifier 8-1
  note 7-15
  style properties 7-18
  styles
    of notes 7-17
Tolerance
  geometric
    reference by gtol 15-2
Toolbar
placing 4-22
Toolkit DLL
Create 53-3
Functions 53-8
Task Libraries 53-3
Tools
auxiliary 30-11
contouring 30-11
creating 30-9
holemaking
parameters 30-12
milling
parameters for 30-11
table
creating 30-23
turning
parameters for 30-10
types 30-8
visiting 30-8
Torus G-7
Trail files 49-3
Transformations
coordinates of an assembly member 10-8
coordinates of sketched entities 10-8
definition Glossary-7
drawing view to screen coordinates 10-7
external object 44-8
screen to drawing coordinates 10-7
screen to window coordinates 10-6
solid to screen coordinates 10-6
in a drawing 10-7
to coordinate system datum coordinates 10-8
Trapping 36-1
Traversal
assembly 13-5
gallery F-1
manufacturing components 30-5
part 9-3
Turning tools
parameters 30-10
U
UDFs 12-18
uiCmdPriority enum
declaration 4-6
Undecorated Layout Dialog 5-36
Unit types 7-7
Unlock
messages 1-38
toolkit
creation D-7
toolkit application 1-38
Updates
Release Wildfire A-2
URL
of notes 7-17
use same version as ProENGINEER 1-23
User attributes
definition Glossary-7
user_change_directory() function
equivalent D-27
user_change_window_post() function
equivalent D-27
user_dbms_copy_post_all() function
equivalent D-27
user_dbms_delete_post_all() function
equivalent D-27
user_dbms_erase_post_all() function
equivalent D-27
user_dbms_failure_function() function
equivalent D-27
user_dbms_retieve_post_all() function
equivalent D-27
user_dbms_save_post_all() function
equivalent D-26
user_eval_srf() function 27-2
user_graphics_object_output_post() function
equivalent D-27
user_graphics_object_output_pre() function
equivalent D-27
user_init_srf() function 27-2
user_initialize() function
adding a new menu button 4-28
changing the color map 8-13
description 1-39
user_intf_arc() function 41-2
user_intf_filled_poly() function 41-2
user_intf_line() function 41-2
user_intf_polyline() function 41-2
user_intf_text() function 41-2
user_menu_dbms_copy_post() function
equivalent D-26
user_menu_dbms_copy_pre() function
equivalent D-26
user_menu_dbms_create_post() function
equivalent D-26
user_menu_dbms_create_pre() function
equivalent D-26
user_menu_dbms_delete_post() function
equivalent D-26
user_menu_dbms_delete_pre() function
equivalent D-26
user_menu_dbms_erase_post() function
equivalent D-26
user_menu_dbms_erase_pre() function
equivalent D-26
user_menu_dbms_purge_post() function
equivalent D-26
user_menu_dbms_purge_pre() function
equivalent D-26

Pro/TOOLKIT User’s Guide
Index - lv

- user_menu_dbms_rename_post() function
  equivalent D-26
- user_menu_dbms_rename_pre() function
  equivalent D-26
- user_menu_dbms_retrieve_post() function
  equivalent D-26
- user_menu_dbms_retrieve_pre() function
  equivalent D-26
- user_menu_dbms_save_post() function
  equivalent D-26
- user_menu_dbms_save_pre() function
  equivalent D-26
- user_mfg_feat_cl_file_post() function
  equivalent D-27
- user_mfg_oper_cl_file_post() function
  equivalent D-27
- user_proe_term_func() function
  defined 1-39
- user_rmov_srf() function
  27-2
- user_terminate() function
  defined 1-39
  description 1-40

User's Guide
  browsing with APIWizard 1-10
documentation
  online 1-2
  online format 1-2
  PDF format 1-2

user-initialize()
described 1-41

User-supplied main 1-45
UsuDisppointsCollect()function description 6-18
Utilities 49-1
  family table 34-2

V

Values
  feature element 16-5
  of feature elements 16-9
  parameter 14-3

Vectors
  definition Glossary-7
  Version compatibility 1-33
  Version stamps 6-10
  definition Glossary-7

View matrix 8-6
Views
  definition Glossary-7
  drawing 39-10
  creating 39-16
  listing
    example 39-11
    modifying 39-13
    orientation 8-6

overlay
  definition Glossary-5

scale 39-10

storing 8-7

Visibility
  of assembly components 13-8
  of menu buttons 4-42

Visit
  animation frames 46-4
  animation movie frames 46-6
  assembly components 13-5
  coordinate system datums 9-11
datum axes 9-10
datum curves 9-12
datum planes 9-11
datum points 9-13
  explode states 13-15
  external object references 44-14
  external objects 44-6
  family tables 34-3
  feature geometry 9-5
  features 12-2
  functions
    description 1-56
  geometry objects 9-3
  layers 6-14
  manufacturing tools 30-8
  notes 7-16
  process step features 33-2
  relation sets 11-10
  simplified representations 38-9
  solid geometry 9-7
  windows 8-5

Visiting
  cross sections 40-5
  displayed entities 6-18
gtols 15-3

W

Warnings 44-17
wchar_t (see also Wide strings)

Weld
  drawing symbols 37-3
  notification callbacks 37-3
  symbol notification types 37-3
    PRO_DRAWING_WELD_GROUPIDS_GET 37-3
    PRO_DRAWING_WELD_SYMPATH_G
      ET 37-3
    PRO_DRAWING_WELD_SYMTEXT_G
      ET 37-3

Weld features
  read access to 37-2

WHandles
  description 1-49

Index - lv
Wide strings 1-60
   checking your declaration 1-62
   definition Glossary-7
   functions 49-5
   manipulating 1-62
Width factor
text 8-17
Window coordinate system 10-3
Window matrix 8-6
Windows
   activating 8-5
   closing 8-4
   creating 8-4
   current 8-4
   definition Glossary-7
   deleting 8-4
   getting the owner 8-5
   identifiers 8-1
   information
      definition Glossary-3
   manipulating 8-2
   matrix 8-6
   orientation 8-6
   pan and zoom matrix 8-6
   repainting 8-3
   visiting 8-5
Wires
   spools 42-5
Workcells
   creating 30-22
   example 30-24
   features
      code example 30-6
Workspace 47-2
Workspace handles
description 1-49
Write
   a message to the Message Window 3-4
   family table to a file 34-6
   message to an internal buffer 3-9
Z
Zones 38-16