

**Parametric Technology Corporation**

**Pro/ENGINEER® Wildfire® 4.0  
VB API User's Guide**

**June 2009**

---

**Copyright © 2009 Parametric Technology Corporation and/or Its Subsidiary Companies. All Rights Reserved.**

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

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

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

**For Important Copyright, Trademark, Patent, Licensing and Data Collection Information:** For Windchill products, select **About Windchill** at the bottom of the product page. For InterComm products, on the Help main page, click the link for **Copyright 20xx**. For other products, click **Help > About** on the main menu of the product.

**UNITED STATES GOVERNMENT RESTRICTED RIGHTS LEGEND**

This document and the software described herein are Commercial Computer Documentation and Software, pursuant to FAR 12.212(a)-(b) (OCT'95) or DFARS 227.7202-1(a) and 227.7202-3(a) (JUN'95), and are provided to the US Government under a limited commercial license only. For procurements predating the above clauses, use, duplication, or disclosure by the Government is subject to the restrictions set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software Clause at DFARS 252.227 7013 (OCT'88) or Commercial Computer Software-Restricted Rights at FAR 52.227 19(c)(1)-(2) (JUN'87), as applicable. 01162009

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

---

# About This Guide

---

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

## Topic

[Purpose](#)

[Audience](#)

[Contents](#)

[Prerequisites](#)

[Documentation](#)

[Software Product Concerns and Documentation Comments](#)

## Purpose

This manual describes how to use the VB API, a Visual Basic toolkit for Pro/ENGINEER. The VB API makes possible the development of Visual Basic programs that access the internal components of a Pro/ENGINEER session, to customize Pro/ENGINEER models.

## Audience

This manual is intended for experienced Pro/ENGINEER users who are familiar with Visual Basic or another object-oriented language.

## Contents

This manual contains the chapters that describe how to work with different functions provided by Visual Basic APIs.

## Prerequisites

This manual assumes you have the following knowledge:

- Pro/ENGINEER
- Visual Basic for Applications (Office macros)
- Visual Basic .NET 2005
- Other languages with the built-in capability to use COM servers:
  - JavaScript
  - VB.Script
  - C++
  - C#

## Documentation

The documentation for Visual Basics APIs includes the following:

- The VB API User's Guide.
- An online browser that describes the syntax of the Visual Basic functions and provides a link to the online version of this manual. The online version of the documentation is updated more frequently than the printed version. If there are any discrepancies, the online version is the correct one.

## Conventions

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

Convention	Description
#	The pound sign (#) is the convention used for a UNIX prompt.
UPPERCASE	Pro/ENGINEER-type menu name (for example, PART).
<b>Boldface</b>	Windows-type menu name or menu or dialog box option (for example, <b>View</b> ), or utility. Boldface font is also used for keywords, VB API methods, names of dialog box buttons, and Pro/ENGINEER commands.

Monospace (Courier)	Code samples appear in courier font like this. Java aspects (methods, classes, data types, object names, and so on) also appear in Courier font.
<i>Emphasis</i>	Important information appears <i>in italics like this</i> . Italic font is also used for file names and uniform resource locators (URLs).
Mode	An environment in Pro/ENGINEER in which you can perform a group of closely related functions (Drawing, for example).
Model	An assembly, part, drawing, format, layout, case study, sketch, and so on.
Solid	A part or an assembly.

### Notes:

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

## Software Product Concerns and Documentation Comments

For resources and services to help you with PTC software products, see the *PTC Customer Service Guide*. It includes instructions for using the World Wide Web or fax transmissions for customer support.

In regard to documentation, PTC welcomes your suggestions and comments. You can send feedback in the following ways:

- Send comments electronically to [doc-webhelp@ptc.com](mailto:doc-webhelp@ptc.com).
- Fill out and mail the PTC Documentation Survey in the customer service guide.

---

# Overview of the VB API

---

This section provides an overview of the VB APIs.

## Topic

[Introduction](#)

[Getting Started](#)

[Object Types](#)

[Programming Considerations](#)

## Introduction

The VB API for Pro/ENGINEER Wildfire 4.0 is an asynchronous application that can be used from any COM-enabled application including Visual Basic.NET (VB.NET), Visual Basic for Applications (VBA), and external Internet Explorer instances using scripting.

## Visual Basic.NET Applications

You can use the VB API for Pro/ENGINEER Wildfire 4.0 to:

- Create a VB.NET form capable of starting or connecting to Pro/ENGINEER non-graphically, accepting user inputs and driving model modifications or deliverables.
- Create a VB.NET application that may or may not have its own User Interface (UI). The application should be able to establish one or more Pro/ENGINEER UI or event listeners in session, and process those events using VB.NET code.

## Visual Basic for Applications

The VB APIs provide support for accessing Pro/ENGINEER from Visual Basic-enabled products such as Microsoft Excel, Microsoft Word, or Microsoft Access. The COM interface is provided to control Pro/ENGINEER asynchronously and use the PFC API's to access its properties.

You can also access data from OLE objects embedded in Pro/ENGINEER. The OLE objects can include VB code that can be used to drive the model from which the object is contained.

## Limitations of the VB API

The asynchronous COM server has the following limitations:

- API calls to Pro/ENGINEER should be made only from a single thread. Other threads can process non Pro/ENGINEER data and set data to be seen by the Pro/ENGINEER thread, but only one thread can communicate with Pro/ENGINEER.
- Only one active connection can be made to a single Pro/ENGINEER session at one time.

## Getting Started

### Setting Up a VB Application

For your application to communicate with Pro/ENGINEER, you must set the `PRO_COMM_MSG_EXE` environment variable to the full path of the executable, `pro_comm_msg.exe`. Typically, the path to the executable is `[Pro/E loadpoint]/[machine type]/obj/pro_comm_msg.exe`, where `machine type` is `i486_nt` for 32-bit Windows and `x86e_win64` for 64-bit Windows installations.

Set PRO\_COMM\_MSG\_EXE as:

1. Click Start > Settings > Control Panel
2. Click **System**. The **System Properties** windows opens.
3. In the **Advanced** tab, click the **Environment Variables** button.
4. Add PRO\_COMM\_MSG\_EXE to **System variables**.

## Registering the COM Server

To register the COM server, run the vb\_api\_register.bat file located at [proe\_loadpoint]/bin.

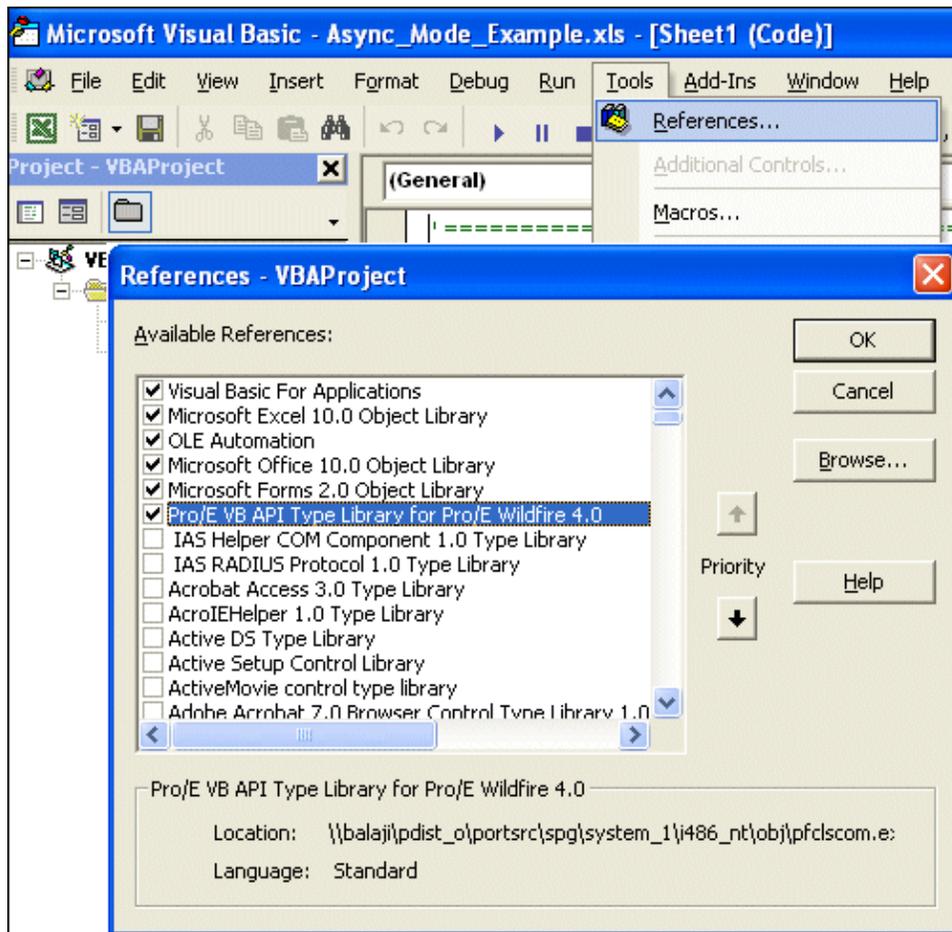
To unregister the COM server, run the vb\_api\_unregister.bat file located at [proe\_loadpoint]/bin.

After the COM server is registered with the system, whenever an application tries to access the types contained in this server the server starts automatically. By default, Windows starts services such as pfclscom.exe in the Windows system directory (c:\winnt\system\_32). Because the server will also start new sessions of Pro/ENGINEER from the process working directory, you may want to control the server run directory. You can configure the server to start in a specific directory by setting the system environment variable PFCLS\_START\_DIR to any existing directory on your computer.

## Setting Project References for the VB API

Set the reference to Pro/E VB API Type Library for Pro/E Wildfire 4.0 through your project. In the VBA environment set this reference as follows:

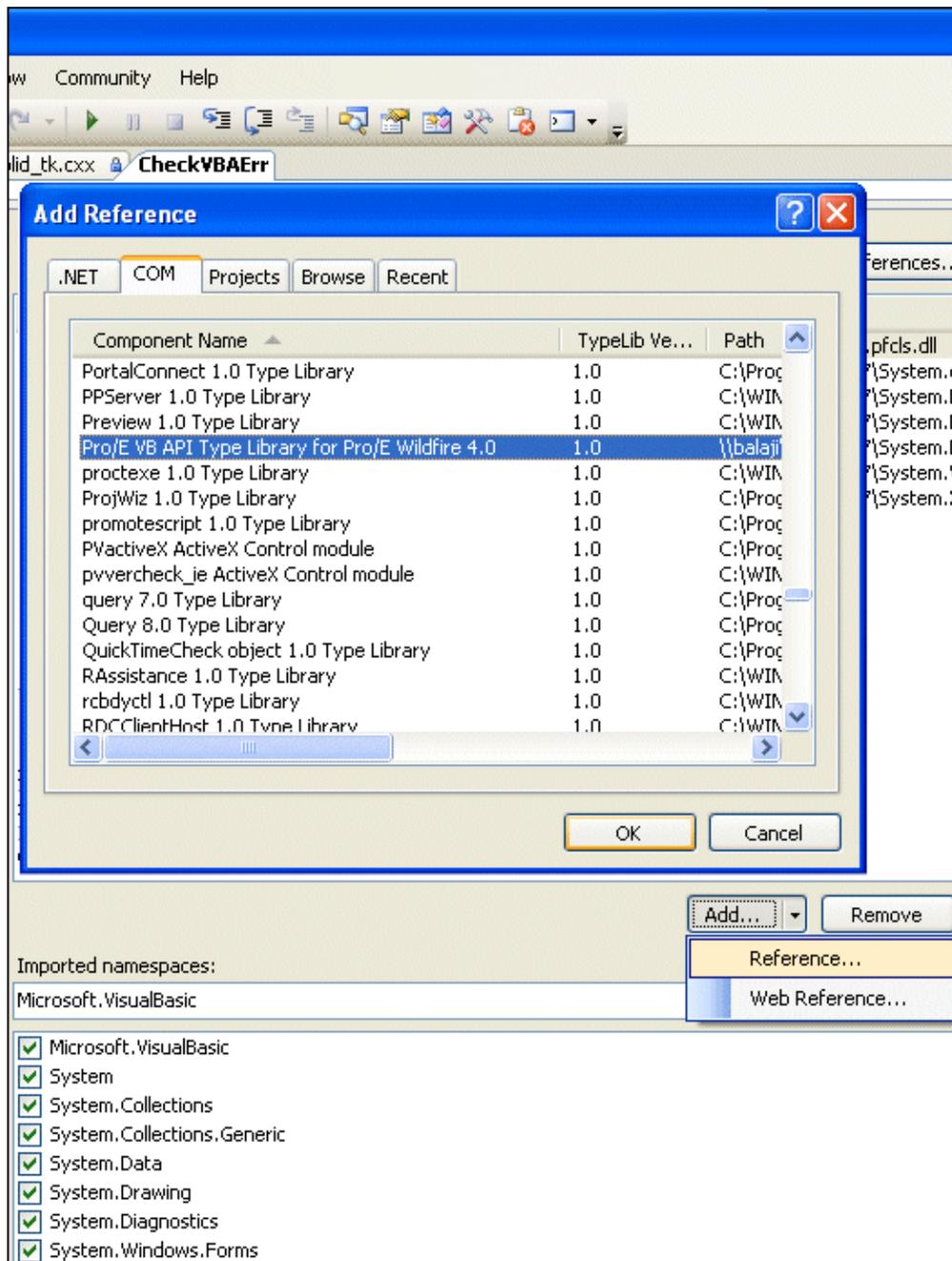
1. Click Tools>References
2. Check the box for **Pro/E VB API Type Library for Pro/E Wildfire 4.0** as shown in the following figure.





In the VB.NET environment, set this reference as follows:

1. Click Project>Properties>Add Reference>COM
2. Check the box for **Pro/E VB API Type Library for Pro/E Wildfire 4.0** as shown in the following figure.



## Object Types

The VB API is made up of a number of classes in many modules. The following are the main class types:

- o Pro/ENGINEER-Related Classes--Contain unique methods and properties that are directly related to the functions in Pro/ENGINEER. See the section "Pro/ENGINEER-Related Classes" for more information.
- o Compact Data Classes--Classes containing data needed as arguments to some VB methods. See the section, "Compact Data Classes", for additional information.
- o Union Classes--Classes with a potential to contain multiple types of values. See the section "Unions" for additional information.

- o Sequence Classes--Expandable arrays of objects or primitive data types. See the section "Sequences" for more information.
- o Array Classes--Arrays that are limited to a certain size. See the section "Arrays" for more information.
- o Enumeration Classes--Enumerated types, which list a restricted and valid set of options for the property. See the section "Enumeration Classes" for more information.
- o Module-Level Classes--Contain static methods used to initialize certain VB objects. See the "Module-Level Classes" section for more information.
- o ActionListener Classes--Enable you to specify code that will run only if certain events in Pro/ENGINEER take place. See the Action Listeners section for more information.

Each class shares specific rules regarding initialization, attributes, methods, inheritance, or exceptions. The following sections describe these classes in detail.

## Pro/ENGINEER-Related Classes

The Pro/ENGINEER-Related Classes contain methods that directly manipulate objects in Pro/ENGINEER. Examples of these objects include models, features, and parameters.

### Initialization

You cannot construct one of these objects using the keyword `New`. Instead, you should obtain the handle to a Pro/ENGINEER-related object by creating or listing that object with a method on the parent object in the hierarchy.

For example, **`IpfcBaseSession.CurrentModel`** returns a `IpfcModel` object set to the current model and **`IpfcParameterOwner.CreateParam`** returns a newly created parameter object for manipulation.

### Properties

Properties within Pro/ENGINEER-related objects are directly accessible. Some attributes that have been designated as read-only can be accessed but not modified by the VB API.

### Methods

You must invoke methods from the object in question and first initialize that object. For example, the following calls are illegal:

```
Dim window as pfcls.IpfcWindow;
window.Activate();           ` The window has not yet
                             ` been initialized.
Repaint();                   ` There is no invoking object.
```

The following calls are legal:

```
Dim window As Pfcls.IpfcWindow
Dim session as pfcls.IpfcSession
Dim asyncConnection as pfcls.IpfcAsyncConnection
Dim Casync as New pfcls.CCpfcAsyncConnection

asyncConnection = Casync.Connect (DBNull.Value, DBNull.Value, DBNull.Value, DBNull.
Value)
session = asyncConnection.Session;
window = session.CurrentWindow;   ` You have initialized
                                  ` the window object.

window.Activate()
window.Repaint()
```

### Inheritance

Many Pro/ENGINEER-related objects inherit methods from other interfaces. In VB.NET and VBA, you must have an object

of the correct type for the compiler and IDE to resolve the methods you wish to call. For example, an `IpfcComponentFeat` object could use the methods and properties as follows:

- o `IpfcObject`
- o `IpfcChild`
- o `IpfcActionSource`
- o `IpfcModelItem`
- o `IpfcFeature`
- o `IpfcComponentFeat`

The following are the approaches to using an object's inherited methods:

1. You can code the method call directly even though it is not available in Intellisense.

```
Dim componentFeat as pfcls.IpfcComponentFeat
MsgBox ("Feature number: " & componentFeat.Number);
```

**Note:**

This works in VB.NET but is likely to result in a compilation error in VBA.

2. You can create another object of the appropriate type and assign it the object handle, and then call the required method.

```
Dim componentFeat as pfcls.IpfcComponentFeat
Dim feat as pfcls.IpfcFeature

feat = componentFeat
MsgBox ("Feature number: " & feat.Number);
```

## Compact Data Classes

Compact data classes are data-only classes. They are used for arguments and return values for some VB API methods. They do not represent actual objects in Pro/ENGINEER. Other than a difference in how they are initialized, compact data classes have similar requirements to Pro/ENGINEER-related classes.

### Initialization

You can create these compact data objects using a designated `Create` method which resides on the CC version of the compact class. You instantiate the CC class object with the keyword `New`.

For example,

```
'Class object, owns Create()
Dim tableCellCreate As New pfcls.CCpfcTableCell
Dim tableCell As pfcls.IpfcTableCell
Set tableCell = tableCellCreate.Create(1, 1)
```

## Unions

Unions are classes containing potentially several different value types. Every union has a discriminator property with the predefined name, `discr`. This property returns a value identifying the type of data that the union object holds. For each union member, a separate property is used to access the different data types. It is illegal to attempt to read any property except the one that matches the value returned from the discriminator. However, any property that switches the discriminator to the new value type can be modified.

The following is an example of a VB API union:

## Interface IpfcParamValue

---

### Description

This class describes the value of the parameter.

---

### Union Discriminant

Property `discr` as `IpfcParamValueType` [readonly]

Returns the union discriminant value.

---

### Property Summary

Property `BoolValue` as `Boolean`

If the parameter type is `PARAM_BOOLEAN`, this is a Boolean value.

Property `DoubleValue` as `Double`

If the parameter type is `PARAM_DOUBLE`, this is a double value.

Property `IntValue` as `Long`

If the parameter type is `PARAM_INTEGER`, this is an integer value.

Property `NoteId` as `Long`

If the parameter type is `PARAM_NOTE`, this is a note identifier.

Property `StringValue` as `String`

If the parameter type is `PARAM_STRING`, this is a string value.

## Sequences

Sequences are expandable arrays of primitive data types or objects in the VB API. All sequence classes have the same methods for adding to and accessing the array. Sequence classes are typically identified by a plural name, or the suffix `seq`.

## Initialization

You can create instances of these classes directly by instantiating the appropriate class object:

## Properties

The read-only `Count` attribute identifies how many members are currently in the sequence. You may also access members of the sequence using the `Item` property or directly:

```
Dim model as IpfcModel
model = models (0)
```

## Methods

Sequence objects always contain the same methods. Use the following methods to access the contents of the sequence:

- `Append()`--Adds a new item to the end of the array
- `Clear()`--Removes all items from the array
- `Insert()`--Inserts a new item at any location of the array
- `InsertSeq()`--Inserts the contents of a sequence of items at any location of the array
- `Set()`--Assigns one item in the array to the input item
- `Remove()`--Removes a range of items from the array

## Inheritance

Sequence classes do not inherit from any other VB API classes. Therefore, you cannot use sequence objects as arguments where any other type of VB API object is expected, including other types of sequences. For example, if you have a list of `IpfcModelItems` that happen to be features, you cannot use the sequence as if it were a sequence of `IpfcFeatures`.

To construct the array of features, you must insert each member of the `IpfcModelItems` list into the new `IpfcFeatures` list.

## Arrays

Arrays are groups of primitive types or objects of a specified size. An array can be one- or two- dimensional. The online reference documentation indicates the exact size of each array class.

### Initialization

You can create instances of these classes directly by instantiating the appropriate class object:

### Properties

You may read members of the sequence using the `Item` property or directly as an array:

```
Dim point as IpfcPoint3D
Dim matrix as IpfcMatrix3D
MsgBox ("Y value of point: " & point.Item (1))
MsgBox ("(2, 2) value of matrix: " & matrix (2, 2))
```

### Methods

Array objects contain only the `Set` method, which assigns one item in the array to the input item.

## Enumeration Classes

In the VB API, an enumeration class defines a limited number of values that correspond to the members of the enumeration. Each value represents an appropriate type and may be accessed by name. In the `EpfcFeatureType` enumeration class, the value `EpfcFEATTYPE_HOLE` represents a Hole feature in Pro/ENGINEER. Enumeration classes in the VB API generally have names of the form `EpfcXYZType` or `EpfcXYZStatus`.

### Initialization

You can directly refer to instance of this class:

```
Dim type as EpfcFeatureType
type = EpfcFeatureType.EpfcFEATTYPE_HOLE
```

### Attributes

An enumeration class is made up of constant integer properties. The names of these properties are all uppercase and describe what the attribute represents. For example:

- `EpfcPARAM_INTEGER`--A value in the `EpfcParamValueType` enumeration class that is used to indicate that a parameter stores an integer value.
- `EpfcITEM_FEATURE`--An value in the `EpfcModelItemType` enumeration class that is used to indicate that a model item is a feature.

An enumeration class always has an integer value named `<type>_nil`, which is one more than the highest acceptable numerical value for that enumeration class.

## Module-Level Classes

Some modules in the VB API have one class that contains special functions used to create and access some of the other classes in the package. These module classes have the naming convention, `CM+` the name of the module, for example `CMpfcSelect`.

### Initialization

You can create instances of these classes directly by instantiating the appropriate class object:

```
Dim mSelect as New CMpfcSelect
```

## Methods

Module-level classes contain only static methods used for initializing certain VB API objects.

## Action Listeners

Action Listeners notify you of events in Pro/ENGINEER. They are also the basis for customization of the Pro/ENGINEER User Interface. ActionListeners are not supported from VBA.

## Initialization

In VB.NET, you can create and assign an ActionListener class as follows.

Create a class implementing the listener in question. It should define all the inherited methods, even if you want to only execute code for a few of the listener methods. Those other methods should be implemented with an empty body.

The class should also implement the interface `IpfcActionListener`, which has no methods.

The class should also implement `ICIPClientObject`. This method defines the object type to the CIP code in the server. This method returns a String which is the name of the listener type interface, for example, `IpfcSessionActionListener`.

```
Private Class ModelEventListener
    Implements IpfcModelEventActionListener
    Implements ICIPClientObject
    Implements IpfcActionListener

    Public Function GetClientInterfaceName() As String _
        Implements ICIPClientObject.GetClientInterfaceName
        GetClientInterfaceName = "IpfcModelEventActionListener"
    End Function
'=====
'Function      :   OnAfterModelCopy
'Purpose      :   This method is executed after successfully
'               copying a model.
'=====
    Public Sub OnAfterModelCopy(ByVal _FromMdl As
        pfcls.IpfcModelDescriptor, ByVal _ToMdl As
        pfcls.IpfcModelDescriptor) Implements
        pfcls.IpfcModelEventActionListener.OnAfterModelCopy
'Method Body
    End Sub
'=====
'Function      :   OnAfterModelRename
'Purpose      :   This method is executed after successfully
'               renaming a model.
'=====
    Public Sub OnAfterModelRename(ByVal _FromMdl As
        pfcls.IpfcModelDescriptor, ByVal _ToMdl As
        pfcls.IpfcModelDescriptor) Implements
        pfcls.IpfcModelEventActionListener.OnAfterModelRename
'Method Body

    End Sub
```

```

Public Sub OnAfterModelCopyAll(ByVal _FromMdl As
    pfcls.IpfcModelDescriptor, ByVal _ToMdl As
    pfcls.IpfcModelDescriptor) Implements
    pfcls.IpfcModelEventListener.OnAfterModelCopyAll

End Sub

Public Sub OnAfterModelDelete(ByVal _Descr As
    pfcls.IpfcModelDescriptor) Implements
    pfcls.IpfcModelEventListener.OnAfterModelDelete

End Sub

Public Sub OnAfterModelErase(ByVal _Descr As
    pfcls.IpfcModelDescriptor) Implements
    pfcls.IpfcModelEventListener.OnAfterModelErase

End Sub

End Class

```

## Exceptions

Action listeners cause methods to be called outside of your application start and stop methods. Therefore, you must include exception-handling code inside the `ActionListener` implementation if you want to respond to exceptions. In some methods called before an event, propagating a `pfcXCancelProEAction` exception out of your method will cancel the impending event.

## Programming Considerations

The items in this section introduce programming tips and techniques used for programming with the VB API .

### Application Hierarchy

The rules of object orientation require a certain hierarchy of object creation when you start a VB application. The application must iterate down to the level of the object you want to access. For example, to list all the datum axes contained in the hole features in all models in session, do the following:

1. Use the method `CCpfcAsyncConnection.Connect` to connect to an existing session of Pro/ENGINEER.

```

Dim connection As IpfcAsyncConnection
Dim classAsyncConnection As New CCpfcAsyncConnection
connection = classAsyncConnection.Connect (DBNull.Value, DBNull.Value, DBNull.Value,
    DBNull.Value)

```

2. Get a handle to the session of Pro/ENGINEER for the current active connection:

```

Dim session As IpfcBaseSession

session = connection.Session

```

3. Get the models that are loaded in the session:

```

Dim models As IpfcModels

models = session.ListModels()

```

4. Get the handle to the first model in the list:

```
Dim model As IpfcModel
```

```
model = models[0]
```

5. Get the feature model items in each model:

```
Dim items As IpfcModelItems
```

```
items = model.ListItems (EpfcModelItemType.EpfcITEM_FEATURE)
```

6. Filter out the features of type hole:

```
if (feature.FeatType = EpfcFeatureType.EpfcFEATTYPE_HOLE) then
```

7. Get the subitems in each feature that are axes:

```
Dim axes As IpfcModelItems
```

```
axes = feature.ListSubItems (EpfcModelItemType.EpfcITEM_AXIS)
```

## Optional Arguments and Tags

Many methods in the VB API are shown in the online documentation as having optional arguments.

For example, the **IpfcModelItemOwner.ListItems()** method takes an optional *Type* argument.

```
IpfcModelItems ListItems (Type as IpfcModelItemType [optional]);
```

In VB.Net, you can pass the keyword `Nothing` in place of any such optional argument. In VBA, use `Null` in place of any such optional argument. The VB API methods that take optional arguments provide default handling for `Nothing` parameters which is described in the online documentation.

### Note:

You can only pass `Nothing` in place of arguments that are shown in the documentation to be optional.

## Optional Returns for the VB API Methods

Some methods in the VB API have an optional return. Usually these correspond to lookup methods that may or may not find an object to return. For example, the **pfcBaseSession.GetModel** method returns an optional model:

```
Function GetModel (Name as String, Type as IpfcModelType) as IpfcModel [optional]
```

The VB API might return `Nothing` in certain cases where these methods are called. You must use appropriate value checks in your application code to handle these situations.

## Parent-Child Relationships between the VB API Objects

Some VB API objects inherit from either the interface `IpfcObject.Parent` or `IpfcObject.Child`. These interfaces are used to maintain a relationship between the two objects. This has nothing to do with object-oriented inheritance, but rather, refers to the relationship between the items in Pro/ENGINEER. In the VB API, the `Child` is owned by the `Parent`.

Property Introduced:

- **IpfcChild.DBParent**

The **IpfcChild.DBParent** property returns the owner of the child object. The application developer must know the expected type of the parent in order to use it in later calls. The following table lists parent/child relationships in the VB API.

Parent	Child
IpfcSession	IpfcModel
IpfcSession	IpfcWindow
IpfcModel	IpfcModelItem
IpfcSolid	IpfcFeature
IpfcModel	IpfcParameter
IpfcModel	IpfcExternalDataAccess
IpfcPart	IpfcMaterial
IpfcModel	IpfcView
IpfcModel2D	IpfcView2D
IpfcSolid	IpfcXSection
IpfcSession	IpfcDll (Pro/TOOLKIT)
IpfcSession	IpfcJLinkApplication (J-Link)

## Run-Time Type Identification in the VB API

The VB API and Visual Basic provide several methods to identify the type of an object.

Many VB API classes provide read access to a type enumerated class. For example, the `IpfcFeature` class has a **IpfcFeature.FeatType** property, returning a `pfcFeatureType` enumeration value representing the type of the feature. Based upon the type, a user can recognize that the `IpfcFeature` object is actually a particular subtype, such as `IpfcComponentFeat`, which is an assembly component.

## Support for Embedded OLE Objects

OLE objects, when activated by the user, can include VB code that can be used to drive the model from which the object is contained. The VB API provides a special property in embedded Microsoft Word, Microsoft Excel and Microsoft PowerPoint documents that can directly return the connection ID of the Pro/ENGINEER session that launched the process containing the OLE object. For information about getting the connection ID from the container, refer to the [VB API Fundamentals: Controlling Pro/ENGINEER](#) section.

The user application code embedded in the OLE object passes the connection ID string to **CCpfcConnectionId.Create()** and **CCpfcAsyncConnection.ConnectById()** to establish the connection. The code may then obtain the owner model of the OLE object by retrieving the current model from the session using standard PFC APIs.

For example,

```
Dim ls As New pfcls.CCpfcAsyncConnection
Dim aC As pfcls.IpfcAsyncConnection
Dim cId As New pfcls.CCpfcConnectionId
Dim id As pfcls.IpfcConnectionId
Dim session As pfcls.IpfcBaseSession
Dim model As pfcls.IpfcModel

Set id = cId.Create(connectionId)
Set aC = ls.ConnectById(id, DBNull.Value, DBNull.Value)

Set session = aC.Session
Set model = session.CurrentModel
```

## Exceptions

All PFC methods that fail may throw exceptions as `System.Runtime.InteropServices.COMException`.

The type of the exception can be obtained from the `Message` property of this exception.

```
Try
    session.SetConfigOption("no_way", "no_how")
Catch ex As Exception
    MsgBox(ex.Message) 'Will show pfcExceptions::XToolkitNotFound
End Try
```

The `Description` property returns the full exception description as `[Exception type]; [additional details]`. The exception type is the module and exception name, for example, `pfExceptions::XToolkitCheckoutConflict`.

The additional details include information that was contained in the exception when it was thrown by the PFC layer, such as conflict descriptions for exceptions caused by server operations and error details for exceptions generated during drawing creation.

## PFC Exceptions

The PFC exceptions are thrown by the classes that make up the VB API's public interface. The following table describes these exceptions.

Exception	Purpose
<code>pfExceptions::XBadExternalData</code>	An attempt to read contents of an external data object that has been terminated.
<code>pfExceptions::XBadGetArgValue</code>	Indicates attempt to read the wrong type of data from the <code>IpfcArgValue</code> union.
<code>pfExceptions::XBadGetExternalData</code>	Indicates attempt to read the wrong type of data from the <code>IpfcExternalData</code> union.

pfcExceptions::XBadGetParamValue	Indicates attempt to read the wrong type of data from the IpfcParamValue union.
pfcExceptions::XBadOutlineExcludeType	Indicates an invalid type of item was passed to the outline calculation method.
pfcExceptions::XCancelProEAction	This exception type will not be thrown by VB API methods, but you may instantiate and throw this from certain ActionListener methods to cancel the corresponding action in Pro/ENGINEER.
pfcExceptions::XCannotAccess	The contents of a VB API object cannot be accessed in this situation.
pfcExceptions::XEmptyString	An empty string was passed to a method that does not accept this type of input.
pfcExceptions::XInvalidEnumValue	Indicates an invalid value for a specified enumeration class.
pfcExceptions::XInvalidFileName	Indicates a file name passed to a method was incorrectly structured.
pfcExceptions::XInvalidFileType	Indicates a model descriptor contained an invalid file type for a requested operation.
pfcExceptions::XInvalidModelItem	Indicates that the item requested to be used is no longer usable (for example, it may have been deleted).
pfcExceptions::XInvalidSelection	Indicates that the IpfcSelection passed is invalid or is missing a needed piece of information. For example, its component path, drawing view, or parameters.
pfcExceptions::XJLinkApplicationException	Contains the details when an attempt to call code in an external J-Link application failed due to an exception.
pfcExceptions::XJLinkApplicationInactive	Unable to operate on the requested IpfcJLinkApplication object because it has been shut down.
pfcExceptions::XJLinkTaskNotFound	Indicates that the J-Link task with the given name could not be found and run.
pfcExceptions::XModelNotInSession	Indicates that the model is no longer in session; it may have been erased or deleted.
pfcExceptions::XNegativeNumber	Numeric argument was negative.
pfcExceptions::XNumberTooLarge	Numeric argument was too large.
pfcExceptions::XProEWasNotConnected	The Pro/ENGINEER session is not available so the operation failed.
pfcExceptions::XSequenceTooLong	Sequence argument was too long.
pfcExceptions::XStringTooLong	String argument was too long.

pfcExceptions::XUnimplemented	Indicates unimplemented method.
pfcExceptions::XUnknownModelExtension	Indicates that a file extension does not match a known Pro/ENGINEER model type.

## Pro/TOOLKIT Errors

The **XToolkitError** exception types provide access to error codes from Pro/TOOLKIT functions that the VB API uses internally and to the names of the functions returning such errors. **XToolkitError** is the exception you are most likely to encounter because the VB API is built on top of Pro/TOOLKIT. The following table lists the **XToolkitError** types method and shows the corresponding Pro/TOOLKIT constant that indicates the cause of the error.

<b>XToolkitError Child Class</b>	<b>Pro/TOOLKIT Error</b>	<b>#</b>
pfcExceptions::XToolkitGeneralError	PRO_TK_GENERAL_ERROR	-1
pfcExceptions::XToolkitBadInputs	PRO_TK_BAD_INPUTS	-2
pfcExceptions::XToolkitUserAbort	PRO_TK_USER_ABORT	-3
pfcExceptions::XToolkitNotFound	PRO_TK_E_NOT_FOUND	-4
pfcExceptions::XToolkitFound	PRO_TK_E_FOUND	-5
pfcExceptions::XToolkitLineTooLong	PRO_TK_LINE_TOO_LONG	-6
pfcExceptions::XToolkitContinue	PRO_TK_CONTINUE	-7
pfcExceptions::XToolkitBadContext	PRO_TK_BAD_CONTEXT	-8
pfcExceptions::XToolkitNotImplemented	PRO_TK_NOT_IMPLEMENTED	-9
pfcExceptions::XToolkitOutOfMemory	PRO_TK_OUT_OF_MEMORY	-10
pfcExceptions::XToolkitCommError	PRO_TK_COMM_ERROR	-11
pfcExceptions::XToolkitNoChange	PRO_TK_NO_CHANGE	-12
pfcExceptions::XToolkitSuppressedParents	PRO_TK_SUPP_PARENTS	-13

pfcExceptions::XToolkitPickAbove	PRO_TK_PICK_ABOVE	- 14
pfcExceptions::XToolkitInvalidDir	PRO_TK_INVALID_DIR	- 15
pfcExceptions::XToolkitInvalidFile	PRO_TK_INVALID_FILE	- 16
pfcExceptions::XToolkitCantWrite	PRO_TK_CANT_WRITE	- 17
pfcExceptions::XToolkitInvalidType	PRO_TK_INVALID_TYPE	- 18
pfcExceptions::XToolkitInvalidPtr	PRO_TK_INVALID_PTR	- 19
pfcExceptions::XToolkitUnavailableSection	PRO_TK_UNAV_SEC	- 20
pfcExceptions::XToolkitInvalidMatrix	PRO_TK_INVALID_MATRIX	- 21
pfcExceptions::XToolkitInvalidName	PRO_TK_INVALID_NAME	- 22
pfcExceptions::XToolkitNotExist	PRO_TK_NOT_EXIST	- 23
pfcExceptions::XToolkitCantOpen	PRO_TK_CANT_OPEN	- 24
pfcExceptions::XToolkitAbort	PRO_TK_ABORT	- 25
pfcExceptions::XToolkitNotValid	PRO_TK_NOT_VALID	- 26
pfcExceptions::XToolkitInvalidItem	PRO_TK_INVALID_ITEM	- 27
pfcExceptions::XToolkitMsgNotFound	PRO_TK_MSG_NOT_FOUND	- 28
pfcExceptions::XToolkitMsgNoTrans	PRO_TK_MSG_NO_TRANS	- 29

pfcExceptions::XToolkitMsgFmtError	PRO_TK_MSG_FMT_ERROR	- 30
pfcExceptions::XToolkitMsgUserQuit	PRO_TK_MSG_USER_QUIT	- 31
pfcExceptions::XToolkitMsgTooLong	PRO_TK_MSG_TOO_LONG	- 32
pfcExceptions::XToolkitCantAccess	PRO_TK_CANT_ACCESS	- 33
pfcExceptions::XToolkitObsoleteFunc	PRO_TK_OBSOLETE_FUNC	- 34
pfcExceptions::XToolkitNoCoordSystem	PRO_TK_NO_COORD_SYSTEM	- 35
pfcExceptions::XToolkitAmbiguous	PRO_TK_E_AMBIGUOUS	- 36
pfcExceptions::XToolkitDeadLock	PRO_TK_E_DEADLOCK	- 37
pfcExceptions::XToolkitBusy	PRO_TK_E_BUSY	- 38
pfcExceptions::XToolkitInUse	PRO_TK_E_IN_USE	- 39
pfcExceptions::XToolkitNoLicense	PRO_TK_NO_LICENSE	- 40
pfcExceptions:: XToolkitBsplUnsuitableDegree	PRO_TK_BSPL_UNSUITABLE_ DEGREE	- 41
pfcExceptions::XToolkitBsplNonStdEndKnots	PRO_TK_BSPL_NON_STD_END_ KNOTS	- 42
IpfcXToolkitBsplMultiInnerKnots	PRO_TK_BSPL_MULTI_INNER_ KNOTS	- 43
IpfcXToolkitBadSrfCrv	PRO_TK_BAD_SRF_CRV	- 44
IpfcXToolkitEmpty	PRO_TK_EMPTY	- 45

IpfcXToolkitBadDimAttach	PRO_TK_BAD_DIM_ATTACH	- 46
IpfcXToolkitNotDisplayed	PRO_TK_NOT_DISPLAYED	- 47
IpfcXToolkitCantModify	PRO_TK_CANT_MODIFY	- 48
IpfcXToolkitCheckoutConflict	PRO_TK_CHECKOUT_CONFLICT	- 49
IpfcXToolkitCreateViewBadSheet	PRO_TK_CRE_VIEW_BAD_SHEET	- 50
IpfcXToolkitCreateViewBadModel	PRO_TK_CRE_VIEW_BAD_MODEL	- 51
IpfcXToolkitCreateViewBadParent	PRO_TK_CRE_VIEW_BAD_P ARENT	- 52
IpfcXToolkitCreateViewBadType	PRO_TK_CRE_VIEW_BAD_TYPE	- 53
IpfcXToolkitCreateViewBadExplode	PRO_TK_CRE_VIEW_BAD_ EXPLODE	- 54
IpfcXToolkitUnattachedFeats	PRO_TK_UNATTACHED_FEATS	- 55
IpfcXToolkitRegenerateAgain	PRO_TK_REGEN_AGAIN	- 56
IpfcXToolkitDrawingCreateErrors	PRO_TK_DWGCREATE_ERRORS	- 57
IpfcXToolkitUnsupported	PRO_TK_UNSUPPORTED	- 58
IpfcXToolkitNoPermission	PRO_TK_NO_PERMISSION	- 59
IpfcXToolkitAuthenticationFailure	PRO_TK_AUTHENTICATION_FAILURE	- 60
IpfcXToolkitAppNoLicense	PRO_TK_APP_NO_LICENSE	- 92

IpfcXToolkitAppExcessCallbacks	PRO_TK_APP_XS_CALLBACKS	- 93
IpfcXToolkitAppStartupFailed	PRO_TK_APP_STARTUP_FAIL	- 94
IpfcXToolkitAppInitialization Failed	PRO_TK_APP_INIT_FAIL	- 95
IpfcXToolkitAppVersionMismatch	PRO_TK_APP_VERSION_ MISMATCH	- 96
IpfcXToolkitAppCommunication Failure	PRO_TK_APP_COMM_FAILURE	- 97
IpfcXToolkitAppNewVersion	PRO_TK_APP_NEW_VERSION	- 98

The exception `XProdevError` represents a general error that occurred while executing a Pro/DEVELOP function and is equivalent to an **`pfExceptions::XToolkitGeneralError`** exception.

The **`pfExceptions::XExternalDataError`** exception types and its children are thrown from External Data methods. See the section on [External Data](#) for more information.

---

# VB API Fundamentals:Controlling Pro/ENGINEER

---

This section explains how to use the VB API to establish a connection to Pro/ENGINEER.

## Topic

[Overview](#)

[Simple Asynchronous Mode](#)

[Starting and Stopping Pro/ENGINEER](#)

[Connecting to a Pro/ENGINEER Process](#)

[Full Asynchronous Mode](#)

[Troubleshooting VB API Applications](#)

## Overview

Asynchronous mode is a multiprocess mode in which the VB API application and Pro/ENGINEER can perform concurrent operations. The VB API application (containing its own main() method) is started independently of Pro/ENGINEER and subsequently either starts or connects to a Pro/ENGINEER process. 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, the VB API cannot plant listeners to be notified when events happen in Pro/ENGINEER. Consequently, Pro/ENGINEER cannot invoke the methods that must be supplied when you add, for example, menu buttons to Pro/ENGINEER.

Despite this limitation, a simple asynchronous mode application can be used to automate processes in Pro/ENGINEER. The application may either start or connect to an existing Pro/ENGINEER session, and may access Pro/ENGINEER in interactive or in a non graphical, non interactive mode. When Pro/ENGINEER is running with graphics, it is an interactive process available to the user.

When you design a VB API application to run in simple asynchronous mode, keep the following points in mind:

- The Pro/ENGINEER process and the application perform operations concurrently.
- None of the application's listener methods can be invoked by Pro/ENGINEER.

## Starting and Stopping Pro/ENGINEER

The following methods are used to start and stop Pro/ENGINEER when using the VB API applications.

Methods Introduced:

- **CCpfcAsyncConnection.Start()**
- **lpfcAsyncConnection.End()**

A VB application can spawn and connect to a Pro/ENGINEER process with the method **CCpfcAsyncConnection.Start()**. After this method returns the asynchronous connection object, the VB API application can call the Pro/ENGINEER process using the appropriate APIs. In the interactive mode, you can also access the Pro/ENGINEER session when it is running.

The asynchronous application is not terminated when Pro/ENGINEER terminates. This is useful when the application needs

to perform Pro/ENGINEER operations intermittently, and therefore, must start and stop Pro/ENGINEER more than once during a session.

The application can connect to or start only one Pro/ENGINEER session at any time. If the VB API application spawns a second session, connection to the first session is lost.

To end any Pro/ENGINEER process that the application is connected to, call the method **IpfcAsyncConnection.End()**.

## Setting Up a Noninteractive Session

You can spawn a Pro/ENGINEER session that is both noninteractive and nongraphical. In asynchronous mode, include the following strings in the Pro/ENGINEER start or connect call to **CCpfcAsyncConnection.Start()**:

- o -g:no\_graphics--Turn off the graphics display.
- o -i:rpc\_input--Causes Pro/ENGINEER to expect input from your asynchronous application only.

### Note:

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:

```
Dim aC as IpfcAsyncConnection
Dim ccAC as New CcpfcAsyncConnection
aC = ccAC.Start ("pro -g:no_graphics -i:rpc_input", <text_dir>);
```

where pro is the command to start Pro/ENGINEER.

## Example Code for Visual Basic.NET

This example demonstrates how to use the VB API to start Pro/ENGINEER asynchronously, retrieve a Session and to open a model in Pro/ENGINEER.

```
Imports pfcls
Public Class pfcAsynchronousModeExamples

    Public Sub runProE(ByVal exePath As String, ByVal workDir As String)

        Dim asyncConnection As IpfcAsyncConnection = Nothing
        Dim cAC As CCpfcAsyncConnection
        Dim session As IpfcBaseSession

        Try

            '=====
            'First Argument : The path to the Pro/E executable along with command
            'line options. -i and -g flags make Pro/ENGINEER run in non-graphic,
            'non-interactive mode
            'Second Argument: String path to menu and message files.
            '=====
            cAC = New CCpfcAsyncConnection
            asyncConnection = cAC.Start(exePath + " -g:no_graphics
                -i:rpc_input", ".")
            session = asyncConnection.Session
            '=====
            'Set working directory
```

```

'=====
      asyncConnection.Session.ChangeDirectory(workDir)
'=====
'VB api process calls and other processing to be done
'=====
      Dim descModel As IpfcModelDescriptor
      Dim model As IpfcModel

      descModel = (New CCpfcModelDescriptor).Create
                  (EpfcModelType.EpfcMDL_PART, _"partModel.prt", Nothing)
      model = session.RetrieveModel(descModel)
Catch ex As Exception
      MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
Finally
'=====
'End the Pro/ENGINEER session when done
'=====

      If Not asyncConnection Is Nothing AndAlso
          asyncConnection.IsRunning Then
          asyncConnection.End()
      End If
End Try
End Sub

End Class

```

## Example Code for Visual Basic for Applications

This example demonstrates the VB API syntax in a macro written in Visual Basic for Applications, for example, as would be run by a button in a Microsoft Word document or Microsoft Excel spreadsheet. This example is identical to the previous example, except for the syntax.

```

Private Sub btnRun_Click()
    Dim asyncConnection As IpfcAsyncConnection
    Dim cAC As CCpfcAsyncConnection
    Dim session As IpfcBaseSession
    Dim descModel As IpfcModelDescriptor
    Dim descModelCreate As CCpfcModelDescriptor
    Dim model As IpfcModel
    Dim workDir As String
    Dim position As Integer
    On Error GoTo RunError
'=====
'First Argument : The path to the Pro/E executable along with command
'line options. -i and -g flags make Pro/ENGINEER run in non-graphic,
'non-interactive mode
'Second Argument: String path to menu and message files.
'=====
    Set cAC = New CCpfcAsyncConnection
    Set asyncConnection = cAC.Start(txtExePath.Text + " -g:no_graphics
    -i:rpc_input", ".")
    Set session = asyncConnection.session
'=====
'Get current directory
'Set it as working directory
'=====
    workDir = ActiveWorkbook.FullName
    position = InStrRev(workDir, "\")

```

```

workDir = Left(workDir, position)

session.ChangeDirectory (workDir)
'=====
'VB api process calls and other processing to be done
'=====
Set descModelCreate = New CCpfcModelDescriptor
Set descModel = descModelCreate.Create(EpfcModelType.EpfcMDL_PART,
    "partModel.prt", dbnull)
Set model = session.RetrieveModel(descModel)
'=====
'End the Pro/E session when done
'=====
If Not asyncConnection Is Nothing Then
    If asyncConnection.IsRunning Then
        asyncConnection.End
    End If
End If

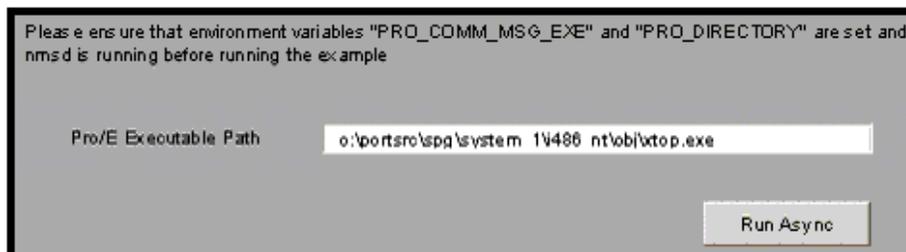
RunError:
If Err.Number <> 0 Then
    MsgBox "Process Failed : Unknown error occurred." + Chr(13) + _
        "Error No: " + CStr(Err.Number) + Chr(13) + _
        "Error: " + Err.Description, vbCritical, "Error"

    If Not asyncConnection Is Nothing Then
        If asyncConnection.IsRunning Then
            asyncConnection.End
        End If
    End If
End If

End Sub

```

The following figure displays the button in Microsoft Excel designed for the above application.



## Connecting to a Pro/ENGINEER Process

Methods Introduced:

- **CCpfcAsyncConnection.Connect()**
- **CCpfcAsyncConnection.ConnectWS()**
- **CCpfcAsyncConnection.GetActiveConnection()**
- **IpfcAsyncConnection.Disconnect()**

A simple asynchronous application can also connect to a Pro/ENGINEER process that is already running on a local computer. The method **CCpfcAsyncConnection.Connect()** performs this connection. This method fails to connect if multiple Pro/ENGINEER sessions are running. If several versions of Pro/ENGINEER are running on the same computer, try to connect by specifying user and display parameters. However, if several versions of Pro/ENGINEER are running in the same user and display parameters, the connection may not be possible.

**CCpfcAsyncConnection.ConnectWS()** connects to both Pro/ENGINEER and Pro/INTRALINK 3.x workspaces simultaneously.

**pfcAsyncConnection.IpfcAsyncConnection\_GetActiveConnection** returns the current connection to a Pro/ENGINEER session.

To disconnect from a Pro/ENGINEER process, call the method **IpfcAsyncConnection.Disconnect()**.

## Connecting Via Connection ID

Methods Introduced:

- **IpfcAsyncConnection.GetConnectionId()**
- **IpfcConnectionId.ExternalRep**
- **CCpfcConnectionId.Create()**
- **CCpfcAsyncConnection.ConnectById()**

Each Pro/ENGINEER process maintains a unique identity for communications purposes. Use this ID to reconnect to a Pro/ENGINEER process.

The method **IpfcAsyncConnection.GetConnectionId()** returns a data structure containing the connection ID.

If the connection id must be passed to some other application the method **IpfcConnectionId.ExternalRep** provides the string external representation for the connection ID.

The method **CCpfcConnectionId.Create()** takes a string representation and creates a **ConnectionId** data object. The method **CCpfcAsyncConnection.ConnectById()** connects to Pro/ENGINEER at the specified connection ID.

### Note:

Connection IDs are unique for each Pro/ENGINEER process and are not maintained after you quit Pro/ENGINEER.

## Status of a Pro/ENGINEER Process

Method Introduced:

- **IpfcAsyncConnection.IsRunning()**

To find out whether a Pro/ENGINEER process is running, use the method **pfcAsyncConnectionAsyncConnection.IsRunning**.

## Getting the Session Object

Method Introduced:

- **IpfcAsyncConnection.Session**

The method **IpfcAsyncConnection.Session** returns the session object representing the Pro/ENGINEER session. Use this object to access the contents of the Pro/ENGINEER session. See the [Session Objects](#) section for additional information.

# Full Asynchronous Mode

Full asynchronous mode is identical to the simple asynchronous mode except in the way the VB API application handles requests from Pro/ENGINEER. In simple asynchronous mode, it is not possible to process these requests. In full asynchronous mode, the application implements a control loop that ``listens" for messages from Pro/ENGINEER. As a result, Pro/ENGINEER can call functions in the application, including callback functions for menu buttons and notifications.

## Note:

Using full asynchronous mode requires starting or connecting to Pro/ENGINEER using the methods described in the previous sections. The difference is that the application must provide an event loop to process calls from menu buttons and listeners.

Methods Introduced:

- **IpfcAsyncConnection.EventProcess()**
- **IpfcAsyncConnection.WaitForEvents()**
- **IpfcAsyncConnection.InterruptEventProcessing()**
- **IpfcAsyncActionListener.OnTerminate()**

The control loop of an application running in full asynchronous mode must contain a call to the method **IpfcAsyncConnection.EventProcess()**, which takes no arguments. This method allows the application to respond to messages sent from Pro/ENGINEER. For example, if the user selects a menu button that is added by your application, **IpfcAsyncConnection.AsyncConnection.EventProcess** processes the call to your listener and returns when the call completes. For more information on listeners and adding menu buttons, see the [Session Objects](#) chapter.

The method **IpfcAsyncConnection.WaitForEvents()** provides an alternative to the development of an event processing loop in a full asynchronous mode application. Call this function to have the application wait in a loop for events to be passed from Pro/ENGINEER. No other processing takes place while the application is waiting. The loop continues until **IpfcAsyncConnection.InterruptEventProcessing()** is called from a VB callback action, or until the application detects the termination of Pro/ENGINEER.

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.

An `AsyncConnection` object can be assigned an Action Listener to bind a termination action that is executed upon the termination of Pro/ENGINEER. The method **IpfcAsyncActionListener.OnTerminate()** handles the termination that you must override. It sends a member of the class `IpfcTerminationStatus`, which is one of the following:

- `EpfcTERM_EXIT`--Normal exit (the user clicks Exit on the menu).
- `EpfcTERM_ABNORMAL`--Quit with error status.
- `EpfcTERM_SIGNAL`--Fatal signal raised.

Your application can interpret the termination type and take appropriate action. For more information on Action Listeners, see the [Action Listeners](#) section.

## Example Code

The following asynchronous class is a fully asynchronous application. It follows the procedure for a full asynchronous application:

1. The application establishes listeners for Pro/ENGINEER events, in this case, the menu button and the termination listener.

- The application goes into a control loop calling **EventProcess** which allows the application to respond to the Pro/ENGINEER events.

```
Public Class pfcFullAsyncExample
```

```
    Private asyncConnection As pfcls.IpfcAsyncConnection
```

```
    Public Sub New(ByVal exePath As String, ByVal workDir as String)
```

```
        Try
```

```
            startProE(exePath, workDir)
            addTerminationListener()
            addMenuAndButton()
            asyncConnection.WaitForEvents()
```

```
        Catch ex As Exception
```

```
            MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
```

```
        Finally
```

```
            If Not asyncConnection Is Nothing AndAlso asyncConnection.IsRunning Then
                asyncConnection.End()
            End If
```

```
    End Try
```

```
End Sub
```

```
=====
```

```
'Function    :    startProE
```

```
'Purpose     :    Start new Pro/ENGINEER session and change to current
'                directory.
```

```
=====
```

```
    Private Sub startProE(ByVal exePath As String, ByVal workDir As
                        String)
```

```
        asyncConnection = (New CCpfcAsyncConnection).Start(exePath, ".")
        asyncConnection.Session.ChangeDirectory
                                (System.Environment.CurrentDirectory)
```

```
    End Sub
```

```
=====
```

```
'Function    :    addTerminationListener
```

```
'Purpose     :    This function adds termination listener to the
'                Pro/ENGINEER session.
```

```
=====
```

```
    Private Sub addTerminationListener()
```

```
        Dim terminationListener As New ProEExitListener()
```

```
        Try
```

```
            asyncConnection.AddActionListener(terminationListener)
```

```
        Catch ex As Exception
```

```
            MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
```

```
        End Try
```

```
    End Sub
```

```
=====
```

```
'Class       :    ProEExitListener
```

```
'Purpose     :    This class must implement the listner interface along
'                with the correct client interface name. The OnTerminate
'                function is called when the Pro/ENGINEER session is ended
'                by the user.
```

```
=====
```

```
    Private Class ProEExitListener
```

```
        Implements IpfcAsyncActionListener
```

```
        Implements ICIPClientObject
```

```
        Implements IpfcActionListener
```

```

Public Function GetClientInterfaceName() As String Implements
    pfcls.ICIPClientObject.GetClientInterfaceName
    GetClientInterfaceName = "IpfcAsyncActionListener"
End Function

Public Sub OnTerminate(ByVal _Status As Integer) Implements
    pfcls.IpfcAsyncActionListener.OnTerminate
    Dim aC As pfcls.IpfcAsyncConnection

    aC = (New CCpfCAsyncConnection).GetActiveConnection
    aC.InterruptEventProcessing()

    MsgBox("ProE Exited")
End Sub

End Class

'=====  

'Function      :   addMenuAndButton  

'Purpose      :   This function demonstrates the usage of UI functions to  

'                :   add a new menu and button to Pro/ENGINEER.  

'=====  

Private Sub addMenuAndButton()  

    Dim session As pfcls.IpfcSession  

    Dim inputCommand As IpfcUICommand  

    Dim buttonListener As IpfcUICommandActionListener  

    Dim exitCommand As IpfcUICommand  

    Dim eListener As IpfcUICommandActionListener

    Try
        session = asyncConnection.Session  

        buttonListener = New ButtonListener()  

        eListener = New ExitListener()
    '=====  

    'Command is created which will be associated with the button. The class  

    'implementing the actionlistener must be given as input.  

    '=====  

        inputCommand = session.UICreateCommand("INPUT",  

            buttonListener)  

        exitCommand = session.UICreateCommand("EXIT", eListener)
    '=====  

    'Menu is created and buttons are created in the menu  

    '=====  

        session.UIAddMenu("VB-Async", "Windows",  

            "pfcAsynchronousModeExamples.txt", Nothing)  

        session.UIAddButton(exitCommand, "VB-Async", Nothing, _  

            "USER Exit Listener", "USER Exit Help",  

            "pfcAsynchronousModeExamples.txt")  

        session.UIAddButton(inputCommand, "VB-Async", Nothing, _  

            "USER Async App", "USER Async Help",  

            "pfcAsynchronousModeExamples.txt")

    Catch ex As Exception
        MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
    End Try

End Sub

'=====  

'Class        :   ButtonListener

```

```

'Purpose      :   This class must implement the listner interface along
'               with the correct client interface name. The OnCommand
'               function is called when the user button is pressed.
'=====
Private Class ButtonListener
    Implements pfcls.IpfcUICommandActionListener
    Implements ICIPClientObject

    Public Function GetClientInterfaceName() As String _
        Implements ICIPClientObject.GetClientInterfaceName
        GetClientInterfaceName = "IpfcUICommandActionListener"
    End Function

    Public Sub OnCommand() Implements
        pfcls.IpfcUICommandActionListener.OnCommand
        Me.UserFunction()
    End Sub

    Public Sub UserFunction()
        MsgBox("User Button Pressed")
    End Sub

End Class

'=====
'Class        :   ExitListener
'Purpose      :   This class must implement the listner interface along
'               with the correct client interface name. The OnCommand
'               function is called when the user button is pressed to
'               exit the session listener.
'=====
Private Class ExitListener
    Implements pfcls.IpfcUICommandActionListener
    Implements ICIPClientObject

    Public Function GetClientInterfaceName() As String _
        Implements ICIPClientObject.GetClientInterfaceName
        GetClientInterfaceName = "IpfcUICommandActionListener"
    End Function

    Public Sub OnCommand() Implements
        pfcls.IpfcUICommandActionListener.OnCommand
        Me.UserFunction()
    End Sub

    Public Sub UserFunction()

        Dim aC As pfcls.IpfcAsyncConnection

        aC = (New CCpfcsAsyncConnection).GetActiveConnection
        aC.InterruptEventProcessing()

        MsgBox("Listener Exited")
    End Sub

End Class

End Class

```

```
#
#
VB-Async
VB-Async
#
#
USER#Async#App
Async Button
#
#
USER#Async#Help
Button added via Async Application
#
#
```

## Troubleshooting VB API Applications

### General Problems

**pfExceptions.XToolkitNotFound** exception on the first call to **CCpfAsyncConnection.Start()** on Windows.

Make sure your command is correct. If it is not a full path to a script or executable, make sure \$PATH is set correctly. Try full path in the command: if it works, then your \$PATH is incorrect.

**pfExceptions.XToolkitGeneralError** or **pfExceptions.XToolkitCommError** on the first call to **CCpfAsyncConnection.Start()** or **CCpfAsyncConnection.Connect()**

- o Make sure the environment variable **PRO\_COMM\_MSG\_EXE** is set to the full path to **pro\_comm\_msg**, including <filename.exe>.
- o Make sure the environment variable **PRO\_DIRECTORY** is set to the Pro/ENGINEER installation directory.
- o Make sure name service (nmsd) is running.

**CCpfAsyncConnection.Start()** hangs, even though Pro/ENGINEER already started

Make sure name service (nmsd) is also started along with Pro/ENGINEER. Open Task Manager and look for **nmsd.exe** in the process listing.

---

# The VB API Online Browser

---

This section describes how to use the online browser provided with the VB APIWizard.

## Topic

[Online Documentation -- VB APIWizard](#)

## Online Documentation -- VB APIWizard

The VB API provides an online browser called the VB APIWizard that displays detailed documentation. This browser displays information from the *VB API User's Guide* and API specifications derived from the VB API header file data.

The VB APIWizard contains the following items:

- Definitions of the VB API modules
- Definitions of the VB API classes and interfaces and their hierarchical relationships
- Descriptions of the VB API methods
- Declarations of data types used by the VB API methods
- The VB API User's Guide that you can browse by topic or by class
- Code examples for the VB API methods (taken from sample applications provided as part of the the VB API installation)

Read the Release Notes and README file for the most up-to-date information on documentation changes.

### Note:

The VB API User's Guide is also available in PDF format at the following location:

```
<Pro/ENGINEER loadpoint>/vbapi/vbug.pdf
```

## Installing the APIWizard

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

```
<Pro/ENGINEER loadpoint>/vbapi/vbdoc
```

## Starting the APIWizard

Start the VB APIWizard by pointing your browser to:

```
<Pro/ENGINEER loadpoint>/vbapi/vbdoc/index.html
```

Your web browser will display the VB APIWizard data in a new window.

## Web Browser Environments

The APIWizard supports Netscape Navigator version 4 and later, and Internet Explorer version 5 and later.

For APIWizard use Internet Explorer, the recommended browser environment requires installation of the Java2 plug-in.

For Netscape Navigator, the recommended browser environment requires installation of the Java Swing foundation class. If this class is not loaded on your computer, the APIWizard can load it for you. This takes several minutes, and is not persistent.

between sessions. See [Loading the Swing Class Library](#) for the procedure on loading Swing permanently.

SGI hardware platform users must install the Swing class. For more information, refer to the section on [SGI Hardware Platforms](#).

## Loading the Swing Class Library

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

If you access the APIWizard with Netscape Navigator, follow these instructions to download and install the Java Foundation Class (Swing) archive:

[Download the Java Foundation Class \(Swing\) Archive](#)

[Modifying the Java Class Path on UNIX Platforms](#)

[Modifying the Java Class Path on NT Platforms](#)

### Download the Java Foundation Class (Swing) Archive

1. Go to the Java Foundation Class Download Page.
2. Go to the heading Downloading the JFC/Swing X.X.X Release, where X.X.X is the latest JFC version.
3. Click on the standard TAR or ZIP file link to go to the heading Download the Standard Version.
4. Do not download the "installer" version.
5. Select a file format, click Continue, and follow the download instructions on the subsequent pages.
6. Uncompress the downloaded bundle.

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. Add this archive to the Java Class Path as shown in the next sections.

### Modifying the Java Class Path on UNIX Platforms

Follow these steps to make the Java Foundation Class (Swing) available in UNIX shell environments:

1. If the CLASSPATH environment variable exists, then add the following line to the end of file ~/.cshrc

```
setenv CLASSPATH "${CLASSPATH}:[path_to_swingall.jar]"
```

Otherwise, add the following line to ~/.cshrc

```
setenv CLASSPATH ":[path_to_swingall.jar]"
```

2. Save and close ~/.cshrc.
3. Enter the following command:

```
source ~/.cshrc
```

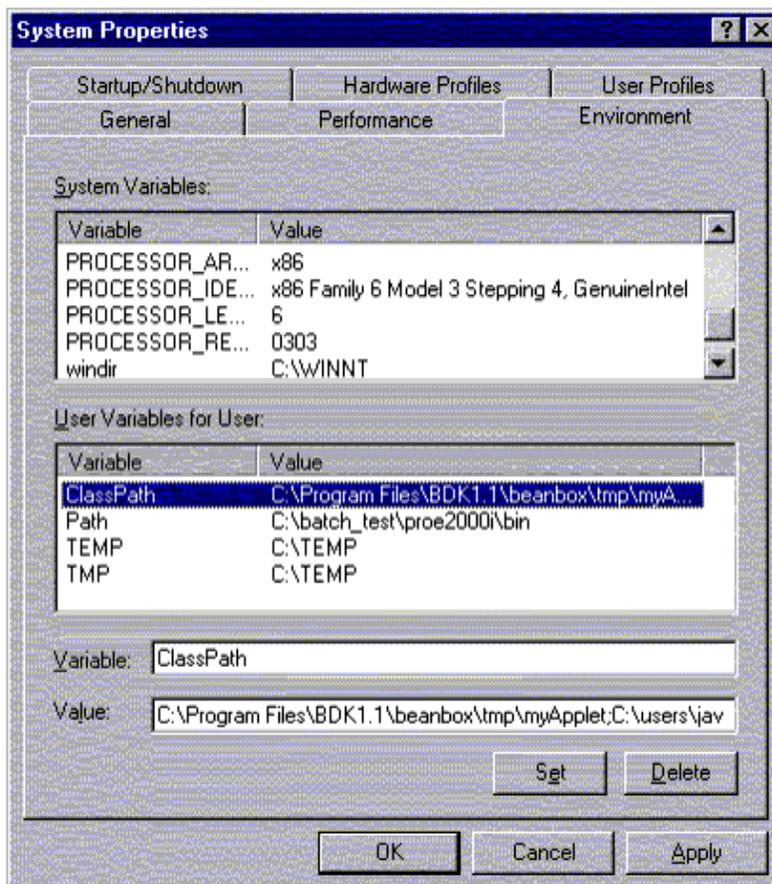
This sets the CLASSPATH environment variable in the current shell. All new shells will be also be affected.

4. Close and restart your internet browser from shell that uses the new class path data.

## Modifying the Java Class Path on NT Platforms

Follow these steps to make the Java Foundation Class (Swing) available on Windows NT Platforms:

1. Click on Start -- Settings -- Control Panel.
2. In the System Properties window, select the Environment tab.
3. Check in the User Variables display area for the ClassPath variable as shown in the following figure.



If the ClassPath variable exists, then follow these steps:

1. Click on ClassPath in the Variable column. The value of ClassPath will appear in the Value text field.
2. Append the path to the swingall.jar archive to the current value of ClassPath:

...:[path\_to\_swingall\_archive];.

Use the semicolon as the path delimiter before and after the path to the archive, and the period (.) at the end of the variable definition. There must be only one semicolon-period ";" entry in the ClassPath variable, and it should appear at the end of the class path.

If the ClassPath variable does not exist, then follow these steps:

1. In the Variable text field, enter ClassPath.
2. In the Value text field, enter:

[path\_to\_swingall\_archive];.

There must be a semicolon-period ";" entry at the end of the ClassPath variable.

3. Click the **Set** button.
4. Click the **Apply** button.
5. Click the **OK** button.
6. Close and restart your internet browser. You do not need to reboot your machine.

## SGI Hardware Platforms

Netscape returns a **Class Not Found** exception when downloading the Swing archive. The class appears to be in the archive, but Netscape improperly processes the archive.

For this reason, SGI hardware platform users must download the Swing archive and install it in their CLASSPATH as described in [Loading the Swing Class Library](#).

SGI platform user's must download and install the Java Foundation Class (Swing) archive. 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. SGI users should download and install the Java Foundation Class (Swing) archive before accessing the APIWizard.

## 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 is automatic tree scrolling.

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 method to work. See [Loading the Swing Class Library](#) for the procedure on loading Swing.

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

## APIWizard Interface

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

### Modules/Classes/Interfaces/Topic Selection Frame

This frame, located on the left of the screen, controls what is presented in the Display frame. Specify what data you want to view by choosing either the VB API **Modules**, **Classes**, **Interfaces**, **Exceptions**, **Enumerated Types**, or **The VB API User's Guide**.

In **Modules** mode, this frame displays an alphabetical list of the VB API modules. A module is a logical subdivision of functionality within the VB API; for example, the `pfcfamily` module contains classes, enumerated types, and collections related to family table operations. The frame can also display VB API classes, interfaces, enumerated types, and methods as subnodes of the modules.

In **Classes** mode, this frame displays an alphabetical list of the VB API classes. It can also display the VB API methods as subnodes of the classes.

In **Interfaces** mode, this frame displays an alphabetical list of the VB API interfaces.

In **Exceptions** mode, this frame displays an alphabetical list of named exceptions in the VB API library.

In **Enumerated Types** mode, this frame displays an alphabetical list of the VB API enumerated type classes.

In **The VB API User's Guide** mode, this frame displays the *VB API User's Guide* table of contents in a tree structure. All chapters are displayed as subnodes of the main *The VB API User's Guide* node.

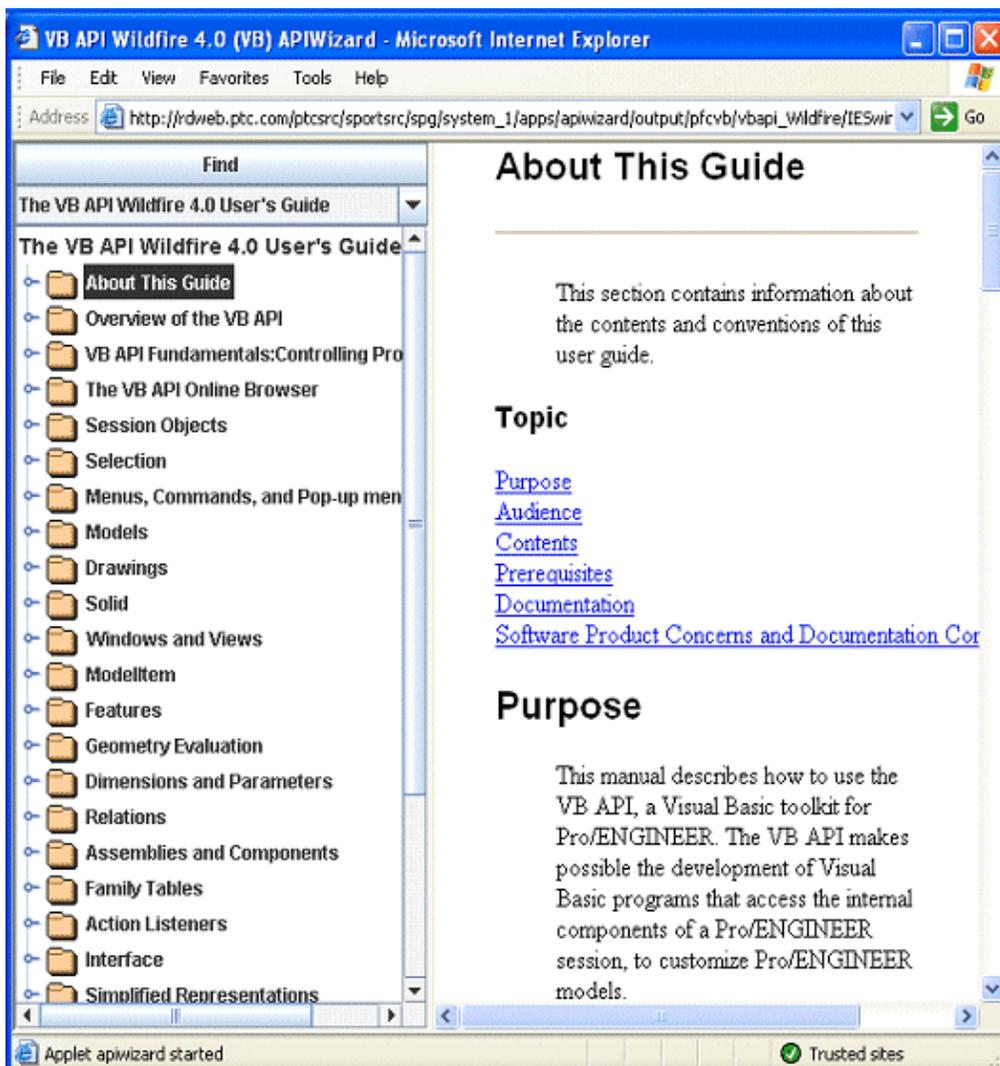
The Modules/Classes/Interfaces/Topic Selection frame includes a **Find** button for data searches of the *VB API User's Guide* or of API specifications taken from header files. See the section [APIWizard Search Feature \(Find\)](#) for more information on the Find feature.

## Display Frame

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

- The VB API module definitions
- The VB API class or interface definitions and their hierarchical relationships
- The VB API method descriptions
- User's Guide content
- Code examples for the VB API methods

The following figure displays the APIWizard interface layout.





## Navigating the Modules/Classes/Interfaces/Topic Selection Tree

Access all VB APIWizard online documentation for modules, classes, interfaces, enumerated types, methods, or the *VB API User's Guide* from the Modules/Classes/Interfaces/Topic Selection frame. This frame displays a tree structure of the data. Expand and collapse the tree as described below to navigate this data.

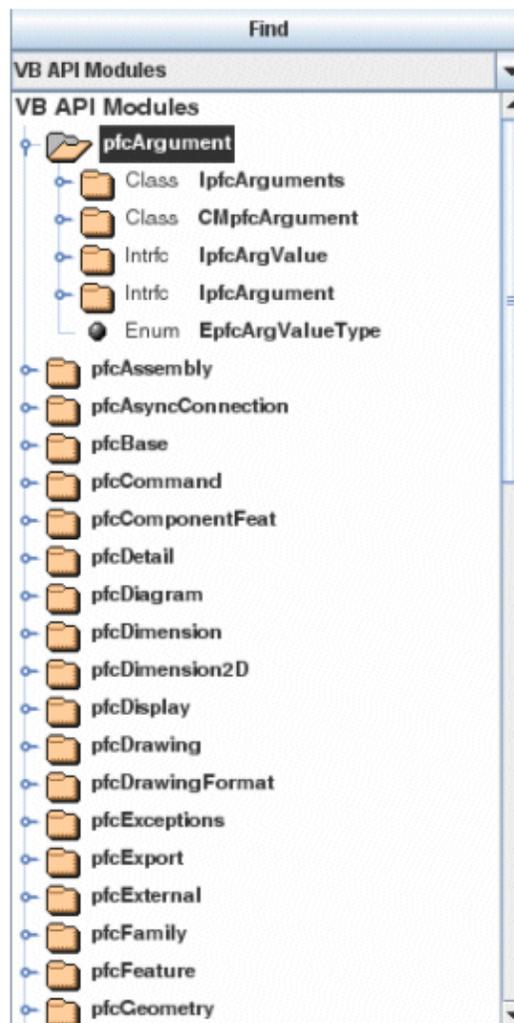
To expand the tree structure, first select the Modules, Classes, Interfaces, Exceptions, Enumerated Types, or the *VB API User's Guide* at the top of the Modules/Classes/Interfaces/Topic Selection frame. The APIWizard displays the tree structure in a collapsed form. The switch icon to the far left of a node (i.e. a module, a class, an interface, or chapter name) signifies that this node contains subnodes. If a node has no switch icon, it has no subnodes. Clicking the switch icon (or double-clicking 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 the VB API Modules

View the VB API modules by choosing **Modules** at the top of the Modules/Classes/Interfaces/Topic Selection frame. In this mode, all the VB API Modules and Classes are displayed in alphabetical order. The following tree displays the layout of the VB API modules in the alphabetical order.

The Display frame for each VB API module displays the information about the classes, enumerated types, and collections that belong to the module. Click the switch icon next to the desired module name, or double-click the module name text to view the classes, interfaces, or enumerated types. You can also view the methods for each class or interface in the expanded tree by clicking the switch icon next to the class or interface name, or by double-clicking the name.

The following figure shows the collapsed tree layout for the VB API modules.



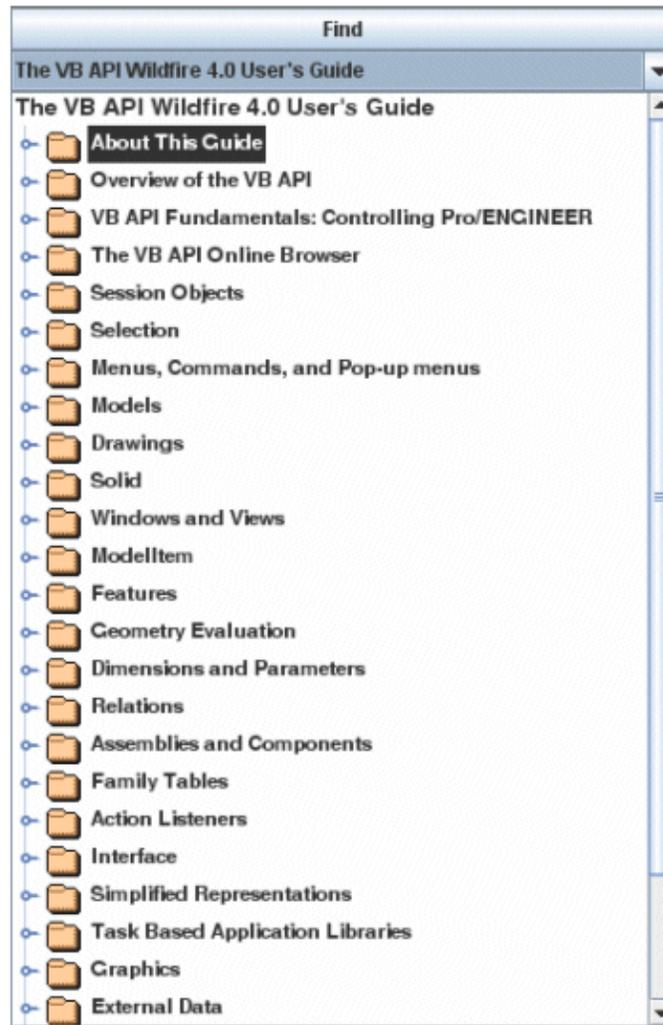


## Browsing the VB API User's Guide

View the *VB API User's Guide* by choosing **VB API User's Guide** at the top of the Modules/Classes/Interfaces/Topic Selection frame. In this mode, the APIWizard displays the User's Guide section headings.

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

The following figure shows the collapsed tree layout for the table of contents of the *VB API User's Guide*.



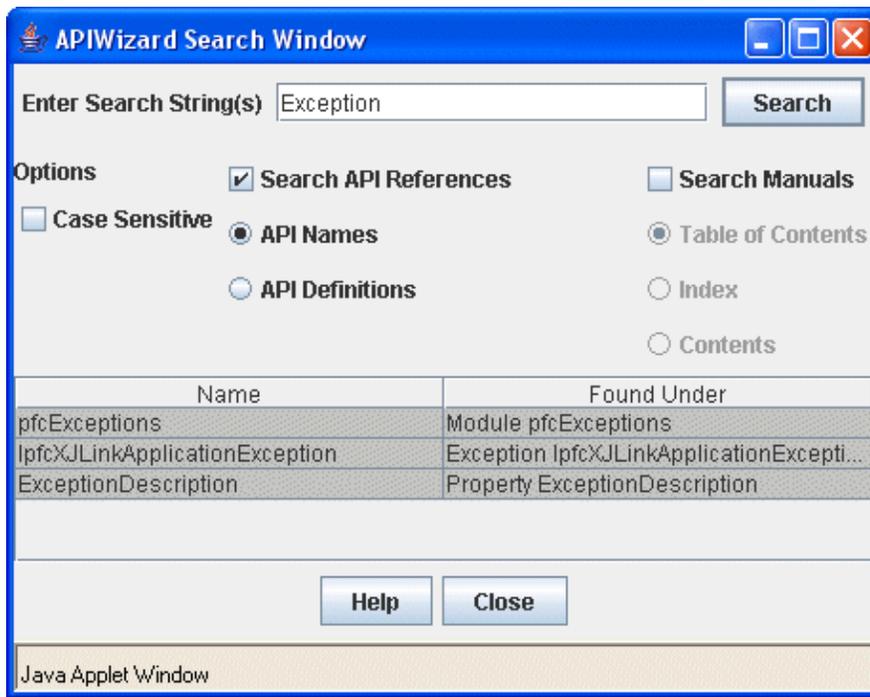
## APIWizard Search Feature (Find)

The APIWizard supports searches for specified strings against both the *VB API User's Guide* and API definition files. Click the **Find** button on the Modules/Classes/Interfaces/Topic Selection frame to display the APIWizard Search dialog.

### Note:

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

The following figure shows the APIWizard search dialog box with the results for the `Exception` search string.



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

- o Enter Search String(s)

Enter the specific search string or strings in this field. By default, the browser performs a non-case-sensitive search.

- o Search/Stop

Select the Search button to begin a search. During a search, this button name changes to Stop. Select the Stop button to stop a search.

- o Search API References

Select this button to search for data on API methods. Select the **API Names** button to search for method names only. Select the **Definitions** button to search the API method names and definitions for specific strings.

- o Search Manuals

Select this button to search the *VB API User's Guide* data. Select the **Table of Contents** button to search on TOC entries only. Select the **Index** button to search only the Index. Select the **Contents** button to search on all text in the *VB API User's Guide*.

- o Case Sensitive

Select this button to specify a case-sensitive search.

- o Name

This frame displays a list of strings found by the APIWizard search.

- o Found Under

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

- o Help

Select this button for help about the APIWizard search feature. The APIWizard presents this help data in the Display frame.

## Supported Search Types

The APIWizard Search supports the following:

- o Case sensitive searches
- o Search of API names and definitions, VB API User's Guide data, or both
- o Search of API data by API names only or by API names and definitions

- o Search of VB API User's Guide by Table of Contents only, by Index, or on the User's Guide contents (the entire text).
- o Wildcard searches--valid characters are:
  - \* (asterisk) matches zero or more non-whitespace characters
  - ? (question mark) matches one and only one non-whitespace character

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

Get\*Name

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

Get?Name

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

Get?\*Name

To search on the string Feature, followed by an \*

Feature\\*

To search on the string Feature, followed by a ?

Feature\?

To search on the string Feature, followed by a \

Feature\\

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

"family table"  
'Model\* methods'

- o 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 GetName OR GetId, enter:

Get\*Name Get\*Id

**Note:**

This search specification also returns strings that match both specified search targets.

For example:

```
FullName
```

returns **Model.GetName** and **ModelDescriptor.GetFullName**

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

```
Full* *Name
```

returns only one entry for each **FullName** property found.

Mix quoted and non-quoted strings as follows:

```
Get*Name "family table"
```

returns all instances of strings containing Get and Name, or strings containing family table.

## Performing an APIWizard Search

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

- o Select the Find icon at the top of the Modules/Classes/Interfaces/Topic Selection frame.
  - o Specify the string or strings to be searched for in the Enter Search String field.
  - o Select Case Sensitive to specify a case-sensitive search. Note that the default search is non-case-sensitive.
  - o Select either or both of the Search API References and Search User's Guide buttons. Select the options under these buttons as desired.
  - o Select the Search button. The APIWizard turns this button red and is renamed it Stop for the duration of the search.
  - o If the APIWizard finds the search string in the specified search area(s), 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.
  - o 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 Modules/Classes/Interfaces/Topic Selection frame tree, and then presents in the Display frame the online help data for the selected string.
-

# Session Objects

---

This section describes how to program on the session level using the VB API.

## Topic

[Overview of Session Objects](#)

[Directories](#)

[Accessing the Pro/ENGINEER Interface](#)

## Overview of Session Objects

The Pro/ENGINEER `Session` object (contained in the class `IpfcSession`) is the highest level object in the VB API. Any program that accesses data from Pro/ENGINEER must first get a handle to the `Session` object before accessing more specific data.

The `Session` object contains methods to perform the following operations:

- Accessing models and windows (described in the Models and Windows chapters).
- Working with the Pro/ENGINEER user interface.
- Allowing interactive selection of items within the session.
- Accessing global settings such as line styles, colors, and configuration options.

The following sections describe these operations in detail. Refer to the chapter Controlling Pro/ENGINEER for more information on how to connect to a Pro/ENGINEER session.

## Directories

Methods Introduced:

- **`IpfcBaseSession.GetCurrentDirectory()`**
- **`IpfcBaseSession.ChangeDirectory()`**

The method **`IpfcBaseSession.GetCurrentDirectory()`** returns the absolute path name for the current working directory of Pro/ENGINEER.

The method **`IpfcBaseSession.ChangeDirectory()`** changes Pro/ENGINEER to another working directory.

## Configuration Options

Methods Introduced:

- **`IpfcBaseSession.GetConfigOptionValues()`**
- **`IpfcBaseSession.SetConfigOption()`**

- **IpfcBaseSession.LoadConfigFile()**

You can access configuration options programmatically using the methods described in this section.

Use the method **IpfcBaseSession.GetConfigOptionValues()** to retrieve the value of a specified configuration file option. Pass the *Name* of the configuration file option as the input to this method. The method returns an array of values that the configuration file option is set to. It returns a single value if the configuration file option is not a multi-valued option. The method returns a null if the specified configuration file option does not exist.

The method **IpfcBaseSession.SetConfigOption()** is used to set the value of a specified configuration file option. If the option is a multi-value option, it adds a new value to the array of values that already exist.

The method **IpfcBaseSession.LoadConfigFile()** loads an entire configuration file into Pro/ENGINEER.

## Macros

Method Introduced:

- **IpfcBaseSession.RunMacro()**

The method **IpfcBaseSession.RunMacro()** runs a macro string. A VB API macro string is equivalent to a Pro/ENGINEER mapkey minus the key sequence and the mapkey name. To generate a macro string, create a mapkey in Pro/ENGINEER. Refer to the Pro/ENGINEER online help for more information about creating a mapkey.

Copy the Value of the generated mapkey Option from the **Tools>Options** dialog box. An example Value is as follows:

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

The key sequence is \$F2. The mapkey name is @MAPKEY\_LABELtest. The remainder of the string following the first semicolon is the macro string that should be passed to the method **IpfcBaseSession.RunMacro()**.

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 the mapkeys are not a supported scripting language. The syntax is not defined for users and is not guaranteed to remain constant across different datecodes of Pro/ENGINEER.

Macros are executed from synchronous mode only when control returns to Pro/ENGINEER from the VB API program. Macros are stored in reverse order (last in, first out).

Macros are executed as soon as they are registered. Macros are run in the same order that they are saved.

## Colors and Line Styles

Methods Introduced:

- **IpfcBaseSession.SetStdColorFromRGB()**

- **IpfcBaseSession.GetRGBFromStdColor()**
- **IpfcBaseSession.SetTextColor()**
- **IpfcBaseSession.SetLineStyle()**

These methods control the general display of a Pro/ENGINEER session.

Use the method **IpfcBaseSession.SetStdColorFromRGB()** to customize any of the Pro/ENGINEER standard colors.

To change the color of any text in the window, use the method **IpfcBaseSession.SetTextColor()**.

To change the appearance of nonsolid lines (for example, datums) use the method **IpfcBaseSession.SetLineStyle()**.

## Accessing the Pro/ENGINEER Interface

The `Session` object has methods that work with the Pro/ENGINEER interface. These methods provide access to the message window section

### The Text Message File

A text message file is where you define strings that are displayed in the Pro/ENGINEER user interface. This includes the strings on the command buttons that you add to the Pro/ENGINEER number, the help string that displays when the user's cursor is positioned over such a command button, and text strings that you display in the Message Window. You have the option of including a translation for each string in the text message file.

### 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 lower case 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 foreign 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

#### Note:

Message files are loaded into Pro/ENGINEER only once during a session. If you make a change to the message file while Pro/ENGINEER is running you must exit and restart Pro/ENGINEER before the change will 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 identifier for the message. This keyword must be unique for all Pro/ENGINEER messages.
2. The string that will be substituted for the identifier.

This string can include placeholders for run-time information stored in a `stringseq` object (shown in Writing Messages to the Message Window).

3. The translation of the message into another language (can be blank).
4. An intentionally blank line reserved for future extensions.

## Writing a Message Using a Message Pop-up Dialog Box

Method Introduced:

- **IpfcSession.UIShowMessageDialog()**

The method **IpfcSession.UIShowMessageDialog()** displays the UI message dialog. The input arguments to the method are:

- Message--The message text to be displayed in the dialog.
- Options--An instance of the `IpfcMessageDialogOptions` containing other options for the resulting displayed message. If this is not supplied, the dialog will show a default message dialog with an Info classification and an OK button. If this is not to be null, create an instance of this options type with `pfcUI.pfcUI.MessageDialogOptions_Create()`. You can set the following options:
  - Buttons--Specifies an array of buttons to include in the dialog. If not supplied, the dialog will include only the OK button. Use the method `IpfcMessageDialogOptions.Buttons` to set this option.
  - DefaultButton--Specifies the identifier of the default button for the dialog box. This must match one of the available buttons. Use the method `IpfcMessageDialogOptions.DefaultButton` to set this option.
  - DialogLabel--The text to display as the title of the dialog box. If not supplied, the label will be the english string "Info". Use the method `IpfcMessageDialogOptions.DialogLabel` to set this option.
  - MessageDialogType--The type of icon to be displayed with the dialog box (Info, Prompt, Warning, or Error). If not supplied, an Info icon is used. Use the method `IpfcMessageDialogOptions.MessageDialogType` to set this option.

## Accessing the Message Window

The following sections describe how to access the message window using the VB API. The topics are as follows:

- Writing Messages to the Message Window
- Writing Messages to an Internal Buffer

### Writing Messages to the Message Window

Methods Introduced:

- **IpfcSession.UIDisplayMessage()**
- **IpfcSession.UIDisplayLocalizedMessage()**
- **IpfcSession.UIClearMessage()**

These methods enable you to display program information on the screen.

The input arguments to the methods **IpfcSession.UIDisplayMessage()** and **IpfcSession.UIDisplayLocalizedMessage()** include the names of the message file, a message identifier, and (optionally) a `stringseq` object that contains upto 10 pieces of run-time information. For **IpfcSession.Session.UIDisplayMessage**, the strings in the `stringseq` are identified as `%0s, %1s, ... %9s` based on their location in the sequence. For **IpfcSession.Session.UIDisplayLocalizedMessage**, the strings in the `stringseq` are identified as `%0w, %1w, ... %9w` based on their location

in the sequence. To include other types of run-time data (such as integers or reals) you must first convert the data to strings and store it in the string sequence.

## Writing Messages to an Internal Buffer

Methods Introduced:

- **IpfcBaseSession.GetMessageContents()**
- **IpfcBaseSession.GetLocalizedMessageContents()**

The methods **IpfcBaseSession.GetMessageContents()** and **IpfcBaseSession.GetLocalizedMessageContents()** enable you to write a message to an internal buffer instead of the Pro/ENGINEER message area.

These methods take the same input arguments and perform exactly the same argument substitution and translation as the **IpfcSession.UIDisplayMessage()** and **IpfcSession.UIDisplayLocalizedMessage()** methods described in the previous section.

## Message Classification

Messages displayed in the VB API include a symbol that identifies the message type. Every message type is identified by a classification that begins with the characters %C. A message classification requires that the message key line (line one in the message file) must be preceded by the classification code.

### Note:

Any message key string used in the code should not contain the classification.

The VB API applications can now display any or all of the following message symbols:

- Prompt--This VB API message is preceded by a green arrow. The user must respond to this message type. Responding includes, specifying input information, accepting the default value offered, or canceling the application. If no action is taken, the progress of the application is halted. A response may either be textual or a selection. The classification for Prompt messages is %CP.
- Info--This VB API message is preceded by a blue dot. Info message types contain information such as user requests or feedback from the VB API or Pro/ENGINEER. The classification for Info messages is %CI.

### Note:

Do not classify messages that display information regarding problems with an operation or process as Info. These types of messages must be classified as Warnings.

- Warning--This VB API message is preceded by a triangle containing an exclamation point. Warning message types contain information to alert users to situations that could potentially lead to an error during a later stage of the process. Examples of warnings could be a process restriction or a suspected data problem. A Warning will not prevent or interrupt a process. Also, a Warning should not be used to indicate a failed operation. Warnings must only caution a user that the completed operation may not have been performed in a completely desirable way. The classification for Warning messages is %CW.
- Error--This VB API message is preceded by a broken square. An Error message informs the user that a required task was not completed successfully. Depending on the application, a failed task may or may not require intervention or correction before work can continue. Whenever possible redress this situation by providing a path. The classification for Error messages is %CE.
- Critical--This VB API message is preceded by a red X. A Critical message type informs the user of an extremely serious situation that is usually preceded by loss of user data. Options redressing this situation, if available, should be provided within the message. The classification for a Critical messages is %CC.

**Example Code: Writing a Message**

The following example code demonstrates how to write a message to the message window. The program uses the message file *mymessages.txt*, which contains the following lines:

---

```
USER Error: %0s of code %1s at %2s
Error: %0s of code %1s at %2s
#
#
```

---

```
Public Sub printError(ByVal session As pfcls.IpfcSession, ByVal location As
String, _ByVal err As String, ByVal errorCode As Integer)
    Dim message As Istringseq

    Try
        message = New Cstringseq
        message.Set(0, err)
        message.Set(1, errorCode.ToString)
        message.Set(2, location)

        session.UIDisplayMessage("pfcSessionObjectsExamples.txt",
            _"USER Error: %0s of code %1s at %2s",
            _message)

    Catch ex As Exception
        MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
        Exit Sub
    End Try
End Sub
```

## Reading Data from the Message Window

Methods Introduced:

- **IpfcSession.UIReadIntMessage()**
- **IpfcSession.UIReadRealMessage()**
- **IpfcSession.UIReadStringMessage()**

These methods enable a program to get data from the user.

The **IpfcSession.UIReadIntMessage()** and **IpfcSession.UIReadRealMessage()** methods contain optional arguments that can be used to limit the value of the data to a certain range.

The method **IpfcSession.UIReadStringMessage()** includes an optional Boolean argument that specifies whether to echo characters entered onto the screen. You would use this argument when prompting a user to enter a password.

## Displaying Feature Parameters

Method Introduced:

- **IpfcSession.UIDisplayFeatureParams()**

The method **IpfcSession.UIDisplayFeatureParams()** forces Pro/ENGINEER to show dimensions or other parameters stored on a specific feature. The displayed dimensions may then be interactively selected by the user.

## File Dialogs

Methods Introduced:

- **IpfcSession.UIOpenFile()**
- **CCpfcFileOpenOptions.Create()**
- **IpfcSession.UISaveFile()**
- **CCpfcFileSaveOptions.Create()**
- **IpfcSession.UISelectDirectory()**
- **CCpfcDirectorySelectionOptions.Create()**

The method **IpfcSession.UIOpenFile()** invokes the Pro/ENGINEER dialog box for opening files and browsing directories. The method lets you specify several options through the input argument *IpfcFileOpenOptions*.

Use the method **CCpfcFileOpenOptions.Create()** to create a new instance of the *IpfcFileOpenOptions* object. You can set the following options for this object:

- **FilterString**--Specifies the filter string for the type of file accepted by the dialog. Multiple file types should be listed with wildcards and separated by commands, for example, "\*.prt,\*.asm". Use the method *IpfcFileOpenOptions.FilterString* to set this option.
- **PreselectedItem**--Specifies the name of an item to preselect in the dialog. Use the method *IpfcFileOpenOptions.PreselectedItem* to set this option.
- **DefaultPath**--Specifies the name of the path to be opened by default in the dialog. Use the method *IpfcFileUIOptions.DefaultPath* to set this option.
- **DialogLabel**--Specifies the title of the dialog. Use the method *IpfcFileUIOptions.DialogLabel* to set this option.
- **Shortcuts**--Specifies the names of shortcut path to make available in the dialog. Use the method *IpfcFileUIOptions.Shortcuts* to set this option. Create these items using the method *pfcUI.FileOpenShortcut\_Create*.

The method returns the file selected by the user. The application must use other methods or techniques to perform the desired action on the file.

The method **IpfcSession.UISaveFile()** invokes the Pro/ENGINEER dialog box for saving a file. The method accepts similar options to **IpfcSession.UIOpenFile()** through the class *IpfcFileSaveOptions*. Create the options using **CCpfcFileSaveOptions.Create()**. When using the **Save** dialog the user will be permitted to set the name to a non-existent file. The method returns the name of the file selected by the user; the application must use other methods or techniques to perform the desired action on the file.

The method **IpfcSession.UISelectDirectory()** 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. This method accepts options for the dialog title, shortcuts, and default path created using **CCpfcDirectorySelectionOptions.Create()**. The method returns the selected directory path; the application must use other methods or techniques to do something with this selected path.

---

# Selection

---

This section describes how to use Interactive Selection in the VB API.

## Topic

[Interactive Selection](#)

[Accessing Selection Data](#)

[Programmatic Selection](#)

[Selection Buffer](#)

## Interactive Selection

Methods and Properties Introduced:

- **IpfcBaseSession.Select()**
- **CCpfcSelectionOptions.Create()**
- **IpfcSelectionOptions.MaxNumSels**
- **IpfcSelectionOptions.OptionKeywords**

The method **IpfcBaseSession.Select()** activates the standard Pro/ENGINEER menu structure for selecting objects and returns a `IpfcSelections` sequence that contains the objects the user selected. Using the *Options* argument, you can control the type of object that can be selected and the maximum number of selections.

In addition, you can pass in a `IpfcSelections` sequence to the method. The returned `IpfcSelections` sequence will contain the input sequence and any new objects.

The method **CCpfcSelectionOptions.Create()** and the property **IpfcSelectionOptions.OptionKeywords** take a `String` argument made up of one or more of the identifiers listed in the table below, separated by commas.

For example, to allow the selection of features and axes, the arguments would be "feature,axis".

Pro/ENGINEER Database Item	String Identifier	ModelItemType
Datum point	point	EpfcITEM_POINT

Datum axis	axis	EpfcITEM_AXIS
Datum plane	datum	EpfcITEM_FEATURE
Coordinate system datum	csys	EpfcITEM_COORD_SYS
Feature	feature	EpfcITEM_FEATURE
Edge (solid or datum surface)	edge	EpfcITEM_EDGE
Edge (solid only)	sldedge	EpfcITEM_EDGE
Edge (datum surface only)	qltedge	EpfcITEM_EDGE
Datum curve	curve	EpfcITEM_CURVE
Composite curve	comp_crv	EpfcITEM_CURVE
Surface (solid or quilt)	surface	EpfcITEM_SURFACE
Surface (solid)	sldface	EpfcITEM_SURFACE
Surface (datum surface)	qltface	EpfcITEM_SURFACE
Quilt	dtmqlt	EpfcITEM_QUILT
Dimension	dimension	EpfcITEM_DIMENSION
Reference dimension	ref_dim	EpfcITEM_REF_DIMENSION
Integer parameter	ipar	EpfcITEM_DIMENSION
Part	part	N/A
Part or subassembly	prt_or_asm	N/A

Assembly component model	component	N/A
Component or feature	membfeat	EpcfITEM_FEATURE
Detail symbol	dtl_symbol	EpcfITEM_DTL_SYM_INSTANCE
Note	any_note	EpcfITEM_NOTE, ITEM_DTL_NOTE
Draft entity	draft_ent	EpcfITEM_DTL_ENTITY
Table	dwg_table	EpcfITEM_TABLE
Table cell	table_cell	EpcfITEM_TABLE
Drawing view	dwg_view	N/A

When you specify the maximum number of selections, the argument to **IpfcSelectionOptions.MaxNumSels** must be an Integer. The default value assigned when creating a `IpfcSelectionOptions` object is -1, which allows any number of selections by the user.

## Accessing Selection Data

Properties Introduced:

- **IpfcSelection.SelModel**
- **IpfcSelection.SelItem**
- **IpfcSelection.Path**
- **IpfcSelection.Params**
- **IpfcSelection.TParam**
- **IpfcSelection.Point**
- **IpfcSelection.Depth**
- **IpfcSelection.SelView2D**
- **IpfcSelection.SelTableCell**

## ● **IpfcSelection.SelTableSegment**

These properties return objects and data that make up the selection object. Using the appropriate properties, you can access the following data:

- For a selected model or model item use `pfcSelection.SelModel` or `pfcSelection.SelItem`.
- For an assembly component use `pfcSelection.Path`.
- For UV parameters of the selection point on a surface use `pfcSelection.Params`.
- For the T parameter of the selection point on an edge or curve use `pfcSelection.TParam`.
- For a three-dimensional point object that contains the selected point use `pfcSelection.Point`.
- For selection depth, in screen coordinates use `pfcSelection.Depth`.
- For the selected drawing view, if the selection was from a drawing, use `pfcSelection.SelView2D`.
- For the selected table cell, if the selection was from a table, use `pfcSelection.SelTableCell`.
- For the selected table segment, if the selection was from a table, use `pfcSelection.GetSelTableSegment`.

## **Controlling Selection Display**

Methods Introduced:

- **IpfcSelection.Highlight()**
- **IpfcSelection.UnHighlight()**
- **IpfcSelection.Display()**

These methods cause a specific selection to be highlighted or dimmed on the screen using the color specified as an argument.

The method **IpfcSelection.Highlight()** highlights the selection in the current window. This highlight is the same as the one used by Pro/ENGINEER 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 **IpfcWindow.Repaint()**; it is not removed if you use **IpfcWindow.Refresh()**.

The method **IpfcSelection.UnHighlight()** removes the highlight.

The method **IpfcSelection.Display()** causes a selected object to be displayed on the screen, even if it is suppressed or hidden.

### **Note:**

This is a one-time action and the next repaint will erase this display.

### **Example Code: Using Interactive Selection**

The following example code demonstrates how to use the VB API to allow interactive selection.

```
Imports pfcls
```

```
Public Class pfcSelectionExamples
```

```

Public Function selectFeatures(ByVal session As IpfcBaseSession,
                              _ByVal max As Integer) As CpfcSelections
    Dim selections As CpfcSelections
    Dim selectionOptions As IpfcSelectionOptions

    Try
'=====
'Selection options are set to select only features with a specified max
'number.
'=====
        selectionOptions = (New CCpfcSelectionOptions).Create("feature")
        selectionOptions.MaxNumSels = max
        selections = session.Select(selectionOptions, Nothing)

        selectFeatures = selections

    Catch ex As Exception
        MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
        Return Nothing
    End Try
    End Function
End Class

```

## Programmatic Selection

The VB API provides methods whereby you can make your own Selection objects, without prompting the user. These Selections are required as inputs to some methods and can also be used to highlight certain objects on the screen.

Methods Introduced:

- **CMpfcSelect.CreateModelItemSelection()**
- **CMpfcSelect.CreateComponentSelection()**

The method **CMpfcSelect.CreateModelItemSelection()** creates a selection out of any model item object. It takes a `IpfcModelItem` and optionally a `IpfcComponentPath` object to identify which component in an assembly the Selection Object belongs to.

The method **CMpfcSelect.CreateComponentSelection()** creates a selection out of any component in an assembly. It takes a `IpfcComponentPath` object. For more information about `IpfcComponentPath` objects, see the section [Getting a Solid Object](#).

Some VB API methods require more information to be set in the selection object. The VB API methods allow you to set the following:

The selected item using the method **IpfcSelection.SelItem**.

The selected component path using the method **IpfcSelection.Path**.

The selected UV parameters using the method **IpfcSelection.Params**.

The selected T parameter (for a curve or edge), using the method **IpfcSelection.TParam**.

The selected XYZ point using the method **IpfcSelection.Point**.

The selected table cell using the method **IpfcSelection.SelTableCell**.

The selected drawing view using the method **IpfcSelection.SelView2D**.

## Selection Buffer

### Introduction to Selection Buffers

Selection is the process of choosing items on which you want to perform an operation. In Pro/ENGINEER, before a feature tool is invoked, the user can select items to be used in a given tool's collectors. Collectors are like storage bins of the references of selected items. The location where preselected items are stored is called the selection buffer.

Depending on the situation, different selection buffers may be active at any one time. In Part and Assembly mode, Pro/ENGINEER offers the default selection buffer, the Edit selection buffer, and other more specialized buffers. Other Pro/ENGINEER modes offer different selection buffers.

In the default Part and Assembly buffer there are two levels at which selection is done:

- First Level Selection

Provides access to higher-level objects such as features or components. You can make a second level selection only after you select the higher-level object.

- Second Level Selection

Provides access to geometric objects such as edges and faces.

**Note:**

First-level and second-level objects are usually incompatible in the selection buffer.

The VB API allows access to the contents of the currently active selection buffer. The available functions allow your application to:

- Get the contents of the active selection buffer.
- Remove the contents of the active selection buffer.
- Add to the contents of the active selection buffer.

### Reading the Contents of the Selection Buffer

Properties Introduced:

- **IpfcSession.CurrentSelectionBuffer**

- **IpfcSelectionBuffer.Contents**

The property **IpfcSession.CurrentSelectionBuffer** returns the selection buffer object for the current active model in session. The selection buffer contains the items preselected by the user to be used by the selection tool and popup menus.

Use the property **IpfcSelectionBuffer.Contents** to access the contents of the current selection buffer. The method returns independent copies of the selections in the selection buffer (if the buffer is cleared, this array is still valid).

## Removing the Items of the Selection Buffer

Methods Introduced:

- **IpfcSelectionBuffer.RemoveSelection()**
- **IpfcSelectionBuffer.Clear()**

Use the method **IpfcSelectionBuffer.RemoveSelection()** to remove a specific selection from the selection buffer. The input argument is the *IndexToRemove* specifies the index where the item was found in the call to the method **IpfcSelectionBuffer.Contents**.

Use the method **IpfcSelectionBuffer.Clear()** to clear the currently active selection buffer of all contents. After the buffer is cleared, all contents are lost.

## Adding Items to the Selection Buffer

Method Introduced:

- **IpfcSelectionBuffer.AddSelection()**

Use the method **IpfcSelectionBuffer.AddSelection()** to add an item to the currently active selection buffer.

**Note:**

The selected item must refer to an item that is in the current model such as its owner, component path or drawing view.

This method may fail due to any of the following reasons:

- There is no current selection buffer active.
  - The selection does not refer to the current model.
  - The item is not currently displayed and so cannot be added to the buffer.
  - The selection cannot be added to the buffer in combination with one or more objects that are already in the buffer. For example: geometry and features cannot be selected in the default buffer at the same time.
-

# Menus, Commands, and Pop-up Menus

---

This section describes the methods provided by the VB API to create and modify menus, menu buttons, commands, and pop-up menus in the Pro/ENGINEER user interface.

## Topic

[Introduction](#)

[Menu Bar Definitions](#)

[Creating New Menus and Buttons](#)

[Designating Commands](#)

[Pop-up Menus](#)

## Introduction

The VB API menu bar classes enable you to modify existing Pro/ENGINEER menu bar menus and to create new menu bar menus.

## Menu Bar Definitions

- Menu bar--The top level horizontal bar in the Pro/ENGINEER UI, containing the main menus, such as 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.
- Menu bar 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.
- Tool bar button--An item with a name or icon or both in a tool bar that is used to launch a set of instructions. An example is the New File command shown on the File toolbar.
- Pop-up menu--A menu invoked by selection of an item in the Pro/ENGINEER graphics window.
- Command--A procedure in Pro/ENGINEER that may be activated from a menu bar, tool bar, or pop-up menu button.

## Creating New Menus and Buttons

The following methods enable you to create new menu buttons in any location on the menu bar.

Methods Introduced:

- **IpfcSession.UICreateCommand()**
- **IpfcSession.UICreateMaxPriorityCommand()**
- **IpfcSession.UIAddButton()**
- **IpfcSession.UIAddMenu()**
- **IpfcUICommandActionListener.OnCommand()**

The method **IpfcSession.UICreateCommand()** creates a `IpfcUICommand` object that contains a `IpfcCommand`. `UICommandActionListener`. You should override the **IpfcUICommandActionListener.OnCommand()** method with the code that you want to execute when the user clicks a button.

The method **IpfcSession.UICreateMaxPriorityCommand()** creates a `pfccommand.UICommand` object having maximum command priority. The priority of the action refers to the level of precedence the added action takes over other Pro/ENGINEER actions. Maximum priority actions dismiss all other actions except asynchronous actions.

Maximum command priority should be used only in commands that open and activate a new model in a window. Create all other commands using the method **IpfcSession.UICreateCommand()**.

The method **IpfcSession.UIAddButton()** enables you to add your command to a menu on the menu bar. It also enables you to specify a help message that is displayed when the user moves the pointer over the button.

The **IpfcSession.UIAddMenu()** method enables you to create new, top-level menus that can contain your own commands or to add submenus to existing menus.

**Note:**

The menu file required when adding a menu or a button must have the same format as the text message file described above.

The listener method **pfccommand.UICommandListener.OnCommand** is called when the command is activated in Pro/ENGINEER by pressing a button.

### Example 1: Adding a Menu Button

The following example code demonstrates the usage of UI methods to add a new button to a Pro/ENGINEER Windows Menu. Note that this operates in a Full Asynchronous Mode

```
'=====
'Class      :   pfccommandObjectsExamples2
'Purpose    :   This class is used for adding button to ProE Windows
'            :   Menu. It uses timer object to handle event callback
'=====
Public Class pfccommandObjectsExamples2
    Implements IpfcAsyncActionListener
    Implements ICIPClientObject
    Implements IpfcActionListener

    Dim WithEvents eventTimer As Timers.Timer
    Dim exitFlag As Boolean = False
    Dim aC As pfcls.IpfcAsyncConnection

    Public Sub New(ByRef asyncConnection As pfcls.IpfcAsyncConnection)
        aC = asyncConnection
    End Sub

    Public Function GetClientInterfaceName() As String Implements pfcls.
ICIPClientObject.GetClientInterfaceName
        GetClientInterfaceName = "IpfcAsyncActionListener"
    End Function

    Public Sub OnTerminate(ByVal _Status As Integer) Implements pfcls.
IpfcAsyncActionListener.OnTerminate
        aC.InterruptEventProcessing()
        exitFlag = True
    End Sub

'Add menu button
```

```

' =====
'Function   :   addInputButton
'Purpose    :   This function demonstrates the usage of UI functions to
'              add a new button to ProE Windows Menu.
'              Note that this operates in Full Asynchronous Mode
' =====

Public Sub addInputButton()

    Dim inputCommand As IpfcUICommand
    Dim buttonListener As IpfcUICommandActionListener

    Try
' =====
' Start the timer to call EventProcess at regular intervals
' =====
        eventTimer = New Timers.Timer(500)
        eventTimer.Enabled = True
        AddHandler eventTimer.Elapsed, AddressOf Me.timeElapsed
' =====
' Command is created which will be associated with the button. The class
' implementing the actionlistener must be given as input.
' =====
        buttonListener = New GatherInputListener()
        inputCommand = aC.Session.UICreateCommand("INPUT",
                                                    buttonListener)
' =====
' Button is created in the menu "Windows"
' =====
        aC.Session.UIAddButton(inputCommand, "Windows", Nothing,
                                _"USER Async App", "USER Async Help",
                                "pfcSessionObjectsExamples.txt")
        aC.AddActionListener(Me)

    Catch ex As Exception
        MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
    End Try
End Sub

' =====
'Function   :   timeElapsed
'Purpose    :   This function handles the time elapsed event of timer
'              which is fired at regular intervals
' =====
Private Sub timeElapsed(ByVal sender As Object, ByVal e As System.Timers.
ElapsedEventArgs)
    If exitFlag = False Then
        aC.EventProcess()
    Else
        eventTimer.Enabled = False
    End If
End Sub

' =====
'Class      :   GatherInputListener
'Purpose    :   This class must implement the listner interface along
'              with the correct client interface name. The OnCommand
'              function is called when the user button is pressed.
' =====
Private Class GatherInputListener
    Implements pfcls.IpfcUICommandActionListener

```

```

Implements ICIPClientObject

Public Function GetClientInterfaceName() As String _
    Implements ICIPClientObject.GetClientInterfaceName
    GetClientInterfaceName = "IpfcUICommandActionListener"
End Function

Public Sub OnCommand() Implements pfcls.IpfcUICommandActionListener.OnCommand
    Me.UserFunction()
End Sub

Public Sub UserFunction()
    MsgBox("User Button Pressed")
End Sub
End Class

```

The corresponding message file for this example code contains two text messages. The first is used as a button name, the second as its help string.

```

#
#
USER#Async#App
Async Button
#
#
USER#Async#Help
Button added via Async Application
#
#

```

## Finding Pro/ENGINEER Commands

This method enables you to find existing Pro/ENGINEER commands in order to modify their behavior.

Method Introduced:

- **IpfcSession.UIGetCommand()**

The method **IpfcSession.UIGetCommand()** returns a `IpfcUICommand` object representing an existing Pro/ENGINEER command. The method allows you to find the command ID for an existing command so that 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.

Use the trail file to find the name of an action command (not a menu button). Click the corresponding icon on the toolbar (not the button in the menu bar) and 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 icon is `ProCmdModelSave`. This string can be used as input to **IpfcSession.UIGetCommand()** to get the command ID.

You can determine a command ID string for an option without an icon by searching through the resource files located in the `<Pro/ENGINEER Loadpoint>/text/resources` directory. If you search for the menu button name, the line

will contain the corresponding action command for the button.

## Access Listeners for Commands

These methods allow you to apply an access listener to a command. The access listener determines whether or not the command is visible at the current time in the current session.

Methods Introduced:

- **IpfcActionSource.AddActionListener()**
- **IpfcUICommandAccessListener.OnCommandAccess()**

Use the method **IpfcActionSource.AddActionListener()** to register a new **pfcCommand.UICCommandAccessListener** on any command (created either by an application or Pro/ENGINEER). This listener will be called when buttons based on the command might be shown.

The listener method **IpfcUICommandAccessListener.OnCommandAccess()** allows you to impose an access function on a particular command. The method determines if the action or command should be available, unavailable, or hidden.

The potential return values are listed in the enumerated type `EpfcCommandAccess` and are as follows:

- `EpfcACCESS_REMOVE`--The button is not visible and if all of the menu buttons in the containing menu possess an access function returning `ACCESS_REMOVE`, the containing menu will also be removed from the Pro/ENGINEER user interface..
- `EpfcACCESS_INVISIBLE`--The button is not visible.
- `EpfcACCESS_UNAVAILABLE`--The button is visible, but gray and cannot be selected.
- `EpfcACCESS_DISALLOW`--The button shows as available, but the command will not be executed when it is chosen.
- `EpfcACCESS_AVAILABLE`--The button is not gray and can be selected by the user. This is the default value.

### Example 2: Command Access Listeners

This example code demonstrates the usage of the access listener method for a particular command. The `OnCommandAccess` function returns "ACCESS\_UNAVAILABLE" that disables button associated with the command, if the model is not present or if it is not of type PART. Else, the function returns "ACCESS\_AVAILABLE" that enables button associated with the command.

```
'=====
'Class      :   CheckAccess
'Purpose    :   This listener class checks if command is accessible to
'            :   the user.
'=====
Private Class CheckAccess
    Implements ICIPClientObject
    Implements IpfcUICommandAccessListener
    Implements IpfcActionListener

    Dim aC As pfcls.IpfcAsyncConnection

    Public Sub New(ByRef asyncConnection As IpfcAsyncConnection)
        aC = asyncConnection
    End Sub

    Public Function GetClientInterfaceName() As String _
```

```

        Implements ICIPClientObject.GetClientInterfaceName
        GetClientInterfaceName = "IpfcUICommandAccessListener"
    End Function

    Public Function OnCommandAccess(ByVal _AllowErrorMessages As
        Boolean) As Integer Implements
        pfcls.IpfcUICommandAccessListener.OnCommandAccess
        Dim model As IpfcModel
        model = aC.Session.CurrentModel
        If model Is Nothing OrElse (Not model.Type =
            EpfcModelType.EpfcMDL_PART) Then
            Return EpfcCommandAccess.EpfcACCESS_UNAVAILABLE
        End If
        Return EpfcCommandAccess.EpfcACCESS_AVAILABLE

    End Function
End Class

```

## Bracket Listeners for Commands

These methods allow you to apply a bracket listener to a command. The bracket listener is called before and after the command runs, which allows your application to provide custom logic to execute whenever the command is selected.

Methods Introduced:

- **IpfcActionSource.AddActionListener()**
- **IpfcUICommandBracketListener.OnBeforeCommand()**
- **IpfcUICommandBracketListener.OnAfterCommand()**

Use the method **IpfcActionSource.AddActionListener()** to register a new **pfcCommand.UICCommandBracketListener** on any command (created either by an application or Pro/ENGINEER). This listener will be called when the command is selected by the user.

The listener methods **pfcCommand.UICCommandBracketListener.OnBeforeComm and** and **pfcCommand.UICCommandBracketListener.OnAfterCommand** allow the creation of functions that will be called immediately before and after execution of a given command. These methods are used to add the business logic to the start or end (or both) of an existing Pro/ENGINEER command.

The method **pfcCommand.UICCommandBracketListener.OnBeforeComm and** could also be used to cancel an upcoming command. To do this, throw a **pfcExceptions.XCancelProEAction** exception from the body of the listener method using **CCpfcXCancelProEAction.Throw()**.

### Example 3: Bracket Listeners

The following example code demonstrates the usage of the bracket listener methods that are called before and after when the user tries to rename the model. If the model contains a certain parameter, the rename attempt will be aborted by this listener.

```

Public Class pfcCommandExamples1
    Implements IpfcAsyncActionListener
    Implements ICIPClientObject
    Implements IpfcActionListener

```

```

Dim WithEvents eventTimer As Timers.Timer
Dim exitFlag As Boolean = False
Dim aC As pfcls.IpfcAsyncConnection

Public Sub New(ByRef asyncConnection As pfcls.IpfcAsyncConnection)
    aC = asyncConnection
End Sub

Public Function GetClientInterfaceName() As String Implements
    pfcls.ICIPClientObject.GetClientInterfaceName
    GetClientInterfaceName = "IpfcAsyncActionListener"
End Function

Public Sub OnTerminate(ByVal _Status As Integer) Implements
    pfcls.IpfcAsyncActionListener.OnTerminate
    aC.InterruptEventProcessing()
    exitFlag = True
End Sub

'=====  

'Function      :   timeElapsed  

'Purpose       :   This function handles the time elapsed event of timer  

'               :   which is fired at regular intervals  

'=====  

Private Sub timeElapsed(ByVal sender As Object, ByVal e As
    System.Timers.ElapsedEventArgs)
    If exitFlag = False Then
        aC.EventProcess()
    Else
        eventTimer.Enabled = False
    End If
End Sub

'=====  

'Function      :   addRenameCheck  

'Purpose       :   This function checks if the given param name is present  

'               :   in model and prevent rename if it is present.  

'=====  

Public Sub addRenameCheck(ByVal paramName As String)
    Dim listenerObj As BracketListener
    Dim command As IpfcUICommand
    Try
        '=====  

        'Start the timer to call EventProcess at regular intervals  

        '=====  

        eventTimer = New Timers.Timer(200)
        eventTimer.Enabled = True
        AddHandler eventTimer.Elapsed, AddressOf Me.timeElapsed

        command = aC.Session.UIGetCommand("ProCmdModelRename")

        If Not command Is Nothing Then
            listenerObj = New BracketListener(aC.Session, paramName)
            command.AddActionListener(listenerObj)
        Else
            Throw New Exception("Command does not exist")
        End If
    End Sub

```

```
aC.AddActionListener(Me)
```

```
Catch ex As Exception
```

```
MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
```

```
End Try
```

```
End Sub
```

```
'=====
'Class      : BracketListener
'Purpose    : This class implements the IpfcUICommandBracketListener
'            Interface along with the correct client interface name.
'            The implemented method will be called when the user
'            tries to rename the model.
'=====
```

```
Private Class BracketListener
```

```
Implements IpfcUICommandBracketListener
```

```
Implements ICIPClientObject
```

```
Implements IpfcActionListener
```

```
Dim s As IpfcSession
```

```
Dim name As String
```

```
Public Sub New(ByRef session As IpfcSession, ByVal paramName As
String)
```

```
s = session
```

```
name = paramName
```

```
End Sub
```

```
Public Function GetClientInterfaceName() As String Implements
pfcls.ICIPClientObject.GetClientInterfaceName
```

```
GetClientInterfaceName = "IpfcUICommandBracketListener"
```

```
End Function
```

```
Public Sub OnAfterCommand() Implements
```

```
pfcls.IpfcUICommandBracketListener.OnAfterCommand
```

```
End Sub
```

```
Public Sub OnBeforeCommand() Implements
```

```
pfcls.IpfcUICommandBracketListener.OnBeforeCommand
```

```
Dim param As IpfcParameter
```

```
Dim model As IpfcModel
```

```
Dim cancelAction As CCpfcXCancelProEAction
```

```
model = s.CurrentModel
```

```
If model Is Nothing Then
```

```
Return
```

```
End If
```

```
param = CType(model, IpfcParameterOwner).GetParam(name)
```

```
If Not param Is Nothing Then
```

```
cancelAction = New CCpfcXCancelProEAction
```

```
cancelAction.Throw()
```

```
End If
```

```
End Sub
```

```
End Class
```

```
End Class
```

# Designating Commands

Using the VB API you can designate Pro/ENGINEER commands to be available to be added to any Pro/ENGINEER toolbar.

To add a command to the toolbar, you must:

1. Define or add the command to be initiated on clicking the icon in the VB application.
2. Optionally designate an icon button to be used with the command.
3. Designate the command to appear in the Screen Customization dialog box of Pro/ENGINEER.
4. Finally, enter the **Screen Customization** dialog, and manually add the designated command to the window. Save the configuration in Pro/ENGINEER so that changes to the toolbar appear when a new session of Pro/ENGINEER is started.

## Command Icons

Method Introduced:

- **IpfcUICommand.SetIcon()**

The method **IpfcUICommand.SetIcon()** allows you to designate an icon to be used with the command you created. The method adds the icon to the Pro/ENGINEER command. Specify the name of the icon file, including the extension as the input argument for this method. A valid format for the icon file is the PTC-proprietary format used by Pro/ENGINEER.*BIF* or a standard *.GIF*. The Pro/ENGINEER toolbar button is replaced with the image of the image.

**Note:**

While specifying the name of the icon file, do not specify the full path to the icon names.

The default search paths for finding the icons are:

- <ProENGINEER loadpoint>/text/resource
- <Application text dir>/resource
- <Apppplication text dir>/(<language>)/resource

The location of the application text directory is specified in the registry file.

Toolbar commands that do not have an icon assigned to them display the button label.

You may also use this method to assign a small icon to a menu button on the menubar. The icon appears to the left of the button label.

Before using the method **IpfcUICommand.SetIcon()**, you must place the command in a menu using the method **IpfcSession.UIAddButton()**.

## Designating the Command

Method Introduced:

- **IpfcUICommand.Designate()**

This method allows you designate the command as available in the Screen Customization dialog box of Pro/ENGINEER. After a VB API application has used the method **IpfcUICommand.Designate()** on a command, you can interactively drag the toolbar button that you associate with the command, on to the Pro/ENGINEER toolbar. If this method is not called, the toolbar button will not be visible in the Screen Customization dialog box of Pro/ENGINEER.

The arguments to this method are:

- 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.
- 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.
- MessageFile--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>.

Before using the method **IpfcUICommand.Designate()**, you must place the command in a menu using the method **IpfcSession.UIAddButton()**.

## Placing the Toolbar Button

Once the toolbar button has been created using the functions discussed above, 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 in the following figure. Save the window configuration settings in the *config.win* file so that the settings are loaded when a new session of Pro/ENGINEER is launched. For more information, see the Pro/ENGINEER menus portion of the Pro/ENGINEER help.

**Figure 6-1: The Customize Screen With The Icons To be Designated**

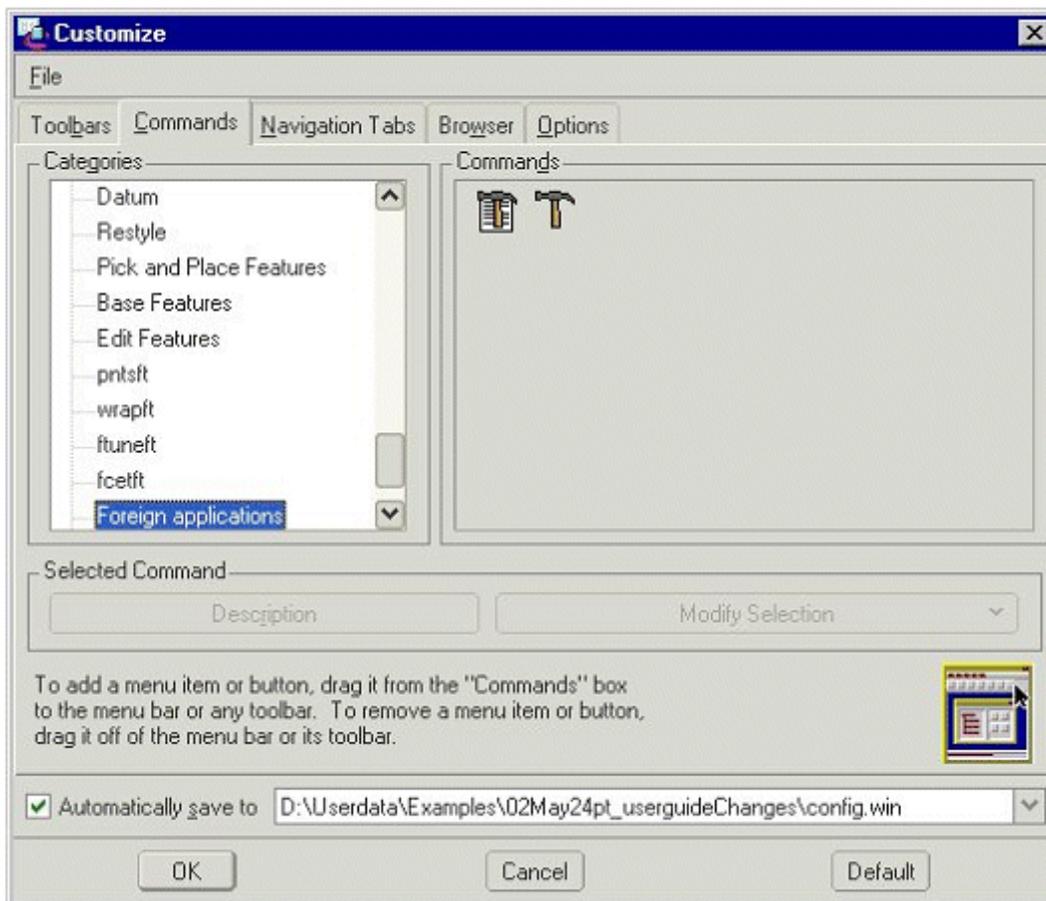


Figure 6-2: The Pro/ENGINEER Toolbar With The Designated Icons



## Pop-up Menus

Pro/ENGINEER provides shortcut menus that contain frequently used commands appropriate to the currently selected items. You can access a shortcut menu by right-clicking a selected item. Shortcut menus are accessible in:

- o Graphics window
- o Model Tree
- o Some dialog boxes
- o Any area where you can perform an object-action operation by selecting an item and choosing a command to perform on the selected item.

The methods described in this section allow you to add menus to a graphics window pop-up menu.

## Adding a Pop-up Menu to the Graphics Window

You can activate different pop-up menus during a given session of Pro/ENGINEER. Every time the Pro/ENGINEER context changes when you open a different model type, enter different tools or special modes such as **Edit**, a different pop-up menu is created. When Pro/ENGINEER moves to the next context, the pop-up menu may be destroyed.

As a result of this, the VB API applications must attach a button to the pop-up menu during initialization of the pop-up menu. The the VB API application is notified each time a particular pop-up menu is created, which then allows the user to add to the pop-up menu.

Use the following procedure to add items to pop-up menus in the graphics window:

1. Obtain the name of the existing pop-up menus to which you want to add a new menu using the trail file.
2. Create commands for the new pop-up menu items.
3. Implement access listeners to provide visibility information for the items.
4. Add an action listener to the session to listen for pop-up menu initialization.
5. In the listener method, if the pop-up menu is the correct menu to which you wish to add the button, then add it.

The following sections describe each of these steps in detail. You can add push buttons and cascade menus to the pop-up menus. You can add pop-up menu items only in the active window. You cannot use this procedure to remove items from existing menus.

## Using the Trail File to Determine Existing Pop-up Menu Names

The trail file in Pro/ENGINEER contains a comment that identifies the name of the pop-up menu if the configuration option, `auxapp_popup_menu_info` is set to `yes`.

For example, the pop-up menu, **Edit Properties**, has the following comment in the trail file:

```
~ Close `rmb_popup` `PopupMenu`  
~ Activate `rmb_popup` `EditProperties`  
!Command ProCmdEditPropertiesDtm was pushed from the software.  
!Item was selected from popup menu 'popup_mnu_edit'
```

## Listening for Pop-up Menu Initialization

Methods Introduced:

- **`IpfcActionSource.AddActionListener()`**
- **`IpfcPopupmenuListener.OnPopupmenuCreate()`**

Use the method **`IpfcActionSource.AddActionListener()`** to register a new `pfcUI.PopupmenuListener` to the session. This listener will be called when pop-up menus are initialized.

The method **`IpfcPopupmenuListener.OnPopupmenuCreate()`** is called after the pop-up menu is created internally in Pro/ENGINEER and may be used to assign application-owned buttons to the pop-up menu.

## Accessing the Pop-up Menus

The method described in this section provides the name of the pop-up menus used to access these menus while using other methods.

Method Introduced:

- **`IpfcPopupmenu.Name`**

The property **IpfcPopupmenu.Name** returns the name of the pop-up menu.

## Adding Content to the Pop-up Menus

Methods Introduced:

- **IpfcPopupmenu.AddButton()**
- **IpfcPopupmenu.AddMenu()**

Use **IpfcPopupmenu.AddButton()** to add a new item to a pop-up menu. The input arguments are:

- Command--Specifies the command associated with the pop-up menu.
- Options - A `pfcUI.PopupmenuOptions` object containing other options for the method. The options that may be included are:
  - PositionIndex--Specifies the position in the pop-up menu at which to add the menu button. Pass null to add the button to the bottom of the menu. Use the property `IpfcPopupmenuOptions.PositionIndex` to set this option.
  - Name--Specifies the name of the added button. The button name is placed in the trail file when the user selects the menu button. Use the property `IpfcPopupmenuOptions.Name` to set this option.
  - SetLabel--Specifies the button label. This label identifies the text displayed when the button is displayed. Use the property `IpfcPopupmenuOptions.Label` to set this option.
  - Helptext--Specifies the help message associated with the button. Use the property `IpfcPopupmenuOptions.Helptext` to set this option.

Use the method **IpfcPopupmenu.AddMenu()** to add a new cascade menu to an existing pop-up menu.

The argument for this method is a **pfcUI.PopupmenuOptions** object, whose members have the same purpose as described for newly added buttons. This method returns a new `pfcUI.Popupmenu` object to which you may add new buttons.

### Example 4: Creating a Pop-up Menu

This example code demonstrates the usage of UI functions to add a new model tree pop-up menu.

```
Public Class pfcPopupMenuExamples
    Implements IpfcAsyncActionListener
    Implements ICIPClientObject
    Implements IpfcActionListener

    Dim WithEvents eventTimer As Timers.Timer
    Dim exitFlag As Boolean = False
    Dim aC As pfcls.IpfcAsyncConnection

    Public Sub New(ByRef asyncConnection As pfcls.IpfcAsyncConnection)
        aC = asyncConnection
    End Sub

    Public Function GetClientInterfaceName() As String Implements
        pfcls.ICIPClientObject.GetClientInterfaceName
        GetClientInterfaceName = "IpfcAsyncActionListener"
    End Function

    Public Sub OnTerminate(ByVal _Status As Integer) Implements
```

```

                pfcls.IpfcAsyncActionListener.OnTerminate
aC.InterruptEventProcessing()
exitFlag = True
End Sub

'=====
'Function      :   timeElapsed
'Purpose      :   This function handles the time elapsed event of timer
'              :   which is fired at regular intervals
'=====
Private Sub timeElapsed(ByVal sender As Object, ByVal e As
                        System.Timers.ElapsedEventArgs)
    If exitFlag = False Then
        aC.EventProcess()
    Else
        eventTimer.Enabled = False
    End If
End Sub

'=====
'Function      :   addMenus
'Purpose      :   This function demonstrates the usage of UI functions to
'              :   add a new button to ProE Graphics Window and model tree
'              :   popup menu.
'=====
Public Sub addMenus()
    Dim inputCommand As IpfcUICommand
    Dim buttonListener As IpfcUICommandActionListener
    Dim popListener As IpfcPopupmenuListener
    Dim listenerObj As IpfcUICommandAccessListener

    Try

'=====
        'Start the timer to call EventProcess at regular intervals
'=====
        eventTimer = New Timers.Timer(200)
        eventTimer.Enabled = True
        AddHandler eventTimer.Elapsed, AddressOf Me.timeElapsed

'=====
        'Command is created which will be associated with the button.
        'The class implementing the actionlistener must be given as
        'input.
'=====
        buttonListener = New AssemblyFunction(aC.Session)
        inputCommand = aC.Session.UICreateCommand("HIGHLIGHT",
                                                buttonListener)

'=====
        'Add action listener to restrict access.
'=====
        listenerObj = New CheckAccess(aC)
        inputCommand.AddActionListener(listenerObj)

'=====
        'Button is created in Graphics Window and model tree popup
        'menu.
'=====

```

```
aC.Session.UIAddButton(inputCommand, "ActionMenu", Nothing, _
    "USER Highlight Constraint", "USER Highlight
    Constraint Help", "pfcPopupMenuExamples.txt")
```

```
popListener = New CreatePopupButton(aC.Session)
aC.Session.AddActionListener(popListener)
```

```
aC.AddActionListener(Me)
```

```
Catch ex As Exception
    MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
End Try
```

```
End Sub
```

```
'=====
'Class      :   AssemblyFunction
'Purpose    :   Listener class to implement Highlight command.
'=====
Private Class AssemblyFunction
    Implements pfcls.IpfcUICommandActionListener
    Implements ICIPClientObject

    Dim session As IpfcBaseSession

    Public Function GetClientInterfaceName() As String _
        Implements ICIPClientObject.GetClientInterfaceName
        GetClientInterfaceName = "IpfcUICommandActionListener"
    End Function

    Public Sub New(ByRef currentSession As IpfcBaseSession)
        session = currentSession
    End Sub

    Public Sub OnCommand() Implements
        pfcls.IpfcUICommandActionListener.OnCommand
        highlightConstraints()
    End Sub

'=====
'Function   :   highlightConstraints
'Purpose    :   This function displays each constraint of the
                component visually on the screen, and includes a
                text explanation for each constraint.
'=====
    Public Sub highlightConstraints()

        Dim options As IpfcSelectionOptions
        Dim selections As IpfcSelections
        Dim item As IpfcModelItem
        Dim feature As IpfcFeature
        Dim asmComp As IpfcComponentFeat
        Dim compConstraints As CpfcComponentConstraints
        Dim i As Integer
        Dim compConstraint As IpfcComponentConstraint
        Dim asmReference As IpfcSelection
        Dim compReference As IpfcSelection
        Dim offset As String
        Dim constraintType As String
        Dim selectionBuffer As IpfcSelectionBuffer
```

```

Dim modelPath As IpfcComponentPath
Dim parentPath As IpfcComponentPath
Dim parentIds As Cintseq
Dim modelParent As IpfcSolid
Dim modelId As Integer

```

```

Try

```

```

=====
'Get selected component
=====
selectionBuffer = session.CurrentSelectionBuffer
selections = selectionBuffer.Contents
If selections Is Nothing Then
    options = (New
                CCpfcSelectionOptions).Create("membfeat")
    options.MaxNumSels = 1

    selections = session.Select(options, Nothing)
    If selections Is Nothing OrElse selections.Count = 0
    Then
        Throw New Exception("Nothing Selected")
    End If
End If

=====
'Get ModelItem from the selected model
=====
item = selections.Item(0).SelItem

If item Is Nothing Then
    modelPath = selections.Item(0).Path

    modelId =
modelPath.ComponentIds.Item(modelPath.ComponentIds.Count - 1)

    parentIds = modelPath.ComponentIds
    parentIds.Remove(parentIds.Count - 1, parentIds.Count)

    If Not parentIds.Count = 0 Then
        parentPath = (New
CMpfcAssembly).CreateComponentPath(modelPath.Root, parentIds)
        modelParent = parentPath.Leaf
    Else
        modelParent = modelPath.Root
    End If

    item = CType(modelParent,
IpfcModelItemOwner).GetItemById(EpfcModelItemType.EpfcITEM_FEATURE,
modelId)

    If item Is Nothing Then
        Throw New Exception("Could not get model item
                            handle")
    End If

End If

feature = CType(item, IpfcFeature)
If Not feature.FeatType =
    EpfcFeatureType.EpfcFEATTYPE_COMPONENT Then

```

```
        Throw New Exception("Assembly Component not Selected")
    End If
```

```
'=====
'Get constraints for the component
'=====
asmComp = CType(item, IpfcComponentFeat)
compConstraints = asmComp.GetConstraints()
If compConstraints Is Nothing OrElse compConstraints.Count
    = 0 Then
    Throw New Exception("No Constraints to display")
End If

'=====
'Loop through all the constraints
'=====
For i = 0 To compConstraints.Count - 1

    compConstraint = compConstraints.Item(i)

'=====
'Highlight the assembly reference geometry
'=====
asmReference = compConstraint.AssemblyReference
If Not asmReference Is Nothing Then
asmReference.Highlight(EpfcStdColor.EpfcCOLOR_ERROR)
End If

'=====
'Highlight the component reference geometry
'=====
compReference = compConstraint.ComponentReference
If Not asmReference Is Nothing Then
compReference.Highlight(EpfcStdColor.EpfcCOLOR_WARNING)
End If

'=====
'Prepare and display the message text
'=====
offset = ""
If Not compConstraint.Offset Is Nothing Then
    offset = ", offset of " +
                                compConstraint.Offset.ToString
End If
constraintType =
    constraintTypeToString(compConstraint.Type)
MsgBox("Showing constraint " + (i + 1).ToString + " of
    " + _
    compConstraints.Count.ToString + Chr(13).ToString + _
    constraintType + offset)

'=====
'Clean up the UI for the next constraint
'=====
If Not asmReference Is Nothing Then
    asmReference.UnHighlight()
End If

If Not compReference Is Nothing Then
```

```

        compReference.UnHighlight()
    End If
Next

Catch ex As Exception
    MsgBox(ex.Message.ToString + Chr(13) +
        ex.StackTrace.ToString)

Exit Sub
End Try

End Sub

'=====
'Function    :    constraintTypeToString
'Purpose    :    This function converts constraint type to string.
'=====
Public Function constraintTypeToString(ByVal type As Integer)
    As String

    Select Case (type)
        Case EpcfComponentConstraintType.EpcfASM_CONSTRAINT_MATE
            Return ("Mate")
        Case
            EpcfComponentConstraintType.EpcfASM_CONSTRAINT_MATE_OFF
            Return ("Mate Offset")
        Case EpcfComponentConstraintType.EpcfASM_CONSTRAINT_ALIGN
            Return ("Align")
        Case
            EpcfComponentConstraintType.EpcfASM_CONSTRAINT_ALIGN_OFF
            Return ("Align Offset")
        Case EpcfComponentConstraintType.EpcfASM_CONSTRAINT_INSERT
            Return ("Insert")
        Case EpcfComponentConstraintType.EpcfASM_CONSTRAINT_ORIENT
            Return ("Orient")
        Case EpcfComponentConstraintType.EpcfASM_CONSTRAINT_CSYS
            Return ("Csys")
        Case EpcfComponentConstraintType.EpcfASM_CONSTRAINT_TANGENT
            Return ("Tangent")
        Case
            EpcfComponentConstraintType.EpcfASM_CONSTRAINT_PNT_ON_SRF
            Return ("Point on Surf")
        Case
            EpcfComponentConstraintType.EpcfASM_CONSTRAINT_EDGE_ON_SRF
            Return ("Edge on Surf")
        Case
            EpcfComponentConstraintType.EpcfASM_CONSTRAINT_DEF_PLACEMENT
            Return ("Default")
        Case
            EpcfComponentConstraintType.EpcfASM_CONSTRAINT_SUBSTITUTE
            Return ("Substitute")
        Case
            EpcfComponentConstraintType.EpcfASM_CONSTRAINT_PNT_ON_LINE
            Return ("Point on Line")
        Case EpcfComponentConstraintType.EpcfASM_CONSTRAINT_FIX
            Return ("Fix")
        Case EpcfComponentConstraintType.EpcfASM_CONSTRAINT_AUTO
            Return ("Auto")
    End Select

    Return ("Unrecognized Type")

```

End Function

End Class

```
'=====
'Class      :   CreatePopupButton
'Purpose    :   Listener class to create Popup menu button when the
'            :   menu is created.
'=====
Private Class CreatePopupButton
    Implements IpfcActionListener
    Implements ICIPClientObject
    Implements IpfcPopupmenuListener

    Dim session As IpfcSession

    Public Function GetClientInterfaceName() As String _
        Implements ICIPClientObject.GetClientInterfaceName
        GetClientInterfaceName = "IpfcPopupmenuListener"
    End Function

    Public Sub New(ByRef currentSession As IpfcSession)
        session = currentSession
    End Sub

    Public Sub OnPopupmenuCreate(ByVal _Menu As pfcls.IpfcPopupmenu)
        Implements pfcls.IpfcPopupmenuListener.OnPopupmenuCreate

        Dim command As IpfcUICommand
        Dim options As IpfcPopupmenuOptions
        Dim cmdString As String
        Dim helpString As String

        If _Menu.Name = "Sel Obj Menu" Then
            command = session.UIGetCommand("HIGHLIGHT")
            If Not command Is Nothing Then
                options = (New
                    CCpfcPopupmenuOptions).Create("HIGHLIGHT_CONSTRAINTS")
                cmdString = session.GetMessageContents
                    ("pfcPopupMenuExamples.txt",
                    "USER Highlight Constraint", Nothing)
                helpString = session.GetMessageContents
                    ("pfcPopupMenuExamples.txt", "USER Highlight
                    Constraint Help", Nothing)

                options.Helptext = helpString
                options.Label = cmdString
                _Menu.AddButton(command, options)
            Else
                Throw New Exception("HIGHLIGHT command does not exist")
            End If
        End If

    End Sub
End Class

'=====
'Class      :   CheckAccess
```

```
'Purpose      : This listener class checks if command is accessible to
'              the user.
```

```
'=====
Private Class CheckAccess
    Implements ICIPClientObject
    Implements IpfcUICommandAccessListener
    Implements IpfcActionListener

    Dim aC As pfcls.IpfcAsyncConnection

    Public Sub New(ByRef asyncConnection As IpfcAsyncConnection)
        aC = asyncConnection
    End Sub

    Public Function GetClientInterfaceName() As String _
        Implements ICIPClientObject.GetClientInterfaceName
        GetClientInterfaceName = "IpfcUICommandAccessListener"
    End Function

    Public Function OnCommandAccess(ByVal _AllowErrorMessages As
        Boolean) As Integer Implements
        pfcls.IpfcUICommandAccessListener.OnCommandAccess
        Dim model As IpfcModel
        Dim selections As IpfcSelections
        Dim selectionBuffer As IpfcSelectionBuffer

        model = aC.Session.CurrentModel
        If model Is Nothing OrElse (Not model.Type =
            EpfcModelType.EpfcMDL_ASSEMBLY) Then
            Return EpfcCommandAccess.EpfcACCESS_UNAVAILABLE
        End If

'=====
        'Get selected component
'=====
        selectionBuffer = IpfcSession.CurrentSelectionBuffer
        selections = selectionBuffer.Contents
        If selections Is Nothing OrElse selections.Count > 1 Then
            Return EpfcCommandAccess.EpfcACCESS_UNAVAILABLE
        End If

        Return EpfcCommandAccess.EpfcACCESS_AVAILABLE

    End Function
End Class

End Class
```

The corresponding message file for the example program contains two text messages. The first is used as the pop-menu button name, the second as its help string.

```
USER#Highlight#Constraint
Highlight Constraint
#
#
USER#Highlight#Constraint#Help
Highlight Assembly Constraints
#
```

#

---

# Models

---

This section describes how to program on the model level using the VB API.

## Topic

[Overview of Model Objects](#)

[Getting a Model Object](#)

[Model Descriptors](#)

[Retrieving Models](#)

[Model Information](#)

[Model Operations](#)

[Running ModelCHECK](#)

## Overview of Model Objects

Models can be any Pro/ENGINEER file type, including parts, assemblies, drawings, sections, and layouts. The classes in the module pfcModel provide generic access to models, regardless of their type. The available methods enable you to do the following:

- Access information about a model.
- Open, copy, rename, and save a model.

## Getting a Model Object

Methods and Properties Introduced:

- **IpfcFamilyTableRow.CreateInstance()**
- **IpfcSelection.SelModel**
- **IpfcBaseSession.GetModel()**
- **IpfcBaseSession.CurrentModel**
- **IpfcBaseSession.ListModels()**
- **IpfcBaseSession.GetByRelationId()**
- **IpfcWindow.Model**

These methods get a model object that is already in session.

The property **IpfcSelection.SelModel** returns the model that was interactively selected.

The method **IpfcBaseSession.GetModel()** returns a model based on its name and type, whereas

**IpfcBaseSession.GetByRelationId()** returns a model in an assembly that has the specified integer identifier.

The property **IpfcBaseSession.CurrentModel** returns the current active model.

Use the method **IpfcBaseSession.ListModels()** to return a sequence of all the models in session.

For more methods that return solid models, refer to the section [Solid](#).

## Model Descriptors

Methods and Properties Introduced:

- **CCpfcModelDescriptor.Create()**
- **IpfcModelDescriptor.GenericName**
- **IpfcModelDescriptor.InstanceName**
- **IpfcModelDescriptor.Type**
- **IpfcModelDescriptor.Host**
- **IpfcModelDescriptor.Device**
- **IpfcModelDescriptor.Path**
- **IpfcModelDescriptor.FileVersion**
- **IpfcModelDescriptor.GetFullName()**
- **IpfcModel.FullName**

Model descriptors are data objects used to describe a model file and its location in the system. The methods in the model descriptor enable you to set specific information that enables Pro/ENGINEER to find the specific model you want.

The static utility method **CCpfcModelDescriptor.Create()** allows you to specify as data to be entered a model type, an instance name, and a generic name. The model descriptor constructs the full name of the model as a string, as follows:

```
String FullName = InstanceName+"<" + GenericName+">"; // As long as the
                                                    // generic name is
                                                    // not an empty
                                                    // string ("")
```

If you want to load a model that is not a family table instance, pass an empty string as the generic name argument so that the full name of the model is constructed correctly. If the model is a family table interface, you should specify both the instance and generic name.

**Note:**

You are allowed to set other fields in the model descriptor object but they may be ignored by some methods.

## Retrieving Models

Methods Introduced:

- **IpfcBaseSession.RetrieveModel()**
- **IpfcBaseSession.OpenFile()**
- **IpfcSolid.HasRetrievalErrors()**

These methods cause Pro/ENGINEER to retrieve the model that corresponds to the *IpfcModelDescriptor* argument.

The method **IpfcBaseSession.RetrieveModel()** brings the model into memory, but does not create a window for it, nor does it display the model anywhere.

The method **IpfcBaseSession.OpenFile()** brings the model into memory, opens a new window for it (or uses the base window, if it is empty), and displays the model.

**Note:**

`IpfcBaseSession.OpenFile()` actually returns a handle to the window it has created.

To get a handle to the model you need, use the property **IpfcWindow.Model**.

The method **IpfcSolid.HasRetrievalErrors()** returns a true value if the features in the solid model were suppressed during the **RetrieveModel** or **OpenFile** operations. The method must be called immediately after the **IpfcBaseSession.RetrieveModel()** method or an equivalent retrieval method.

### Example Code: Retrieving a Model

The following example code demonstrates how to retrieve a model.

```
Imports pfcls

Public Class pfModelsExamples

    Public Function retrieveModelFromStdDir(ByVal session As
                                         IpfcBaseSession,
                                         _ByVal modelName As String,
                                         _ByVal type As EpfcModelType,
                                         _ByVal stdpath As String) As
                                         IpfcModel

        Dim descModel As IpfcModelDescriptor
        Dim options As IpfcRetrieveModelOptions
        Dim model As IpfcModel
```

```

    Try
'=====
'Model is retrieved using a model descriptor object.
'This method loads the model identified by modelname and type from a
'standard directory location.
'=====
        options = (New CCpfcRetrieveModelOptions).Create
        options.AskUserAboutReps = False
        'descModel = (New CCpfcModelDescriptor).Create(type,
                                                    modelName, Nothing)
        descModel = (New CCpfcModelDescriptor).Create(type, Nothing,
                                                    Nothing)

        descModel.Path = stdpath
        model = session.RetrieveModelWithOpts(descModel, options)
'model = session.RetrieveModel(descModel)

        retrieveModelFromStdDir = model

    Catch ex As Exception
        MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
        Return Nothing
    End Try
End Function
End Class

```

## Model Information

Methods and Properties Introduced:

- **IpfcModel.FileName**
- **IpfcModel.CommonName**
- **IpfcModel.IsCommonNameModifiable()**
- **IpfcModel.FullName**
- **IpfcModel.GenericName**
- **IpfcModel.InstanceName**
- **IpfcModel.Origin**
- **IpfcModel.RelationId**
- **IpfcModel.Descr**
- **IpfcModel.Type**
- **IpfcModel.IsModified**
- **IpfcModel.Version**

- **IpfcModel.Revision**
- **IpfcModel.Branch**
- **IpfcModel.ReleaseLevel**
- **IpfcModel.VersionStamp**
- **IpfcModel.ListDependencies()**
- **IpfcModel.ListDeclaredModels()**
- **IpfcModel.CheckIsModifiable()**
- **IpfcModel.CheckIsSaveAllowed()**

The property **IpfcModel.FileName** retrieves the model file name in the "name"."type" format.

The property **IpfcModel.CommonName** retrieves the common name for the model. This name is displayed for the model in Windchill PDMLink.

Use the method **IpfcModel.IsCommonNameModifiable()** to identify if the common name of the model can be modified. You can modify the name for models that are not yet owned by Windchill PDMLink, or in certain situations if the configuration option `let_proe_rename_pdm_objects` is set to yes.

The property **IpfcModel.FullName** retrieves the full name of the model in the instance <generic> format.

The property **IpfcModel.GenericName** retrieves the name of the generic model. If the model is not an instance, this name must be NULL or an empty string.

The property **IpfcModel.InstanceName** retrieves the name of the model. If the model is an instance, this method retrieves the instance name.

The property **IpfcModel.Origin** returns the complete path to the file from which the model was opened. This path can be a location on disk from a Windchill workspace, or from a downloaded URL.

The property **IpfcModel.RelationId** retrieves the relation identifier of the specified model. It can be NULL.

The property **IpfcModel.Descr** returns the descriptor for the specified model. Model descriptors can be used to represent models not currently in session.

The property **IpfcModel.Type** returns the type of model in the form of the **IpfcModelType** object. The types of models are as follows:

- EpfcMDL\_ASSEMBLY--Specifies an assembly.
- EpfcMDL\_PART--Specifies a part.
- EpfcMDL\_DRAWING--Specifies a drawing.
- EpfcMDL\_2D\_SECTION--Specifies a 2D section.
- EpfcMDL\_LAYOUT--Specifies a layout.
- EpfcMDL\_DWG\_FORMAT--Specifies a drawing format.
- EpfcMDL\_MFG--Specifies a manufacturing model.

- `EpfcMDL_REPORT`--Specifies a report.
- `EpfcMDL_MARKUP`--Specifies a drawing markup.
- `EpfcMDL_DIAGRAM`--Specifies a diagram

The property **`IpfcModel.IsModified`** identifies whether the model has been modified since it was last saved.

The property **`IpfcModel.Version`** returns the version of the specified model from the PDM system. It can be `NULL`, if not set.

The property **`IpfcModel.Revision`** returns the revision number of the specified model from the PDM system. It can be `NULL`, if not set.

The property **`IpfcModel.Branch`** returns the branch of the specified model from the PDM system. It can be `NULL`, if not set.

The property **`IpfcModel.ReleaseLevel`** returns the release level of the specified model from the PDM system. It can be `NULL`, if not set.

The property **`IpfcModel.VersionStamp`** returns the version stamp of the specified model. The version stamp is a Pro/ENGINEER specific identifier that changes with each change made to the model.

The method **`IpfcModel.ListDependencies()`** returns a list of the first-level dependencies for the specified model in the Pro/ENGINEER workspace in the form of the **`IpfcDependencies`** object.

The method **`IpfcModel.ListDeclaredModels()`** returns a list of all the first-level objects declared for the specified model.

The method **`IpfcModel.CheckIsModifiable()`** identifies if a given model can be modified without checking for any subordinate models. This method takes a boolean argument *ShowUI* that determines whether the Pro/ENGINEER conflict resolution dialog box should be displayed to resolve conflicts, if detected. If this argument is false, then the conflict resolution dialog box is not displayed, and the model can be modified only if there are no conflicts that cannot be overridden, or are resolved by default resolution actions. For a generic model, if *ShowUI* is true, then all instances of the model are also checked.

The method **`IpfcModel.CheckIsSaveAllowed()`** identifies if a given model can be saved along with all of its subordinate models. The subordinate models can be saved based on their modification status and the value of the configuration option `save_objects`. This method also checks the current user interface context to identify if it is currently safe to save the model. Thus, calling this method at different times might return different results. This method takes a boolean argument *ShowUI*. Refer to the previous method for more information on this argument.

## Model Operations

Methods and Property Introduced:

- **`IpfcModel.Backup()`**
- **`IpfcModel.Copy()`**
- **`IpfcModel.CopyAndRetrieve()`**
- **`IpfcModel.Rename()`**

- **IpfcModel.Save()**
- **IpfcModel.Erase()**
- **IpfcModel.EraseWithDependencies()**
- **IpfcModel.Delete()**
- **IpfcModel.Display()**
- **IpfcModel.CommonName**

These model operations duplicate most of the commands available in the Pro/ENGINEER File menu.

The method **IpfcModel.Backup()** makes a backup of an object in memory to a disk in a specified directory.

The method **IpfcModel.Copy()** copies the specified model to another file.

The method **IpfcModel.CopyAndRetrieve()** copies the model to another name, and retrieves that new model into session.

The method **IpfcModel.Rename()** renames a specified model.

The method **IpfcModel.Save()** stores the specified model to a disk.

The method **IpfcModel.Erase()** erases the specified model from the session. Models used by other models cannot be erased until the models dependent upon them are erased.

The method **IpfcModel.EraseWithDependencies()** erases the specified model from the session and all the models on which the specified model depends from disk, if the dependencies are not needed by other items in session.

The method **IpfcModel.Delete()** removes the specified model from memory and disk.

The method **IpfcModel.Display()** displays the specified model. You must call this method if you create a new window for a model because the model will not be displayed in the window until you call **IpfcModel.Display**.

The property **IpfcModel.CommonName** modifies the common name of the specified model. You can modify this name for models that are not yet owned by Windchill PDMLink, or in certain situations if the configuration option `let_proe_rename_pdm_objects` is set to yes.

## Running ModelCHECK

ModelCHECK is an integrated application that runs transparently within Pro/ENGINEER. ModelCHECK uses a configurable list of company design standards and best modeling practices. You can configure ModelCHECK to run interactively or automatically when you regenerate or save a model.

Methods and Properties Introduced:

- **IpfcBaseSession.ExecuteModelCheck()**

- **CCpfcModelCheckInstructions.Create()**
- **IpfcModelCheckInstructions.ConfigDir**
- **IpfcModelCheckInstructions.Mode**
- **IpfcModelCheckInstructions.OutputDir**
- **IpfcModelCheckInstructions.ShowInBrowser**
- **IpfcModelCheckResults.NumberOfErrors**
- **IpfcModelCheckResults.NumberOfWarnings**
- **IpfcModelCheckResults.WasModelSaved**

You can run ModelCHECK from an external application using the method **IpfcBaseSession.**

**ExecuteModelCheck()**. This method takes the model *Model* on which you want to run ModelCHECK and instructions in the form of the object **IpfcModelCheckInstructions** as its input parameters. This object contains the following parameters:

- ConfigDir--Specifies the location of the configuration files. If this parameter is set to NULL, the default ModelCHECK configuration files are used.
- Mode--Specifies the mode in which you want to run ModelCHECK. The modes are:
  - MODELCHECK\_GRAPHICS--Interactive mode
  - MODELCHECK\_NO\_GRAPHICS--Batch mode
- OutputDir--Specifies the location for the reports. If you set this parameter to NULL, the default ModelCHECK directory, as per config\_init.mc, will be used.
- ShowInBrowser--Specifies if the results report should be displayed in the Web browser.

The method **CCpfcModelCheckInstructions.Create()** creates the **IpfcModelCheckInstructions** object containing the ModelCHECK instructions described above.

Use the methods and properties **IpfcModelCheckInstructions.ConfigDir**, **IpfcModelCheckInstructions.Mode**, **IpfcModelCheckInstructions.OutputDir**, and **IpfcModelCheckInstructions.ShowInBrowser** to modify the ModelCHECK instructions.

The method **IpfcBaseSession.ExecuteModelCheck()** returns the results of the ModelCHECK run in the form of the **IpfcModelCheckResults** object. This object contains the following parameters:

- NumberOfErrors--Specifies the number of errors detected.
- NumberOfWarnings--Specifies the number of warnings found.
- WasModelSaved--Specifies whether the model is saved with updates.

Use the properties **IpfcModelCheckResults.NumberOfErrors**, **IpfcModelCheckResults.NumberOfWarnings**, **IpfcModelCheckResults.WasModelSaved** to access the results obtained.

## Custom Checks

This section describes how to define custom checks in ModelCHECK that users can run using the standard ModelCHECK interface in Pro/ENGINEER.

To define and register a custom check:

1. Set the CUSTMTK\_CHECKS\_FILE configuration option in the start configuration file to a text file that stores the check definition. For example:

```
CUSTMTK_CHECKS_FILE text/custmtk_checks.txt
```

2. Set the contents of the CUSTMTK\_CHECKS\_FILE file to define the checks. Each check should list the following items:

- DEF\_<checkname>--Specifies the name of the check. The format must be CHKTK\_<checkname>\_<mode>, where mode is PRT, ASM, or DRW.
- TAB\_<checkname>--Specifies the tab category in the ModelCHECK report under which the check is classified. Valid tab values are:
  - INFO
  - PARAMETER
  - LAYER
  - FEATURE
  - RELATION
  - DATUM
  - MISC
  - VDA
  - VIEWS
- MSG\_<checkname>--Specifies the description of the check that appears in the lower part of the ModelCHECK report when you select the name.
- DSC\_<checkname>--Specifies the name of the check as it appears in the ModelCHECK report table.
- ERM\_<checkname>--If set to INFO, the check is considered an INFO check and the report table displays the text from the first item returned by the check, instead of a count of the items. Otherwise, this value must be included, but is ignored by Pro/ENGINEER.

See the [Example 1: Text File for Custom Checks](#) for a sample custom checks text file.

3. Add the check and its values to the ModelCHECK configuration file.
4. Register the ModelCHECK check from the VB API application.

**Note:**

Other than the requirements listed above, the VB API custom checks do not have access to the rest of the values in the ModelCHECK configuration files. All the custom settings specific to the check, such as start parameters, constants, and so on, must be supported by the user application and not ModelCHECK.

## Registering Custom Checks

Methods and Properties Introduced:

- **IpfcBaseSession.RegisterCustomModelCheck()**
- **CCpfcCustomCheckInstructions.Create()**
- **IpfcCustomCheckInstructions.CheckName**

- **IpfcCustomCheckInstructions.CheckLabel**
- **IpfcCustomCheckInstructions.Listener**
- **IpfcCustomCheckInstructions.ActionButtonLabel**
- **IpfcCustomCheckInstructions.UpdateButtonLabel**

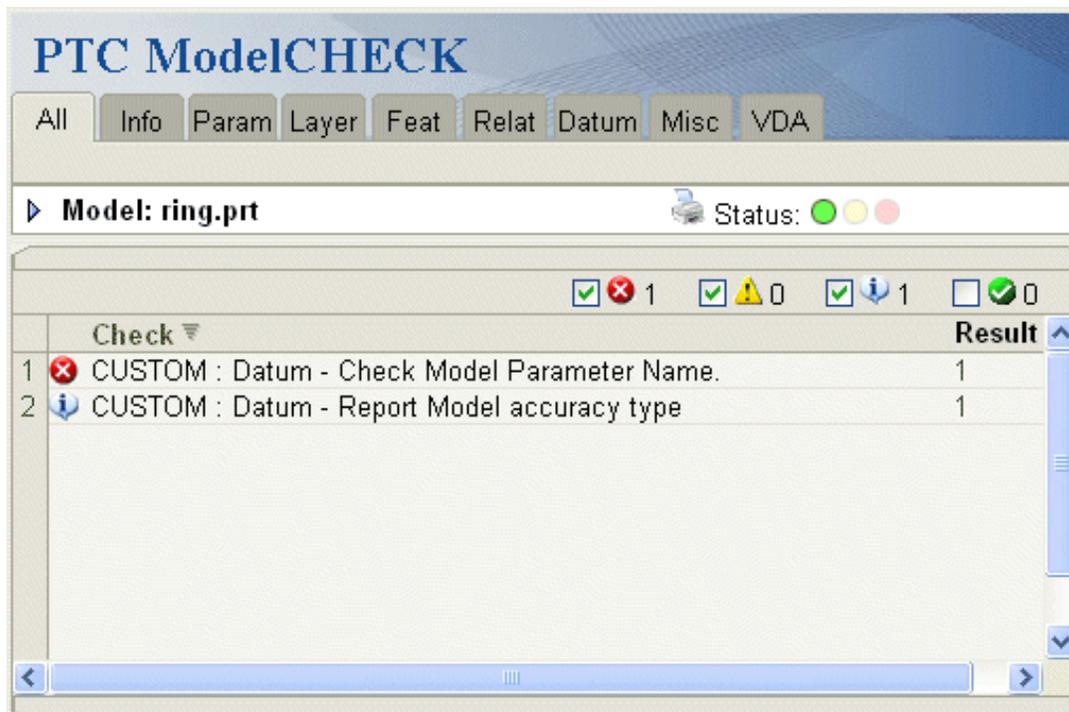
The method **IpfcBaseSession.RegisterCustomModelCheck()** registers a custom check that can be included in any ModelCHECK run. This method takes the instructions in the form of the **IpfcCustomCheckInstructions** object as its input argument. This object contains the following parameters:

- CheckName--Specifies the name of the custom check.
- CheckLabel--Specifies the label of the custom check.
- Listener--Specifies the listener object containing the custom check methods. Refer to the section Custom Check Listeners for more information.
- ActionButtonLabel--Specifies the label for the action button. If you specify NULL for this parameter, this button is not shown.
- UpdateButtonLabel--Specifies the label for the update button. If you specify NULL for this parameter, this button is not shown.

The method **CCpfcCustomCheckInstructions.Create()** creates the **IpfcCustomCheckInstructions** object containing the custom check instructions described above.

Use the methodsproperties **IpfcCustomCheckInstructions.CheckName**, **IpfcCustomCheckInstructions.CheckLabel**, **IpfcCustomCheckInstructions.Listener**, **IpfcCustomCheckInstructions.ActionButtonLabel**, and **IpfcCustomCheckInstructions.UpdateButtonLabel** to modify the instructions.

The following figure illustrates how the results of some custom checks might be displayed in the ModelCHECK report.



## Custom Check Listeners

Methods and Properties Introduced:

- **IpfcModelCheckCustomCheckListener.OnCustomCheck()**
- **CCpfcCustomCheckResults.Create()**
- **IpfcCustomCheckResults.ResultsCount**
- **IpfcCustomCheckResults.ResultsTable**
- **IpfcCustomCheckResults.ResultsUrl**
- **IpfcModelCheckCustomCheckListener.OnCustomCheckAction()**
- **IpfcModelCheckCustomCheckListener.OnCustomCheckUpdate()**

The interface **IpfcModelCheckCustomCheckListener** provides the method signatures to implement a custom ModelCheck check.

Each listener method takes the following input arguments:

- CheckName--The name of the custom check as defined in the original call to the method pfcSession.BaseSession.RegisterCustomModelCheck.
- Mdl--The model being checked.

The application method that overrides **IpfcModelCheckCustomCheckListener.OnCustomCheck()** is used to evaluate a custom defined check. The user application runs the check on the specified model and returns the results in the form of the **IpfcCustomCheckResults** object. This object contains the following parameters:

- ResultsCount--Specifies an integer indicating the number of errors found by the check. This value is displayed in the ModelCHECK report generated.
- ResultsTable--Specifies a list of text descriptions of the problem encountered for each error or warning.
- ResultsUrl--Specifies the link to an application-owned page that provides information on the results of the custom check.

The method **CCpfcCustomCheckResults.Create()** creates the **IpfcCustomCheckResults** object containing the custom check results described above.

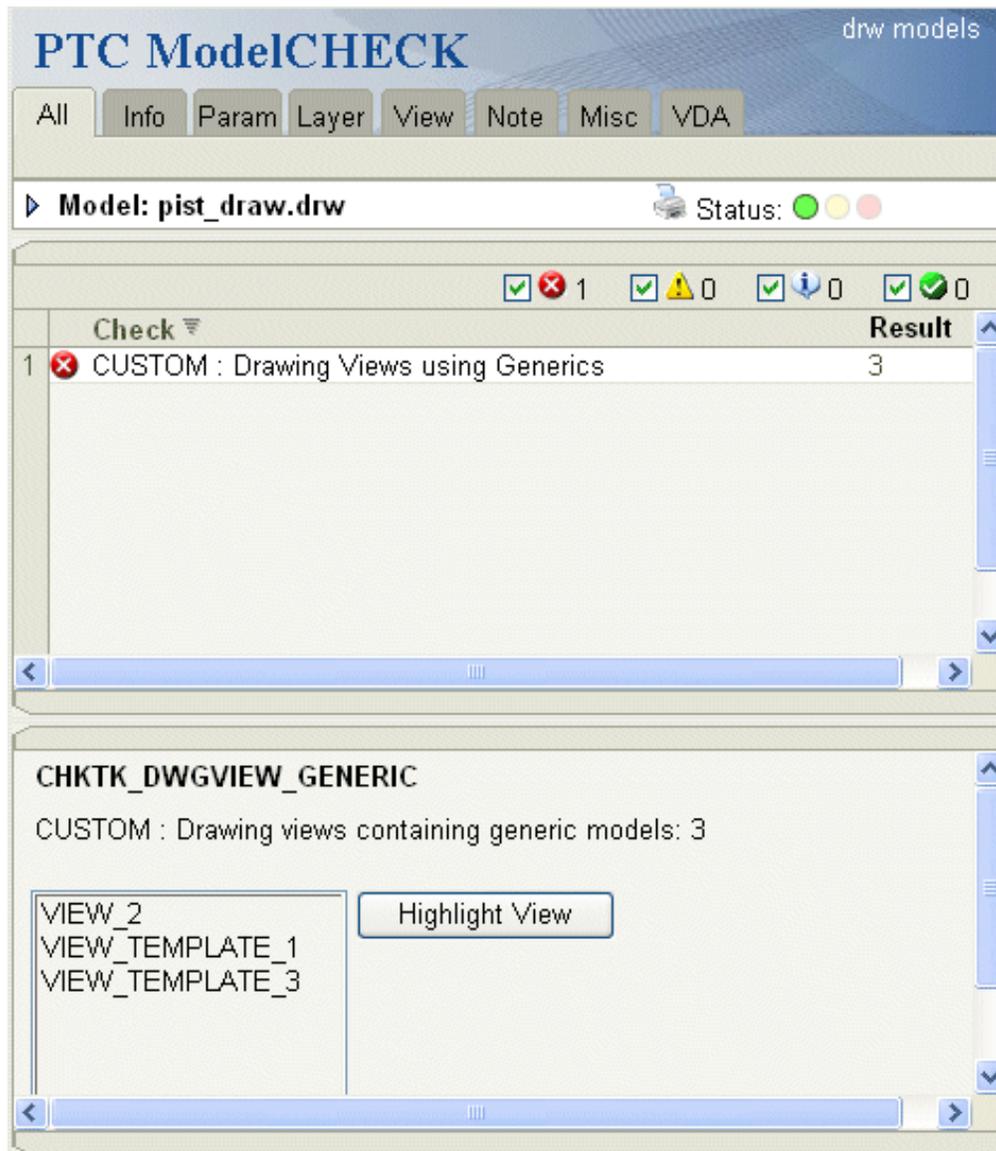
Use the properties **IpfcCustomCheckResults.ResultsCount**, **IpfcCustomCheckResults.ResultsTable**, and **IpfcCustomCheckResults.ResultsUrl** listed above to modify the custom checks results obtained.

The method that overrides **IpfcModelCheckCustomCheckListener.OnCustomCheckAction()** is called when the custom check's Action button is pressed. The input supplied includes the text selected by the user from the custom check results.

The function that overrides **IpfcModelCheckCustomCheckListener.OnCustomCheckUpdate()** is called when the custom check's Update button is pressed. The input supplied includes the text selected by the user from the custom check results.

Custom ModelCHECK checks can have an Action button to highlight the problem, and possibly an Update button to fix it automatically.

The following figure displays the ModelCHECK report with an Action button that invokes the **IpfcModelCheckCustomCheckListener.OnCustomCheckAction()** listener method.



### Example 1: Text File for Custom Checks

The following is the text file `custmtk_checks.txt` for custom checks examples.

```
# Custom TK Check File
# def-name of check as registered

# MDLPARAM_NAME
DEF_MDLPARAM_NAME CHKTK_MDLPARAM_NAME_PRT
TAB_MDLPARAM_NAME DATUM
MSG_MDLPARAM_NAME CUSTOM : Datum - Model Param Name
ERM_MDLPARAM_NAME CUSTOM : Datum - Invalid Parameter value.
DSC_MDLPARAM_NAME CUSTOM : Datum - Check Model Parameter Name.
```

```

# MODEL_ACCURACY
DEF_MODEL_ACCURACY CHKTK_MODEL_ACCURACY_PRT
TAB_MODEL_ACCURACY DATUM
MSG_MODEL_ACCURACY CUSTOM : Datum - Report Model accuracy type
ERM_MODEL_ACCURACY CUSTOM : Datum - Report Model accuracy type
DSC_MODEL_ACCURACY CUSTOM : Datum - Report Model accuracy type

# DWGVIEW_GENERIC
DEF_DWGVIEW_GENERIC CHKTK_DWGVIEW_GENERIC_DRW
TAB_DWGVIEW_GENERIC VIEWS
MSG_DWGVIEW_GENERIC CUSTOM : Drawing views containing generic models:
ERM_DWGVIEW_GENERIC N/A
DSC_DWGVIEW_GENERIC CUSTOM : Drawing Views using Generics

```

## Example 2: Registering Custom ModelCHECK Checks

This example demonstrates how to register custom ModelCHECK checks using the VB API. The following custom checks are registered:

- o CHKTK\_MDLPARAM\_NAME--Determines if the model has a parameter whose name is equal to the model name.
- o CHKTK\_MODEL\_ACCURACY--Checks the type of accuracy defined for the model.
- o CHKTK\_DWGVIEW\_GENERIC--Drawing mode check that identifies the drawing views that use generic models.

```

Dim aC As pfcls.IpfcAsyncConnection

Public Sub New(ByRef asyncConnection As pfcls.IpfcAsyncConnection)
    aC = asyncConnection
End Sub

'=====
'Function      :   addCheck
'Purpose      :   This function is used to register Model checks for part
'               and drawing models.
'=====
Public Sub addCheck(ByVal bParamName As Boolean, ByVal bAccType As
Boolean, _ByVal bDwgView As Boolean)

    Dim instructions As IpfcCustomCheckInstructions
    Dim paramCheck As ModelParamNameCheck
    Dim accCheck As ModelAccTypeCheck
    Dim drwCheck As ModelGenDrawViewCheck
    Dim checksAdded As Integer = 0

    Try
'=====
        'Create check instructions for Param Name
'=====
        If bParamName Then
            paramCheck = New ModelParamNameCheck(aC.Session)

```

```

instructions = (New CCpfcCustomCheckInstructions). _
                Create("CHKTK_MDLPARAM_NAME", _
                    "Model with invalid parameter : Datum", paramCheck)
instructions.UpdateButtonLabel = "Update Model Parameter"

'=====
'If the check is already registered, then do nothing
'=====
    Try
    aC.Session.RegisterCustomModelCheck(instructions)
    checksAdded = checksAdded + 1
    Catch ex As Exception
    If Not ex.Message.ToString = "pfcXToolkitFound" Then
        Throw ex
    End If
    End Try
End If

'=====
'Create check instructions for Accuracy Type
'=====
If bAccType Then
    accCheck = New ModelAccTypeCheck

    instructions = (New CCpfcCustomCheckInstructions). _
                    Create("CHKTK_MODEL_ACCURACY", _
                        "Check Model Accuracy : Datum", accCheck)
'=====
'If the check is already registered, then do nothing
'=====
    Try
    aC.Session.RegisterCustomModelCheck(instructions)
    checksAdded = checksAdded + 1
    Catch ex As Exception
    If Not ex.Message.ToString = "pfcXToolkitFound" Then
        Throw ex
    End If
    End Try

End If

'=====
'Create check instructions for Drawing View display
'=====
If bDwgView Then

    drwCheck = New ModelGenDrawViewCheck(aC.Session)
    instructions = (New CCpfcCustomCheckInstructions). _
                    Create("CHKTK_DWGVIEW_GENERIC", _
                        "Drawing Views Generic : View", drwCheck)
    instructions.ActionButtonLabel = "Highlight View"

'=====
'If the check is already registered, then do nothing
'=====

```

```

    Try
    aC.Session.RegisterCustomModelCheck(instructions)
    checksAdded = checksAdded + 1
    Catch ex As Exception
    If Not ex.Message.ToString = "pfcXToolkitFound" Then
        Throw ex
    End If
    End Try

End If

```

### Example 3: Implementing a Model Name Parameter Check

The following example defines the custom ModelCHECK check for the parameter name in a model. This check updates the parameter name to be equal to the model name, if the parameter exists with a different name, or creates the parameter with the model name if it does not exist.

```

'=====
'Class      :   ModelParamNameCheck
'Purpose    :   This class is used for checking if correct paramter is
'              present in model being checked and providing correction
'              actions if that is not the case
'=====
Private Class ModelParamNameCheck
    Implements ICIPClientObject
    Implements IpfcActionListener
    Implements IpfcModelCheckCustomCheckListener

    Const ParamName As String = "MDL_NAME_PARAM"
    Dim session As IpfcSession

    Public Sub New(ByVal asyncSession As IpfcSession)
        session = asyncSession
    End Sub

    Public Function GetClientInterfaceName() As String Implements
        pfcls.ICIPClientObject.GetClientInterfaceName
        GetClientInterfaceName = "IpfcModelCheckCustomCheckListener"
    End Function

'=====
'Function    :   OnCustomCheck
'Purpose     :   Check function for the Parameter ModelCheck
'=====
Public Function OnCustomCheck(ByVal _CheckName As String, ByVal _Mdl
    As pfcls.IpfcModel) As pfcls.IpfcCustomCheckResults
    Implements pfcls.IpfcModelCheckCustomCheckListener.OnCustomCheck

    Dim result As IpfcCustomCheckResults = Nothing
    Dim resultCount As Integer = 0
    Dim resultTable As Cstringseq
    Dim paramCompResult As Integer

```

```

Try
    paramCompResult = ModelParamNameCompare(_Mdl, ParamName)
    resultTable = New Cstringseq

    If paramCompResult = CORRECT_PARAM_VALUE Then
        resultTable = Nothing
        resultCount = 0
    Else
        resultCount = resultCount + 1

        Select Case paramCompResult

            Case MISSING_PARAM
                resultTable.Append("Parameter " + ParamName + _
                    " not found in the model " + _Mdl.FullName)

            Case INVALID_PARAM_TYPE
                resultTable.Append("Parameter " + ParamName + " in " + _
                    _Mdl.FullName + " is not a String parameter")

            Case INCORRECT_PARAM_VALUE
                resultTable.Append("Parameter " + ParamName + _
                    " value does not match model name " +
                    _Mdl.FullName)

        End Select
    End If

    result = (New CCpfcCustomCheckResults).Create(resultCount)
    result.ResultsTable = resultTable
    result.ResultsUrl = "http://www.ptc.com/"

    Return result

Catch ex As Exception
    MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
    Return Nothing
End Try
End Function
Public Sub OnCustomCheckAction(ByVal _CheckName As String, ByVal
    _Mdl As pfcls.IpfcModel, ByVal _SelectedItem As Object)
Implements pfcls.IpfcModelCheckCustomCheckListener.OnCustomCheckAction

End Sub

'=====
'Function      :   OnCustomCheckUpdate
'Purpose       :   Update function for the Parameter ModelCheck
'=====
Public Sub OnCustomCheckUpdate(ByVal _CheckName As String, ByVal
    _Mdl As pfcls.IpfcModel, ByVal _SelectedItem As Object)
Implements pfcls.IpfcModelCheckCustomCheckListener.OnCustomCheckUpdate

    Dim paramCompResult As Integer
    Dim param As IpfcParameter

```

```

Dim paramValue As IpfcParamValue
Dim message As Cstringseq

Try

    paramCompResult = ModelParamNameCompare(_Mdl, ParamName)
    If Not paramCompResult = CORRECT_PARAM_VALUE Then
        message = New Cstringseq
        message.Set(0, ParamName)

        Select Case paramCompResult

            Case MISSING_PARAM
                paramValue = (New
                    CmpfcModelItem).CreateStringParamValue(_Mdl.FullName)
                CType(_Mdl, IpfcParameterOwner).CreateParam(ParamName,
                    paramValue)
                session.UIDisplayMessage("pfcModelCheckExamples.txt", "UG
                    CustomCheck: MDL PARAM UPDATED", message)

            Case INCORRECT_PARAM_VALUE
                paramValue = (New
                    CmpfcModelItem).CreateStringParamValue(_Mdl.FullName)
                param = _Mdl.GetParam(ParamName)
                CType(param, IpfcBaseParameter).Value =
paramValue
                session.UIDisplayMessage("pfcModelCheckExamples.txt", "UG
                    CustomCheck: MDL PARAM UPDATED", message)

            Case INVALID_PARAM_TYPE
                session.UIDisplayMessage("pfcModelCheckExamples.txt", "UG
                    CustomCheck: MDL PARAM UPDATE TYPE", message)

        End Select
    End If

    Catch ex As Exception
        MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
    End Try
End Sub

Const MISSING_PARAM As Integer = 999
Const INVALID_PARAM_TYPE As Integer = 9999
Const CORRECT_PARAM_VALUE As Integer = 0
Const INCORRECT_PARAM_VALUE As Integer = 1
'=====
'Function      :   ModelParamNameCompare
'Purpose       :   Utility function to check if given param is present
                  in model and its value is equal to model name
'=====
Private Function ModelParamNameCompare(ByVal model As IpfcModel, _
                                        ByVal paramName As String) _
                                        As Integer

    Dim param As IpfcParameter
    Dim paramValue As IpfcParamValue

```

```

param = CType(model, IpfcParameterOwner).GetParam(paramName)
If param Is Nothing Then
    Return MISSING_PARAM
End If

paramValue = param.Value

If Not paramValue.dscr = EpfcParamValueType.EpfcPARAM_STRING Then
    Return INVALID_PARAM_TYPE
End If

If paramValue.StringValue = model.FullName Then
    Return CORRECT_PARAM_VALUE
Else
    Return INCORRECT_PARAM_VALUE
End If

End Function

End Class

```

#### Example 4: Implementing a Model Accuracy Type Check

The following example defines the custom ModelCHECK check for the type of accuracy whether relative or absolute that has been set for a model. This check has a check listener method, but no action or update listener method since it is an info-only check.

```

'=====
'Class      :   ModelAccTypeCheck
'Purpose    :   This class is used for checking which accuracy type has
'              been set, relative or absolute
'=====
Private Class ModelAccTypeCheck
    Implements ICIPClientObject
    Implements IpfcActionListener
    Implements IpfcModelCheckCustomCheckListener

    Public Function GetClientInterfaceName() As String Implements
        pfcls.ICIPClientObject.GetClientInterfaceName
        GetClientInterfaceName = "IpfcModelCheckCustomCheckListener"
    End Function

'=====
'Function   :   OnCustomCheck
'Purpose    :   Check function for the Model Accuracy Type
'=====
Public Function OnCustomCheck(ByVal _CheckName As String, ByVal _Mdl
    As pfcls.IpfcModel) As pfcls.IpfcCustomCheckResults
Implements pfcls.IpfcModelCheckCustomCheckListener.OnCustomCheck

    Dim result As IpfcCustomCheckResults = Nothing
    Dim resultCount As Integer = 0

```

```

Dim resultTable As Cstringseq
Dim accuracy As Object

Try
    resultCount = resultCount + 1
    resultTable = New Cstringseq

    accuracy = CType(_Mdl, IpfcSolid).AbsoluteAccuracy
    If accuracy Is Nothing Then
        resultTable.Append("Relative accuracy")
    Else
        resultTable.Append("Absolute Accuracy")
    End If

    result = (New CCpfcCustomCheckResults).Create(resultCount)
    result.ResultsTable = resultTable
    Return result

Catch ex As Exception
    MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
    Return Nothing
End Try
End Function

Public Sub OnCustomCheckAction(ByVal _CheckName As String, ByVal
    _Mdl As pfcls.IpfcModel, ByVal _SelectedItem As Object)
Implements pfcls.IpfcModelCheckCustomCheckListener.OnCustomCheckAction

End Sub

Public Sub OnCustomCheckUpdate(ByVal _CheckName As String, ByVal
    _Mdl As pfcls.IpfcModel, ByVal _SelectedItem As Object)
Implements pfcls.IpfcModelCheckCustomCheckListener.OnCustomCheckUpdate

End Sub

End Class

```

#### Example 4: Implementing a Check for Drawing Views Using Generic Models

The following example defines the custom ModelCHECK check for identifying drawing views using generic models. This check has a check listener method and an action listener method to highlight the views that use generic models.

```

'=====
'Class      :   ModelGenDrawViewCheck
'Purpose    :   This class is used for checking generics in drawing
'              views.  Outputs a list of the view names.
'=====
Private Class ModelGenDrawViewCheck
    Implements ICIPClientObject
    Implements IpfcActionListener

```

```
Implements IpfcModelCheckCustomCheckListener
```

```
Dim session As IpfcSession
```

```
Public Sub New(ByVal asyncSession As IpfcSession)  
    session = asyncSession  
End Sub
```

```
Public Function GetClientInterfaceName() As String Implements  
    pfcls.ICIPClientObject.GetClientInterfaceName  
    GetClientInterfaceName = "IpfcModelCheckCustomCheckListener"  
End Function
```

```
'=====
```

```
'Function      :   OnCustomCheck  
'Purpose       :   Check function for the generic Views in drawing
```

```
'=====
```

```
Public Function OnCustomCheck(ByVal _CheckName As String, ByVal _Mdl  
    As pfcls.IpfcModel) As pfcls.IpfcCustomCheckResults  
    Implements pfcls.IpfcModelCheckCustomCheckListener.OnCustomCheck
```

```
Dim result As IpfcCustomCheckResults = Nothing
```

```
Dim resultCount As Integer = 0
```

```
Dim resultTable As Cstringseq
```

```
Dim drawing As IpfcDrawing
```

```
Dim model As IpfcModel
```

```
Dim solid As IpfcSolid
```

```
Dim views As IpfcView2Ds
```

```
Dim view As IpfcView2D
```

```
Dim i As Integer
```

```
Try
```

```
    resultTable = New Cstringseq
```

```
    drawing = CType(_Mdl, IpfcDrawing)
```

```
    views = drawing.List2DViews
```

```
    For i = 0 To views.Count - 1
```

```
        view = views(i)
```

```
        model = view.GetModel()
```

```
        solid = CType(model, IpfcSolid)
```

```
        If solid.Parent Is Nothing Then
```

```
            resultCount = resultCount + 1
```

```
            resultTable.Append(view.Name)
```

```
        End If
```

```
    Next
```

```
    result = (New CCpfcCustomCheckResults).Create(resultCount)
```

```
    result.ResultsTable = resultTable
```

```
    Return result
```

```
    Catch ex As Exception
```

```
        MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
```

```
        Return Nothing
```

```
End Try
```

```
End Function
```

```

'=====
'Function      :   OnCustomCheck
'Purpose       :   Check function which highlights the selected
                   drawing view
'=====

Public Sub OnCustomCheckAction(ByVal _CheckName As String,
                               ByVal _Mdl As pfcls.IpfcModel, ByVal _SelectedItem As Object)
Implements pfcls.IpfcModelCheckCustomCheckListener.OnCustomCheckAction

    Const DELTA_X As Double = 10.0 'Screen coordinates
    Const DELTA_Y As Double = 10.0 'Screen coordinates
    Dim drawing As IpfcDrawing
    Dim view As IpfcView2D
    Dim outline As CpfcOutline3D
    Dim point0, point1, point2, point3, point4 As CpfcPoint3D
    Dim points As CpfcPoint3Ds

    Dim lineStyle As Integer
    Dim graphicsColour As Integer

Try
    If Not (_SelectedItem Is Nothing) Then
        drawing = CType(_Mdl, IpfcDrawing)
        view = drawing.GetViewByName(_SelectedItem.ToString)
        outline = view.Outline

        points = New CpfcPoint3Ds

        point0 = New CpfcPoint3D
        point0.Set(0, outline.Item(0).Item(0) - DELTA_X)
        point0.Set(1, outline.Item(0).Item(1) - DELTA_Y)
        point0.Set(2, 0.0)
        points.Insert(0, point0)

        point1 = New CpfcPoint3D
        point1.Set(0, outline.Item(1).Item(0) - DELTA_X)
        point1.Set(1, outline.Item(1).Item(1) + DELTA_Y)
        point1.Set(2, 0.0)
        points.Insert(1, point1)

        point2 = New CpfcPoint3D
        point2.Set(0, outline.Item(1).Item(0) + DELTA_X)
        point2.Set(1, outline.Item(1).Item(1) + DELTA_Y)
        point2.Set(2, 0.0)
        points.Insert(2, point2)

        point3 = New CpfcPoint3D
        point3.Set(0, outline.Item(1).Item(0) + DELTA_X)
        point3.Set(1, outline.Item(0).Item(1) - DELTA_Y)
        point3.Set(2, 0.0)
        points.Insert(3, point3)

        point4 = New CpfcPoint3D
        point4.Set(0, outline.Item(0).Item(0) - DELTA_X)
        point4.Set(1, outline.Item(0).Item(1) - DELTA_Y)

```

```

point4.Set(2, 0.0)
points.Insert(4, point4)

graphicsColour = session.CurrentGraphicsColor
session.CurrentGraphicsColor = EpfcStdColor.EpfcCOLOR_HIGHLIGHT

lineStyle =
    session.SetLineStyle(EpfcStdLineStyle.EpfcLINE_PHANTOM)

session.DrawPolyline(points)

lineStyle = session.SetLineStyle(lineStyle)
session.CurrentGraphicsColor = graphicsColour

End If

Catch ex As Exception
    MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
End Try

End Sub

Public Sub OnCustomCheckUpdate(ByVal _CheckName As String,
    ByVal _Mdl As pfcls.IpfcModel, ByVal _SelectedItem As Object)
Implements pfcls.IpfcModelCheckCustomCheckListener.OnCustomCheckUpdate

End Sub

End Class

```

## Example 6: Changes to the ModelCHECK Configuration Files to enable Custom Checks

Lines added to the ModelCheck configuration file (default\_checks.mch)

```

E    Check the item.  If not succeed, report as an error
W    Check the item.  If not succeed, report as a warning
N    Do not check the item.
Y    Check the item.  If not succeed, do not report err or warn

```

CHKTK_MDLPARAM_NAME_PRT	YNEW	E	E	E	E	Y
CHKTK_MODEL_ACCURACY_PRT	YNEW	Y	Y	Y	Y	Y
CHKTK_DWGVIEW_GENERIC_DRW	YNEW	E	E	E	E	Y

Lines added to the ModelCheck start file (sample\_start.mcs)

```

CUSTMTK_CHECKS_FILE      custmtk_checks.txt

```

---

# Drawings

---

This section describes how to program drawing functions using the VB API.

## Topic

[Overview of Drawings in the VB API](#)

[Creating Drawings from Templates](#)

[Obtaining Drawing Models](#)

[Drawing Information](#)

[Drawing Operations](#)

[Drawing Sheets](#)

[Drawing Views](#)

[Drawing Dimensions](#)

[Drawing Tables](#)

[Detail Items](#)

[Detail Entities](#)

[OLE Objects](#)

[Detail Notes](#)

[Detail Groups](#)

[Detail Symbols](#)

[Detail Attachments](#)

## Overview of Drawings in the VB API

This section describes the functions that deal with drawings. You can create drawings of all Pro/ENGINEER models using the functions in the VB API. You can annotate the drawing, manipulate dimensions, and use layers to manage the display of different items.

Unless otherwise specified, the VB API functions that operate on drawings use world units.

## Creating Drawings from Templates

Drawing templates simplify the process of creating a drawing using the VB API. Pro/ENGINEER can create views, set the view display, create snap lines, and show the model dimensions based on the template. Use templates to:

- Define layout views
- Set view display
- Place notes
- Place symbols
- Define tables
- Show dimensions

Method Introduced:

- **`IpfcBaseSession.CreateDrawingFromTemplate()`**

Use the method **`IpfcBaseSession.CreateDrawingFromTemplate()`** to create a drawing from the drawing template and to return the created drawing. The attributes are:

- o New drawing name
- o Name of an existing template
- o Name and type of the solid model to use while populating template views
- o Sequence of options to create the drawing. The options are as follows:
  - EpfcDRAWINGCREATE\_DISPLAY\_DRAWING--display the new drawing.
  - EpfcDRAWINGCREATE\_SHOW\_ERROR\_DIALOG--display the error dialog box.
  - EpfcDRAWINGCREATE\_WRITE\_ERROR\_FILE--write the errors to a file.
  - EpfcDRAWINGCREATE\_PROMPT\_UNKNOWN\_PARAMS--prompt the user on encountering unknown parameters.

## Drawing Creation Errors

The exception **XToolkitDrawingCreateErrors** is thrown if an error is encountered when creating a drawing from a template. This exception contains a list of errors which occurred during drawing creation.

### Note:

When this exception type is encountered, the drawing is actually created, but some of the contents failed to generate correctly.

The exception message will list the details for each error including its type, sheet number, view name, and (if applicable) item name, The types of errors are as follows:

- EpfcDWGCREATE\_ERR\_SAVED\_VIEW\_DOESNT\_EXIST--Saved view does not exist.
- EpfcDWGCREATE\_ERR\_X\_SEC\_DOESNT\_EXIST--Specified cross section does not exist.
- EpfcDWGCREATE\_ERR\_EXPLODE\_DOESNT\_EXIST--Exploded state did not exist.
- EpfcDWGCREATE\_ERR\_MODEL\_NOT\_EXPLODABLE--Model cannot be exploded.
- EpfcDWGCREATE\_ERR\_SEC\_NOT\_PERP--Cross section view not perpendicular to the given view.
- EpfcDWGCREATE\_ERR\_NO\_RPT\_REGIONS--Repeat regions not available.
- EpfcDWGCREATE\_ERR\_FIRST\_REGION\_USED--Repeat region was unable to use the region specified.
- EpfcDWGCREATE\_ERR\_NOT\_PROCESS\_ASSEM-- Model is not a process assembly view.
- EpfcDWGCREATE\_ERR\_NO\_STEP\_NUM--The process step number does not exist.
- EpfcDWGCREATE\_ERR\_TEMPLATE\_USED--The template does not exist.
- EpfcDWGCREATE\_ERR\_NO\_PARENT\_VIEW\_FOR\_PROJ--There is no possible parent view for this projected view.
- EpfcDWGCREATE\_ERR\_CANT\_GET\_PROJ\_PARENT--Could not get the projected parent for a drawing view.
- EpfcDWGCREATE\_ERR\_SEC\_NOT\_PARALLEL--The designated cross section was not parallel to the created view.
- EpfcDWGCREATE\_ERR\_SIMP\_REP\_DOESNT\_EXIST--The designated simplified representation does not exist.

### Example: Drawing Creation from a Template

The following code creates a new drawing using a predefined template.

```
Imports System.IO
Imports pfcls

Public Class pfcdrawingExamples
    Public Sub createDrawingFromTemplate(ByRef session As IpfcBaseSession,
                                       ByVal drawingName As String)
        Dim predefinedTemplate As String = "c_drawing"
        Dim model As IpfcModel
        Dim drawingOptions As New CpfcDrawingCreateOptions
        Dim drawing As IpfcDrawing
```

```

    Try
'=====
'Use the current model to create the drawing
'=====
        model = session.CurrentModel
        If model Is Nothing Then
            Throw New Exception("Model not present")
        End If

        drawingOptions.Insert(0, EpfcDrawingCreateOption.
EpfcDRAWINGCREATE_DISPLAY_DRAWING)
        drawingOptions.Insert(1, EpfcDrawingCreateOption.
EpfcDRAWINGCREATE_SHOW_ERROR_DIALOG)
        drawing = session.CreateDrawingFromTemplate(drawingName,
predefinedTemplate, _model.Descr, drawingOptions)

    Catch ex As Exception
        MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
    End Try

End Sub

End Class

```

## Obtaining Drawing Models

This section describes how to obtain drawing models.

Methods Introduced:

- **IpfcBaseSession.RetrieveModel()**
- **IpfcBaseSession.GetModel()**
- **IpfcBaseSession.GetModelFromDescr()**
- **IpfcBaseSession.ListModels()**
- **IpfcBaseSession.ListModelsByType()**

The method **IpfcBaseSession.RetrieveModel()** retrieves the drawing specified by the model descriptor. Model descriptors are data objects used to describe a model file and its location in the system. The method returns the retrieved drawing.

The method **IpfcBaseSession.GetModel()** returns a drawing based on its name and type, whereas **IpfcBaseSession.GetModelFromDescr()** returns a drawing specified by the model descriptor. The model must be in session.

Use the method **IpfcBaseSession.ListModels()** to return a sequence of all the drawings in session.

## Drawing Information

Methods and Property Introduced:

- **IpfcModel2D.ListModels()**

- **IpfcModel2D.GetCurrentSolid()**
- **IpfcModel2D.ListSimplifiedReps()**
- **IpfcModel2D.TextHeight**

The method **IpfcModel2D.ListModels()** returns a list of all the solid models used in the drawing.

The method **IpfcModel2D.GetCurrentSolid()** returns the current solid model of the drawing.

The method **IpfcModel2D.ListSimplifiedReps()** returns the simplified representations of a solid model that are assigned to the drawing.

The property **IpfcModel2D.TextHeight** returns the text height of the drawing.

## Drawing Operations

Methods Introduced:

- **IpfcModel2D.AddModel()**
- **IpfcModel2D.DeleteModel()**
- **IpfcModel2D.ReplaceModel()**
- **IpfcModel2D.SetCurrentSolid()**
- **IpfcModel2D.AddSimplifiedRep()**
- **IpfcModel2D.DeleteSimplifiedRep()**
- **IpfcModel2D.Regenerate()**
- **IpfcModel2D.CreateDrawingDimension()**
- **IpfcModel2D.CreateView()**

The method **IpfcModel2D.AddModel()** adds a new solid model to the drawing.

The method **IpfcModel2D.DeleteModel()** removes a model from the drawing. The model to be deleted should not appear in any of the drawing views.

The method **IpfcModel2D.ReplaceModel()** replaces a model in the drawing with a related model (the relationship should be by family table or interchange assembly). It allows you to replace models that are shown in drawing views and regenerates the view.

The method **IpfcModel2D.SetCurrentSolid()** assigns the current solid model for the drawing. Before calling this method, the solid model must be assigned to the drawing using the method **IpfcModel2D.AddModel()**. To see the changes to parameters and fields reflecting the change of the current solid model, regenerate the drawing using the method **IpfcSheetOwner.RegenerateSheet()**.

The method **IpfcModel2D.AddSimplifiedRep()** associates the drawing with the simplified representation of an assembly .

The method **IpfcModel2D.DeleteSimplifiedRep()** removes the association of the drawing with an assembly simplified representation. The simplified representation to be deleted should not appear in any of the drawing views.

Use the method **IpfcModel2D.Regenerate()** to regenerate the drawing draft entities and appearance.

The method **IpfcModel2D.CreateDrawingDimension()** creates a new drawing dimension based on the data object that contains information about the location of the dimension. This method returns the created dimension. Refer to the section [Drawing Dimensions](#).

The method **IpfcModel2D.CreateView()** creates a new drawing view based on the data object that contains information about how to create the view. The method returns the created drawing view. Refer to the section [Creating Drawing Views](#).

**Example: Replace Drawing Model Solid with its Generic**

The following code replaces all solid model instances in a drawing with its generic. Models are not replaced if the generic model is already present in the drawing.

```
Imports System.IO
Imports pfcls

Public Class pfcdrawingexamples

    Public Sub replaceModels(ByRef session As IpfcBaseSession)
        Dim model As IpfcModel
        Dim models As IpfcModels
        Dim drawing As IpfcDrawing
        Dim solid As IpfcSolid
        Dim generic As IpfcSolid
        Dim i As Integer

        Try
            '=====
            'Get the current model to create the drawing
            '=====
                model = session.CurrentModel
                If model Is Nothing Then
                    Throw New Exception("Model not present")
                End If
                If Not model.Type = EpfcModelType.EpfcMDL_DRAWING Then
                    Throw New Exception("Model is not drawing")
                End If
                drawing = CType(model, IpfcDrawing)
            '=====
            'Visit the drawing models
            '=====
                models = drawing.ListModels()
            '=====
            'Loop on all of the drawing models
            '=====
                For i = 0 To models.Count - 1
                    solid = CType(models.Item(i), IpfcSolid)
                    generic = solid.Parent

                    If Not generic Is Nothing Then
            '=====
```

```
'Replace all instances with their generic
'=====
        drawing.ReplaceModel(solid, generic, True)
    End If
Next

Catch ex As Exception
    MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
End Try
End Sub
```

## Drawing Sheets

A drawing sheet is represented by its number. Drawing sheets in the VB API are identified by the same sheet numbers seen by a Pro/Engineer user.

### Note:

These identifiers may change if the sheets are moved as a consequence of adding, removing or reordering sheets.

## Drawing Sheet Information

Methods and Properties Introduced

- **IpfcSheetOwner.GetSheetData()**
- **IpfcSheetOwner.GetSheetTransform()**
- **IpfcSheetOwner.GetSheetScale()**
- **IpfcSheetOwner.GetSheetFormat()**
- **IpfcSheetOwner.GetSheetBackgroundView()**
- **IpfcSheetOwner.NumberOfSheets**
- **IpfcSheetOwner.CurrentSheetNumber**
- **IpfcSheetOwner.GetSheetUnits()**

The method **IpfcSheetOwner.GetSheetData()** returns sheet data including the size, orientation, and units of the sheet specified by the sheet number.

The method **IpfcSheetOwner.GetSheetTransform()** returns the transformation matrix for the sheet specified by the sheet number. This transformation matrix includes the scaling needed to convert screen coordinates to drawing coordinates (which use the designated drawing units).

The method **IpfcSheetOwner.GetSheetScale()** returns the scale of the drawing on a particular sheet based on the drawing model used to measure the scale. If no models are used in the drawing then the default scale value is 1.0.

The method **IpfcSheetOwner.GetSheetFormat()** returns the drawing format used for the sheet specified by the sheet number. It returns a null value if no format is assigned to the sheet.

The method **IpfcSheetOwner.GetSheetBackgroundView()** returns the view object representing the background view of the sheet specified by the sheet number.

The property **IpfcSheetOwner.NumberOfSheets** returns the number of sheets in the model.

The property **IpfcSheetOwner.CurrentSheetNumber** returns the current sheet number in the model.

**Note:**

The sheet numbers range from 1 to n, where n is the number of sheets.

The method **IpfcSheetOwner.GetSheetUnits()** returns the units used by the sheet specified by the sheet number.

## Drawing Sheet Operations

Methods Introduced:

- **IpfcSheetOwner.AddSheet()**
- **IpfcSheetOwner.DeleteSheet()**
- **IpfcSheetOwner.ReorderSheet()**
- **IpfcSheetOwner.RegenerateSheet()**
- **IpfcSheetOwner.SetSheetScale()**
- **IpfcSheetOwner.SetSheetFormat()**

The method **IpfcSheetOwner.AddSheet()** adds a new sheet to the model and returns the number of the new sheet.

The method **IpfcSheetOwner.DeleteSheet()** removes the sheet specified by the sheet number from the model.

Use the method **IpfcSheetOwner.ReorderSheet()** to reorder the sheet from a specified sheet number to a new sheet number.

**Note:**

The sheet number of other affected sheets also changes due to reordering or deletion.

The method **IpfcSheetOwner.RegenerateSheet()** regenerates the sheet specified by the sheet number.

**Note:**

You can regenerate a sheet only if it is displayed.

Use the method **IpfcSheetOwner.SetSheetScale()** to set the scale of a model on the sheet based on the drawing model to scale and the scale to be used. Pass the value of the *DrawingModel* parameter as null to select the current drawing model.

Use the method **IpfcSheetOwner.SetSheetFormat()** to apply the specified format to a drawing sheet based on the drawing format, sheet number of the format, and the drawing model.

The sheet number of the format is specified by the *FormatSheetNumber* parameter. This number ranges from 1 to the number of sheets in the format. Pass the value of this parameter as null to use the first format sheet.

The drawing model is specified by the *DrawingModel* parameter. Pass the value of this parameter as null to select the current drawing model.

**Example: Listing Drawing Sheets**

The following example shows how to list the sheets in the current drawing. The information is placed in an external browser window.

```
Public Sub listSheets(ByRef session As IpfcBaseSession, ByVal fileName
                    As String)
    Dim file As StreamWriter = Nothing
    Dim formatName As String
    Dim model As IpfcModel
    Dim drawing As IpfcDrawing
    Dim sheets As Integer
    Dim i As Integer
    Dim sheetData As IpfcSheetData
    Dim sheetFormat As IpfcDrawingFormat
    Dim unit As String

    Try
        '=====
        'Create file to store information to be displayed
        '=====
            file = New StreamWriter(fileName)
            file.WriteLine("<html><head></head><body>")
        '=====
        'Get current model and check that it is a drawing
        '=====
            model = session.CurrentModel
            If model Is Nothing Then
                Throw New Exception("Model not present")
            End If
            If Not model.Type = EpfcModelType.EpfcMDL_DRAWING Then
                Throw New Exception("Model is not drawing")
            End If
            drawing = CType(model, IpfcDrawing)
            sheets = drawing.NumberOfSheets

            For i = 1 To sheets
                '=====
                'Get information about each sheet
                '=====
                    sheetData = drawing.GetSheetData(i)
                    sheetFormat = drawing.GetSheetFormat(i)

                    unit = "unknown"

                    Select Case sheetData.Units.GetType
                        Case EpfcLengthUnitType.EpfcLENGTHUNIT_CM
                            unit = "cm"
                        Case EpfcLengthUnitType.EpfcLENGTHUNIT_FOOT
                            unit = "feet"
                        Case EpfcLengthUnitType.EpfcLENGTHUNIT_INCH
                            unit = "inches"
                        Case EpfcLengthUnitType.EpfcLENGTHUNIT_M
                            unit = "m"
                        Case EpfcLengthUnitType.EpfcLENGTHUNIT_MCM
                            unit = "mcm"
                        Case EpfcLengthUnitType.EpfcLENGTHUNIT_MM
                            unit = "mm"
                    End Select
                End If
            Next i
        End Try
    End Sub
```

```

        End Select
' =====
'Store sheet information
' =====
        file.WriteLine("<h2>Sheet " + i.ToString + "</h2>")
        file.WriteLine("<table>")
        file.WriteLine(" <tr><td> Width </td><td> " +
            sheetData.Width.ToString + " </td></tr> ")
        file.WriteLine(" <tr><td> Height </td><td> " +
            sheetData.Height.ToString + " </td></tr> ")
        file.WriteLine(" <tr><td> Units </td><td> " + unit +
            " </td></tr> ")

        If (sheetFormat Is Nothing) Then
            formatName = "none"
        Else
            formatName = sheetFormat.FullName
        End If
        file.WriteLine(" <tr><td> Format </td><td> " + formatName
            + " </td></tr> ")

        file.WriteLine("</table>")
        file.WriteLine("<br>")
    Next

    file.WriteLine("</body></html>")
    file.Close()
    file = Nothing

    session.CurrentWindow.SetURL(fileName)

Catch ex As Exception
    MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
    If Not file Is Nothing Then
        file.Close()
    End If
End Try

End Sub

```

## Drawing Views

A drawing view is represented by the interface **IpfcView2D**. All model views in the drawing are associative, that is, if you change a dimensional value in one view, the system updates other drawing views accordingly. The model automatically reflects any dimensional changes that you make to a drawing. In addition, corresponding drawings also reflect any changes that you make to a model such as the addition or deletion of features and dimensional changes.

## Creating Drawing Views

Method Introduced:

- **IpfcModel2D.CreateView()**

The method **IpfcModel2D.CreateView()** creates a new view in the drawing. Before calling this method, the drawing must be displayed in a window.

The interface **IpfcView2DCreateInstructions** contains details on how to create the view. The types of drawing views supported for creation are:

- EpfcDRAWVIEW GENERAL--General drawing views
- EpfcDRAWVIEW PROJECTION--Projected drawing views

## General Drawing Views

The interface **IpfcGeneralViewCreateInstructions** contains details on how to create general drawing views.

Methods and Properties Introduced:

- **CCpfcGeneralViewCreateInstructions.Create()**
- **IpfcGeneralViewCreateInstructions.ViewModel**
- **IpfcGeneralViewCreateInstructions.Location**
- **IpfcGeneralViewCreateInstructions.SheetNumber**
- **IpfcGeneralViewCreateInstructions.Orientation**
- **IpfcGeneralViewCreateInstructions.Exploded**
- **IpfcGeneralViewCreateInstructions.Scale**

The method **CCpfcGeneralViewCreateInstructions.Create()** creates the **IpfcGeneralViewCreateInstructions** data object used for creating general drawing views.

Use the property **IpfcGeneralViewCreateInstructions.ViewModel** to assign the solid model to display in the created general drawing view.

Use the property **IpfcGeneralViewCreateInstructions.Location** to assign the location in a drawing sheet to place the created general drawing view.

Use the property **IpfcGeneralViewCreateInstructions.SheetNumber** to set the number of the drawing sheet in which the general drawing view is created.

The property **IpfcGeneralViewCreateInstructions.Orientation** assigns the orientation of the model in the general drawing view in the form of the **IpfcTransform3D** data object. The transformation matrix must only consist of the rotation to be applied to the model. It must not consist of any displacement or scale components. If necessary, set the displacement to {0, 0, 0} using the method **IpfcTransform3D.SetOrigin()**, and remove any scaling factor by normalizing the matrix.

Use the property **IpfcGeneralViewCreateInstructions.Exploded** to set the created general drawing view to be an exploded view.

Use the property **IpfcGeneralViewCreateInstructions.Scale** to assign a scale to the created general drawing view. This value is optional, if not assigned, the default drawing scale is used.

## Projected Drawing Views

The interface **IpfcProjectionViewCreateInstructions** contains details on how to create general drawing views.

Methods and Properties Introduced:

- **CCpfcProjectionViewCreateInstructions.Create()**

- **IpfcProjectionViewCreateInstructions.ParentView**
- **IpfcProjectionViewCreateInstructions.Location**
- **IpfcProjectionViewCreateInstructions.Exploded**

The method **CCpfcProjectionViewCreateInstructions.Create()** creates the **IpfcProjectionViewCreateInstructions** data object used for creating projected drawing views.

Use the property **IpfcProjectionViewCreateInstructions.ParentView** to assign the parent view for the projected drawing view.

Use the property **IpfcProjectionViewCreateInstructions.Location** to assign the location of the projected drawing view. This location determines how the drawing view will be oriented.

Use the property **IpfcProjectionViewCreateInstructions.Exploded** to set the created projected drawing view to be an exploded view.

#### **Example: Creating Drawing Views**

The following example code adds a new sheet to a drawing and creates three views of a selected model.

```
Public Sub createSheetAndViews(ByRef session As IpfcBaseSession, ByVal
                               solidName As String)

    Dim model As IpfcModel
    Dim solidModel As IpfcModel
    Dim drawing As IpfcDrawing
    Dim sheetNo As Integer
    Dim modelDesc As IpfcModelDescriptor
    Dim matrix As CpfcMatrix3D
    Dim i, j As Integer
    Dim transF As IpfcTransform3D
    Dim pointLoc As IpfcPoint3D
    Dim genViewInstructions As IpfcGeneralViewCreateInstructions
    Dim proViewInstructions As IpfcProjectionViewCreateInstructions
    Dim view2D As IpfcView2D
    Dim outline As CpfcOutline3D

    Try
'=====
'Get current model and check that it is a drawing
'=====
        model = session.CurrentModel
        If model Is Nothing Then
            Throw New Exception("Model not present")
        End If
        If Not model.Type = EpfcModelType.EpfcMDL_DRAWING Then
            Throw New Exception("Model is not drawing")
        End If
        drawing = CType(model, IpfcDrawing)
'=====
'Add new sheet to drawing
'=====
        sheetNo = drawing.AddSheet()
        drawing.CurrentSheetNumber = sheetNo
    End Try
End Sub
```

```

' =====
'Find the model in session or retrieve from disk
' =====
    modelDesc = (New CCpfcModelDescriptor).CreateFromFileName(solidName)
    solidModel = session.GetModelFromDescr(modelDesc)

    If solidModel Is Nothing Then
        solidModel = session.RetrieveModel(modelDesc)
        If solidModel Is Nothing Then
            Throw New Exception("Unable to load Model " + solidName)
        End If
    End If

' =====
'Add the model to drawing
' =====
    Try
        drawing.AddModel(solidModel)
    Catch ex As Exception
        Throw New Exception("Unable to add Model " + solidName + "
                               to drawing")
    End Try

' =====
'Create a general view from the Z axis direction at a predefined
'Location
' =====
    matrix = New CpfcMatrix3D
    For i = 0 To 3
        For j = 0 To 3
            If i = j Then
                matrix.Set(i, j, 1.0)
            Else
                matrix.Set(i, j, 0.0)
            End If
        Next
    Next
    transF = (New CCpfcTransform3D).Create(matrix)

    pointLoc = New CpfcPoint3D
    pointLoc.Set(0, 200.0)
    pointLoc.Set(1, 600.0)
    pointLoc.Set(2, 0.0)

    genViewInstructions = (New CCpfcGeneralViewCreateInstructions). _Create
(solidModel, sheetNo,
                                                pointLoc, transF)

    view2D = drawing.CreateView(genViewInstructions)
' =====
'Get the position and size of the new view
' =====
    outline = view2D.Outline
' =====
'Create a projected view to the right of the general view
' =====
    pointLoc.Set(0, outline.Item(1).Item(0) + (outline.Item(1).Item(0) _ -
outline.Item(0).Item(0)))
    pointLoc.Set(1, (outline.Item(0).Item(1) + outline.Item(1).Item(1)) / 2)

    proViewInstructions = (New CCpfcProjectionViewCreateInstructions).

```

```

_Create(view2D, pointLoc)
    drawing.CreateView(proViewInstructions)
'=====
'Create a projected view bellow the general view
'=====
    pointLoc.Set(0, (outline.Item(0).Item(0) + outline.Item(1).Item(0)) / 2)
    pointLoc.Set(1, outline.Item(0).Item(1) - (outline.Item(1).Item(1) -
outline.Item(0).Item(1)))

    proViewInstructions = (New CCpfcProjectionViewCreateInstructions).
_Create(view2D, pointLoc)
    drawing.CreateView(proViewInstructions)

    drawing.Regenerate()

Catch ex As Exception
    MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
End Try

End Sub

```

## Obtaining Drawing Views

Methods and Property Introduced:

- **IpfcSelection.SelView2D**
- **IpfcModel2D.List2DViews()**
- **IpfcModel2D.GetViewByName()**
- **IpfcModel2D.GetViewDisplaying()**
- **IpfcSheetOwner.GetSheetBackgroundView()**

The property **IpfcSelection.SelView2D** returns the selected drawing view (if the user selected an item from a drawing view). It returns a null value if the selection does not contain a drawing view.

The method **IpfcModel2D.List2DViews()** lists and returns the drawing views found. This method does not include the drawing sheet background views returned by the method **IpfcSheetOwner.GetSheetBackgroundView()**.

The method **IpfcModel2D.GetViewByName()** returns the drawing view based on the name. This method returns a null value if the specified view does not exist.

The method **IpfcModel2D.GetViewDisplaying()** returns the drawing view that displays a dimension. This method returns a null value if the dimension is not displayed in the drawing.

### Note:

This method works for solid and drawing dimensions.

The method **IpfcSheetOwner.GetSheetBackgroundView()** returns the drawing sheet background views.

## Drawing View Information

Methods and Properties Introduced:

- **IpfcChild.DBParent**
- **IpfcView2D.GetSheetNumber()**
- **IpfcView2D.IsBackground**
- **IpfcView2D.GetModel()**
- **IpfcView2D.Scale**
- **IpfcView2D.GetIsScaleUserdefined()**
- **IpfcView2D.Outline**
- **IpfcView2D.GetLayerDisplayStatus()**
- **IpfcView2D.IsViewdisplayLayerDependent**
- **IpfcView2D.Display**
- **IpfcView2D.GetTransform()**
- **IpfcView2D.Name**

The inherited property **IpfcChild.DBParent**, when called on a **IpfcView2D** object, provides the drawing model which owns the specified drawing view. The return value of the method can be downcast to a **IpfcModel2D** object.

The method **IpfcView2D.GetSheetNumber()** returns the sheet number of the sheet that contains the drawing view.

The property **IpfcView2D.IsBackground** returns a value that indicates whether the view is a background view or a model view.

The method **IpfcView2D.GetModel()** returns the solid model displayed in the drawing view.

The property **IpfcView2D.Scale** returns the scale of the drawing view.

The method **IpfcView2D.GetIsScaleUserdefined()** specifies if the drawing has a user-defined scale.

The property **IpfcView2D.Outline** returns the position of the view in the sheet in world units.

The method **IpfcView2D.GetLayerDisplayStatus()** returns the display status of the specified layer in the drawing view.

The property **IpfcView2D.Display** returns an output structure that describes the display settings of the drawing view. The fields in the structure are as follows:

- **Style**--Whether to display as wireframe, hidden lines, no hidden lines, or shaded
- **TangentStyle**--Linestyle used for tangent edges
- **CableStyle**--Linestyle used to display cables
- **RemoveQuiltHiddenLines**--Whether or not to apply hidden-line-removal to quilts
- **ShowConceptModel**--Whether or not to display the skeleton
- **ShowWeldXSection**--Whether or not to include welds in the cross-section

The method **IpfcView2D.GetTransform()** returns a matrix that describes the transform between 3D solid coordinates and 2D world units for that drawing view. The transformation matrix is a combination of the following factors:

- o The location of the view origin with respect to the drawing origin.
- o The scale of the view units with respect to the drawing units
- o The rotation of the model with respect to the drawing coordinate system.

The property **IpfcView2D.Name** returns the name of the specified view in the drawing.

**Example: Listing the Views in a Drawing**

The following example creates an information window about all the views in a drawing. The information is placed in an external browser window

```
Public Sub listViews(ByRef session As IpfcBaseSession, ByVal fileName
                    As String)
    Dim file As StreamWriter = Nothing
    Dim model As IpfcModel
    Dim drawing As IpfcDrawing
    Dim view2Ds As IpfcView2Ds
    Dim i As Integer
    Dim view2D As IpfcView2D
    Dim viewName As String
    Dim sheetNo As Integer
    Dim solid As IpfcModel
    Dim solidDesc As IpfcModelDescriptor
    Dim outline As CpfcOutline3D
    Dim scale As Double
    Dim viewDisplay As IpfcViewDisplay
    Dim displayStyle As String

    Try
' =====
' Create file to store information to be displayed
' =====
        file = New StreamWriter(fileName)
        file.WriteLine("<html><head></head><body>")
' =====
' Get current model and check that it is a drawing
' =====
        model = session.CurrentModel
        If model Is Nothing Then
            Throw New Exception("Model not present")
        End If
        If Not model.Type = EpfcModelType.EpfcMDL_DRAWING Then
            Throw New Exception("Model is not drawing")
        End If
        drawing = CType(model, IpfcDrawing)

        view2Ds = drawing.List2DViews
        For i = 0 To view2Ds.Count - 1
' =====
' Get information about each view such as name, model
' =====
            view2D = view2Ds.Item(i)

            viewName = view2D.Name
            sheetNo = view2D.GetSheetNumber

```

```

    solid = view2D.GetModel
    solidDesc = solid.Descr

    outline = view2D.Outline
    scale = view2D.Scale
    viewDisplay = view2D.Display
    displayStyle = "unknown"

    Select Case viewDisplay.Style
        Case EpfcDisplayStyle.EpfcDISPSTYLE_DEFAULT
            displayStyle = "default"
        Case EpfcDisplayStyle.EpfcDISPSTYLE_HIDDEN_LINE
            displayStyle = "hidden line"
        Case EpfcDisplayStyle.EpfcDISPSTYLE_NO_HIDDEN
            displayStyle = "no hidden"
        Case EpfcDisplayStyle.EpfcDISPSTYLE_SHADED
            displayStyle = "shaded"
        Case EpfcDisplayStyle.EpfcDISPSTYLE_WIREFRAME
            displayStyle = "wireframe"
    End Select

'=====
'Store the view information
'=====

    file.WriteLine("<h2>View " + viewName + "</h2>")
    file.WriteLine("<table>")
    file.WriteLine(" <tr><td> Sheet </td><td> " + sheetNo.ToString + " </
td></tr> ")
    file.WriteLine(" <tr><td> Model </td><td> " + solidDesc.GetFullName
+ " </td></tr> ")
    file.WriteLine(" <tr><td> Outline </td><td> ")
    file.WriteLine("<table><tr><td> <i>Lower left:</i> </td><td>")
    file.WriteLine(outline.Item(0).Item(0).ToString + ", " +
        _outline.Item(0).Item(1).ToString + ", " +
        _outline.Item(0).Item(2).ToString)
    file.WriteLine("</td></tr><tr><td> <i>Upper right:</i></td><td>")
    file.WriteLine(outline.Item(1).Item(0).ToString + ", " +
        _outline.Item(1).Item(1).ToString + ", " +
        _outline.Item(1).Item(2).ToString)
    file.WriteLine("</td></tr></table></td>")
    file.WriteLine(" <tr><td> Scale </td><td> " + scale.ToString + " </
td></tr> ")
    file.WriteLine(" <tr><td> Display Style </td><td> " + displayStyle +
" </td></tr>")
    file.WriteLine("</table>")
    file.WriteLine("<br>")

    Next

    file.WriteLine("</body></html>")
    file.Close()
    file = Nothing

    session.CurrentWindow.SetURL(fileName)

Catch ex As Exception
    MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
Finally

```

```
        If Not file Is Nothing Then
            file.Close()
        End If
    End Try
```

```
End Sub
```

## Drawing Views Operations

Methods Introduced:

- **IpfcView2D.Translate()**
- **IpfcView2D.Delete()**
- **IpfcView2D.Regenerate()**
- **IpfcView2D.SetLayerDisplayStatus()**

The method **IpfcView2D.Translate()** moves the drawing view by the specified transformation vector.

The method **IpfcView2D.Delete()** deletes a specified drawing view. Set the *DeleteChildren* parameter to true to delete the children of the view. Set this parameter to false or null to prevent deletion of the view if it has children.

The method **IpfcView2D.Regenerate()** erases the displayed view of the current object, regenerates the view from the current drawing, and redisplay the view.

The method **IpfcView2D.SetLayerDisplayStatus()** sets the display status for the layer in the drawing view.

## Drawing Dimensions

This section describes the VB API methods that give access to the types of dimensions that can be created in the drawing mode. They do not apply to dimensions created in the solid mode, either those created automatically as a result of feature creation, or reference dimension created in a solid. A drawing dimension or a reference dimension shown in a drawing is represented by the interface **IpfcDimension2D**.

## Obtaining Drawing Dimensions

Methods and Property Introduced:

- **IpfcModelItemOwner.ListItems()**
- **IpfcModelItemOwner.GetItemById()**
- **IpfcSelection.SellItem**

The method **IpfcModelItemOwner.ListItems()** returns a list of drawing dimensions specified by the parameter *Type* or returns null if no drawing dimensions of the specified type are found. This method lists only those dimensions created in the drawing.

The values of the parameter *Type* for the drawing dimensions are:

- ITEM\_DIMENSION--Dimension
- ITEM\_REF\_DIMENSION--Reference dimension

Set the parameter *Type* to the type of drawing dimension to retrieve. If this parameter is set to null, then all the dimensions in the drawing are listed.

The method **IpfcModelItemOwner.GetItemById()** returns a drawing dimension based on the type and the integer identifier. The method returns only those dimensions created in the drawing. It returns a null if a drawing dimension with the specified attributes is not found.

The property **IpfcSelection.SelectedItem** returns the value of the selected drawing dimension.

## Creating Drawing Dimensions

Methods Introduced:

- **CCpfcDrawingDimCreateInstructions.Create()**
- **IpfcModel2D.CreateDrawingDimension()**
- **CCpfcEmptyDimensionSense.Create()**
- **CCpfcPointDimensionSense.Create()**
- **CCpfcSplinePointDimensionSense.Create()**
- **CCpfcTangentIndexDimensionSense.Create()**
- **CCpfcLinAOCTangentDimensionSense.Create()**
- **CCpfcAngleDimensionSense.Create()**
- **CCpfcPointToAngleDimensionSense.Create()**

The method **CCpfcDrawingDimCreateInstructions.Create()** creates an instructions object that describes how to create a drawing dimension using the method **IpfcModel2D.CreateDrawingDimension()**.

The parameters of the instruction object are:

- Attachments--The entities that the dimension is attached to. The selections should include the drawing model view.
- IsRefDimension--True if the dimension is a reference dimension, otherwise null or false.
- OrientationHint--Describes the orientation of the dimensions in cases where this cannot be deduced from the attachments themselves.
- Senses--Gives more information about how the dimension attaches to the entity, i.e., to what part of the entity and in what direction the dimension runs. The types of dimension senses are as follows:
  - EpcfDIMSENSE\_NONE
  - EpcfDIMSENSE\_POINT
  - EpcfDIMSENSE\_SPLINE\_PT
  - EpcfDIMSENSE\_TANGENT\_INDEX
  - EpcfDIMSENSE\_LINEAR\_TO\_ARC\_OR\_CIRCLE\_TANGENT
  - EpcfDIMSENSE\_ANGLE
  - EpcfDIMSENSE\_POINT\_TO\_ANGLE
- TextLocation--The location of the dimension text, in world units.

The method **IpfcModel2D.CreateDrawingDimension()** creates a dimension in the drawing based on the instructions data object that contains information needed to place the dimension. It takes as input an array of pfcSelection objects and an array of pfcDimensionSense structures that describe the required attachments. The method returns the created drawing dimension.

The method **CCpfcEmptyDimensionSense.Create()** creates a new dimension sense associated with the type DIMSENSE NONE. The "sense" field is set to *Type*. In this case no information such as location or direction 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 straight line. If the attachments are two parallel lines, the dimension is the distance between them.

The method **CCpfcPointDimensionSense.Create()** creates a new dimension sense associated with the type DIMSENSE POINT which specifies the part of the entity to which the dimension is attached. The "sense" field is set to the value of the parameter *PointType*.

The possible values of *PointType* are:

- EpcfDIMPOINT\_END1-- The first end of the entity
- EpcfDIMPOINT\_END2--The second end of the entity
- EpcfDIMPOINT\_CENTER--The center of an arc or circle
- EpcfDIMPOINT\_NONE--No information such as location or direction of the attachment is specified. This is similar to setting the PointType to DIMSENSE NONE.
- EpcfDIMPOINT\_MIDPOINT--The mid point of the entity

The method **CCpfcSplinePointDimensionSense.Create()** creates a dimension sense associated with the type DIMSENSE\_SPLINE\_PT. This means that the attachment is to a point on a spline. The "sense" field is set to *SplinePointIndex* i.e., the index of the spline point.

The method **CCpfcTangentIndexDimensionSense.Create()** creates a new dimension sense associated with the type DIMSENSE\_TANGENT\_INDEX. The attachment is to a tangent of the entity, which is an arc or a circle. The sense field is set to *TangentIndex*, i.e., the index of the tangent of the entity.

The method **CCpfcLinAOCTangentDimensionSense.Create()** creates a new dimension sense associated with the type *DIMSENSE\_LINEAR\_TO\_ARC\_OR\_CIRCLE\_TANGENT*. The dimension is the perpendicular distance between the a line and a tangent to an arc or a circle that is parallel to the line. The sense field is set to the value of the parameter *TangentType*.

The possible values of *TangentType* are:

- EpcfDIMLINAOCTANGENT\_LEFT0--The tangent is to the left of the line, and is on the same side, of the center of the arc or circle, as the line.
- EpcfDIMLINAOCTANGENT\_RIGHT0--The tangent is to the right of the line, and is on the same side, of the center of the arc or circle, as the line.
- EpcfDIMLINAOCTANGENT\_LEFT1--The tangent is to the left of the line, and is on the opposite side of the line.
- EpcfDIMLINAOCTANGENT\_RIGHT1-- The tangent is to the right of the line, and is on the opposite side of the line.

The method **CCpfcAngleDimensionSense.Create()** creates a new dimension sense associated with the type *DIMSENSE\_ANGLE*. The dimension is the angle between two straight entities. The "sense" field is set to the value of the parameter *AngleOptions*.

The possible values of *AngleOptions* are:

- IsFirst--Is set to TRUE if the angle dimension starts from the specified entity in a counterclockwise direction. Is set to FALSE if the dimension ends at the specified entity. The value is TRUE for one entity and FALSE for the other entity forming the angle.
- ShouldFlip--If the value of ShouldFlip is FALSE, and the direction of the specified entity is away from the vertex of the angle, then the dimension attaches directly to the entity. If the direction of the entity is away from the vertex of the angle, then the dimension is attached to the a witness line. The witness line is in line with the entity but in the direction opposite to the vertex of the angle. If the value of ShouldFlip is TRUE then the above cases are reversed.

The method **CCpfcPointToAngleDimensionSense.Create()** creates a new dimension sense associated with the type *DIMSENSE\_POINT\_TO\_ANGLE*. The dimension is the angle between a line entity and the tangent to a curved entity.

The curve attachment is of the type *DIMSENSE\_POINT\_TO\_ANGLE* and the line attachment is of the type *DIMSENSE\_POINT*. In this case both the "angle" and the "angle\_sense" fields must be set. The field "sense" shows which end of the curve the dimension is attached to and the field "angle\_sense" shows the direction in which the dimension rotates and to which side of the tangent it attaches.

## Drawing Dimensions Information

Methods and Properties Introduced:

- **IpfcDimension2D.IsAssociative**
- **IpfcDimension2D.GetIsReference()**
- **IpfcDimension2D.IsDisplayed**
- **IpfcDimension2D.GetAttachmentPoints()**
- **IpfcDimension2D.GetDimensionSenses()**
- **IpfcDimension2D.GetOrientationHint()**
- **IpfcDimension2D.GetBaselineDimension()**
- **IpfcDimension2D.Location**
- **IpfcDimension2D.GetView()**
- **IpfcDimension2D.GetTolerance()**
- **IpfcDimension2D.IsToleranceDisplayed**

The property **IpfcDimension2D.IsAssociative** returns whether the dimension or reference dimension in a drawing is associative.

The method **IpfcDimension2D.GetIsReference()** determines whether the drawing dimension is a reference dimension.

The method **IpfcDimension2D.IsDisplayed** determines whether the dimension will be displayed in the drawing.

The method **IpfcDimension2D.GetAttachmentPoints()** returns a sequence of attachment points. The dimension senses array returned by the method **IpfcDimension2D.GetDimensionSenses()** gives more information on how these attachments are interpreted.

The method **IpfcDimension2D.GetDimensionSenses()** returns a sequence of dimension senses, describing how the dimension is attached to each attachment returned by the method **IpfcDimension2D.GetAttachmentPoints()**.

The method **IpfcDimension2D.GetOrientationHint()** returns the orientation hint for placing the drawing dimensions. The orientation hint determines how Pro/ENGINEER will orient the dimension with respect to the attachment points.

### Note:

This methods described above are applicable only for dimensions created in the drawing mode. It does not support dimensions created at intersection points of entities.

The method **IpfcDimension2D.GetBaselineDimension()** returns an ordinate baseline drawing dimension. It returns a null value if the dimension is not an ordinate dimension.

**Note:**

The method updates the display of the dimension only if it is currently displayed.

The property **IpfcDimension2D.Location** returns the placement location of the dimension.

The method **IpfcDimension2D.GetView()** returns the drawing view in which the dimension is displayed. This method applies to dimensions stored in the solid or in the drawing.

The method **IpfcDimension2D.GetTolerance()** retrieves the upper and lower tolerance limits of the drawing dimension in the form of the **IpfcDimTolerance** object. A null value indicates a nominal tolerance.

Use the method **IpfcDimension2D.IsToleranceDisplayed** determines whether or not the dimension's tolerance is displayed in the drawing.

## Drawing Dimensions Operations

Methods Introduced:

- **IpfcDimension2D.ConvertToLinear()**
- **IpfcDimension2D.ConvertToOrdinate()**
- **IpfcDimension2D.ConvertToBaseline()**
- **IpfcDimension2D.SwitchView()**
- **IpfcDimension2D.SetTolerance()**
- **IpfcDimension2D.EraseFromModel2D()**
- **IpfcModel2D.SetViewDisplaying()**

The method **IpfcDimension2D.ConvertToLinear()** converts an ordinate drawing dimension to a linear drawing dimension. The drawing containing the dimension must be displayed.

The method **IpfcDimension2D.ConvertToOrdinate()** converts a linear drawing dimension to an ordinate baseline dimension.

The method **IpfcDimension2D.ConvertToBaseline()** converts a location on a linear drawing dimension to an ordinate baseline dimension. The method returns the newly created baseline dimension.

**Note:**

The method updates the display of the dimension only if it is currently displayed.

The method **IpfcDimension2D.SwitchView()** changes the view where a dimension created in the drawing is displayed.

The method **IpfcDimension2D.SetTolerance()** assigns the upper and lower tolerance limits of the drawing dimension.

The method **IpfcDimension2D.EraseFromModel2D()** permanently erases the dimension from the drawing.

The method **IpfcModel2D.SetViewDisplaying()** changes the view where a dimension created in a solid model is displayed.

**Example: Command Creation of Dimensions from Model Datum Points**

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.

```
'=====
'Function      :   createPointDims
'Purpose       :   This function creates vertical and horizontal ordinate
'               :   dimensions from each datum point in a
'               :   drawing view to a selected coordinate system datum.
'=====
Public Sub createPointDims(ByRef session As IpfcBaseSession)

    Dim hBaseLine As IpfcDimension2D = Nothing
    Dim vBaseLine As IpfcDimension2D = Nothing
    Dim selectionOptions As IpfcSelectionOptions
    Dim selections As CpfcSelections
    Dim csysSelection As IpfcSelection
    Dim selItem As IpfcModelItem
    Dim selPath As IpfcComponentPath
    Dim selView As IpfcView2D
    Dim selPosition As CpfcPoint3D
    Dim drawing As IpfcModel2D
    Dim rootSolid As IpfcSolid
    Dim asmTransform As IpfcTransform3D
    Dim points As IpfcModelItems
    Dim csysPosition As CpfcPoint3D
    Dim viewTransform As IpfcTransform3D
    Dim csys3DPosition As CpfcVector2D
    Dim outline As IpfcOutline3D
    Dim senses As CpfcDimensionSenses
    Dim attachments As CpfcSelections
    Dim p As Integer
    Dim point As IpfcPoint
    Dim pointPosition As CpfcPoint3D
    Dim sense1 As IpfcPointDimensionSense
    Dim sense2 As IpfcPointDimensionSense
    Dim pointSelection As IpfcSelection
    Dim dimPosition As CpfcVector2D
    Dim createInstructions As IpfcDrawingDimCreateInstructions
    Dim showInstructions As IpfcDrawingDimensionShowInstructions
    Dim dimension As IpfcDimension2D

    Try

'=====
        'Select a coordinate system. This defines the model (the top one
        'in that view), and the common attachments for the dimensions
'=====
        selectionOptions = (New CCpfcSelectionOptions).Create("csys")
        selectionOptions.MaxNumSels = 1
        selections = session.Select(selectionOptions, Nothing)
        If (selections Is Nothing) Or (selections.Count) = 0 Then
            Throw New Exception("Nothing Selected")
        End If

'=====
        'Extract the csys handle, and assembly path, and view handle
'=====
        csysSelection = selections.Item(0)
```

```

selItem = csysSelection.SelItem
selPath = csysSelection.Path
selView = csysSelection.SelView2D
selPosition = csysSelection.Point

If selView Is Nothing Then
    Throw New Exception("Must select coordinate system from a
                        drawing view.")
End If

'=====
'Get the root solid, and the transform from the root to the
'component owning the csys
'=====
asmTransform = Nothing
drawing = selView.DBParent
rootSolid = selItem.DBParent
If Not selPath Is Nothing Then
    rootSolid = selPath.Root
    asmTransform = selPath.GetTransform(True)
End If

'=====
'Get a list of datum points in the model
'=====
points = rootSolid.ListItems(EpfcModelItemType.EpfcITEM_POINT)
If (points Is Nothing) Or (points.Count = 0) Then
    Throw New Exception("Nothing Selected")
End If

'=====
'Calculate where the csys is located on the drawing
'=====
csysPosition = selPosition
If Not asmTransform Is Nothing Then
    csysPosition = asmTransform.TransformPoint(selPosition)
End If
viewTransform = selView.GetTransform
csysPosition = viewTransform.TransformPoint(csysPosition)

csys3DPosition = New CpfcVector2D
csys3DPosition.Set(0, csysPosition.Item(0))
csys3DPosition.Set(1, csysPosition.Item(1))

'=====
'Get the view outline
'=====
outline = selView.Outline

'=====
'Allocate the attachment arrays
'=====
senses = New CpfcDimensionSenses
attachments = New CpfcSelections

'=====
'Loop through all the datum points
'=====
For p = 0 To points.Count - 1
'=====

```

```

'Calculate the position of the point on the drawing
'=====
point = points.Item(p)
pointPosition = point.Point
pointPosition = viewTransform.TransformPoint(pointPosition)

'=====
'Set up the "sense" information
'=====
sense1 = (New CCpfcPointDimensionSense). _
        Create(EpfcDimensionPointType.EpfcDIMPOINT_CENTER)
senses.Set(0, sense1)
sense2 = (New CCpfcPointDimensionSense). _
        Create(EpfcDimensionPointType.EpfcDIMPOINT_CENTER)
senses.Set(1, sense2)

'=====
'Set the attachment information
'=====
pointSelection = (New
        CMpfcSelect).CreateModelItemSelection(point, Nothing)
pointSelection.SelView2D = selView
attachments.Set(0, pointSelection)
attachments.Set(1, csysSelection)

'=====
'Calculate the dim position to be just to the left of the
'drawing view, midway between the point and csys
'=====
dimPosition = New CpfcVector2D
dimPosition.Set(0, outline.Item(0).Item(0) - 20.0)
dimPosition.Set(1, (csysPosition.Item(1) +
        pointPosition.Item(1)) / 2)

'=====
'Create and display the dimension
'=====
createInstructions = (New
        CCpfcDrawingDimCreateInstructions).Create _
        (attachments, _
        senses, _
        dimPosition, _
        EpfcOrientationHint.EpfcORIENTHINT_VERTICAL)

dimension = drawing.CreateDrawingDimension(createInstructions)
showInstructions = (New CCpfcDrawingDimensionShowInstructions).
Create _
        (selView, Nothing)

CType(dimension, IpfcBaseDimension).Show(showInstructions)

'=====
'If this is the first vertical dim, create an ordinate base
'line from it, else just convert it to ordinate
'=====
If (p = 0) Then
    vBaseLine = dimension.ConvertToBaseline(csys3DPosition)
Else

```

```

        dimension.ConvertToOrdinate(vBaseLine)
    End If

'=====
    'Set this dimension to be horizontal
'=====

        createInstructions.OrientationHint = EpfcOrientationHint.
EpfcORIENTHINT_HORIZONTAL

'=====
    'Calculate the dim position to be just to the bottom of the
    'drawing view, midway between the point and csys
'=====
    dimPosition.Set(0, (csysPosition.Item(0) + pointPosition.Item(0)) /
2)
    dimPosition.Set(1, outline.Item(1).Item(1) - 20.0)
    createInstructions.TextLocation = dimPosition

'=====
    'Create and display the dimension
'=====
    dimension = drawing.CreateDrawingDimension(createInstructions)
    'dimension.Show(showInstructions)
    CType(dimension, IpfcBaseDimension).Show(showInstructions)

'=====
    'If this is the first horizontal dim, create an ordinate base
    'line from it, else just convert it to ordinate
'=====
    If (p = 0) Then
        hBaseLine = dimension.ConvertToBaseline(csys3DPosition)
    Else
        dimension.ConvertToOrdinate(hBaseLine)
    End If
Next

    Catch ex As Exception
        MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
    End Try

End Sub

```

## Drawing Tables

A drawing table in the VB API is represented by the interface **IpfcTable**. It is a child of the **IpfcModelItem** interface.

Some drawing table methods operate on specific rows or columns. The row and column numbers in the VB API begin with 1 and range up to the total number of rows or columns in the table. Some drawing table methods operate on specific table cells. The interface **IpfcTableCell** is used to represent a drawing table cell.

## Creating Drawing Cells

Method Introduced:

- **CCpfcTableCell.Create()**

The method **CCpfcTableCell.Create()** creates the **IpfcTableCell** object representing a cell in the drawing table.

Some drawing table methods operate on specific drawing segment. A multisegmented drawing table contains 2 or more areas in the drawing. Inserting or deleting rows in one segment of the table can affect the contents of other segments. Table segments are numbered beginning with 0. If the table has only a single segment, use 0 as the segment id in the relevant methods.

## Selecting Drawing Tables and Cells

Methods and Properties Introduced:

- **IpfcBaseSession.Select()**
- **IpfcSelection.SelItem**
- **IpfcSelection.SelTableCell**
- **IpfcSelection.SelTableSegment**

Tables may be selected using the method **IpfcBaseSession.Select()**. Pass the filter `dwg_table` to select an entire table and the filter `table_cell` to prompt the user to select a particular table cell.

The property **IpfcSelection.SelItem** returns the selected table handle. It is a model item that can be cast to a **IpfcTable** object.

The property **IpfcSelection.SelTableCell** returns the row and column indices of the selected table cell.

The property **IpfcSelection.SelTableSegment** returns the table segment identifier for the selected table cell. If the table consists of a single segment, this method returns the identifier 0.

## Creating Drawing Tables

Methods Introduced:

- **CCpfcTableCreateInstructions.Create()**
- **IpfcTableOwner.CreateTable()**

The method **CCpfcTableCreateInstructions.Create()** creates the **IpfcTableCreateInstructions** data object that describes how to construct a new table using the method **IpfcTableOwner.CreateTable()**.

The parameters of the instructions data object are:

- **Origin**--This parameter stores a three dimensional point specifying the location of the table origin. The origin is the position of the top left corner of the table.
- **RowHeights**--Specifies the height of each row of the table.
- **ColumnData**--Specifies the width of each column of the table and its justification.
- **SizeTypes**--Indicates the scale used to measure the column width and row height of the table.

The method **IpfcTableOwner.CreateTable()** creates a table in the drawing specified by the **IpfcTableCreateInstructions** data object.

## Retrieving Drawing Tables

## Methods Introduced

- **CCpfcTableRetrieveInstructions.Create()**
- **IpfcTableOwner.RetrieveTable()**

The method **CCpfcTableRetrieveInstructions.Create()** creates the **IpfcTableRetrieveInstructions** data object that describes how to retrieve a drawing table using the method **IpfcTableOwner.RetrieveTable()**. The method returns the created instructions data object.

The parameters of the instruction object are:

- **FileName**--Name of the file containing the drawing table.
- **Position**--The location of left top corner of the retrieved table.

The method **IpfcTableOwner.RetrieveTable()** retrieves a table specified by the **IpfcTableRetrieveInstructions** data object from a file on the disk. It returns the retrieved table. The data object contains information on the table to retrieve and is returned by the method **CCpfcTableRetrieveInstructions.Create()**.

## Drawing Tables Information

### Methods Introduced:

- **IpfcTableOwner.ListTables()**
- **IpfcTableOwner.GetTable()**
- **IpfcTable.GetRowCount()**
- **IpfcTable.GetColumnCount()**
- **IpfcTable.CheckIfIsFromFormat()**
- **IpfcTable.GetRowSize()**
- **IpfcTable.GetColumnSize()**
- **IpfcTable.GetText()**
- **IpfcTable.GetCellNote()**

The method **IpfcTableOwner.ListTables()** returns a sequence of tables found in the model.

The method **IpfcTableOwner.GetTable()** returns a table specified by the table identifier in the model. It returns a null value if the table is not found.

The method **IpfcTable.GetRowCount()** returns the number of rows in the table.

The method **IpfcTable.GetColumnCount()** returns the number of columns in the table.

The method **IpfcTable.CheckIfIsFromFormat()** verifies if the drawing table was created using the format of the drawing sheet specified by the sheet number. The method returns a true value if the table was created by applying the drawing format.

The method **IpfcTable.GetRowSize()** returns the height of the drawing table row specified by the segment identifier and the row number.

The method **IpfcTable.GetColumnSize()** returns the width of the drawing table column specified by the segment identifier and the column number.

The method **IpfcTable.GetText()** returns the sequence of text in a drawing table cell. Set the value of the parameter *Mode* to DWGTABLE\_NORMAL to get the text as displayed on the screen. Set it to DWGTABLE\_FULL to get symbolic text, which includes the names of parameter references in the table text.

The method **IpfcTable.GetCellNote()** returns the detail note item contained in the table cell.

## Drawing Tables Operations

Methods Introduced:

- **IpfcTable.Erase()**
- **IpfcTable.Display()**
- **IpfcTable.RotateClockwise()**
- **IpfcTable.InsertRow()**
- **IpfcTable.InsertColumn()**
- **IpfcTable.MergeRegion()**
- **IpfcTable.SubdivideRegion()**
- **IpfcTable.DeleteRow()**
- **IpfcTable.DeleteColumn()**
- **IpfcTable.SetText()**
- **IpfcTableOwner.DeleteTable()**

The method **IpfcTable.Erase()** erases the specified table temporarily from the display. It still exists in the drawing. The erased table can be displayed again using the method **IpfcTable.Display()**. The table will also be redisplayed by a window repaint or a regeneration of the drawing. Use these methods to hide a table from the display while you are making multiple changes to the table.

The method **IpfcTable.RotateClockwise()** rotates a table clockwise by the specified amount of rotation.

The method **IpfcTable.InsertRow()** inserts a new row in the drawing table. Set the value of the parameter *RowHeight* to specify the height of the row. Set the value of the parameter *InsertAfterRow* to specify the row number after which the new row has to be inserted. Specify 0 to insert a new first row.

The method **IpfcTable.InsertColumn()** inserts a new column in the drawing table. Set the value of the parameter *ColumnWidth* to specify the width of the column. Set the value of the parameter *InsertAfterColumn* to specify the column number after which the new column has to be inserted. Specify 0 to insert a new first column.

The method **IpfcTable.MergeRegion()** merges table cells within a specified range of rows and columns to form a single cell. The range is a rectangular region specified by the table cell on the upper left of the region and the table cell

on the lower right of the region.

The method **IpfcTable.SubdivideRegion()** removes merges from a region of table cells that were previously merged. The region to remove merges is specified by the table cell on the upper left of the region and the table cell on the lower right of the region.

The methods **IpfcTable.DeleteRow()** and **IpfcTable.DeleteColumn()** delete any specified row or column from the table. The methods also remove the text from the affected cells.

The method **IpfcTable.SetText()** sets text in the table cell.

Use the method **IpfcTableOwner.DeleteTable()** to delete a specified drawing table from the model permanently. The deleted table cannot be displayed again.

**Note:**

Many of the above methods provide a parameter `Repaint`. If this is set to true the table will be repainted after the change. If set to false or null Pro/ENGINEER will delay the repaint, allowing you to perform several operations before showing changes on the screen.

**Example: Creation of a Table Listing Datum Points**

The following example creates a drawing table that lists the datum points in a model shown in a drawing view.

```
Public Sub createTableOfPoints(ByRef session As IpfcBaseSession)
    Dim widths(4) As Double
    Dim selectionOptions As IpfcSelectionOptions
    Dim selections As CpfcSelections
    Dim csysSelection As IpfcSelection
    Dim selItem As IpfcModelItem
    Dim selPath As IpfcComponentPath
    Dim selView As IpfcView2D
    Dim drawing As IpfcModel2D
    Dim csys As IpfcCoordSystem
    Dim csysTransform As IpfcTransform3D
    Dim csysName As String
    Dim rootSolid As IpfcSolid
    Dim asmTransform As IpfcTransform3D
    Dim points As IpfcModelItems
    Dim location As CpfcPoint3D
    Dim tableInstructions As IpfcTableCreateInstructions
    Dim columnInfo As CpfcColumnCreateOptions
    Dim column As IpfcColumnCreateOption
    Dim i As Integer
    Dim rowInfo As Crealseq
    Dim drawTable As IpfcTable
    Dim topLeft As IpfcTableCell
    Dim bottomRight As IpfcTableCell
    Dim p As Integer
    Dim geomPoint As IpfcPoint
    Dim trfPoint As IpfcPoint3D

    Try
        widths(0) = 8.0
        widths(1) = 10.0
        widths(2) = 10.0
        widths(3) = 10.0
```

```

' =====
'Select a coordinate system. This defines the model (the top one
'in that view), and the common attachments for the dimensions
' =====
        selectionOptions = (New CCpfcSelectionOptions).Create("csys")
        selectionOptions.MaxNumSels = 1
        selections = session.Select(selectionOptions, Nothing)
        If (selections Is Nothing) Or (selections.Count) = 0 Then
            Throw New Exception("Nothing Selected")
        End If
' =====
'Extract the csys handle, and assembly path, and view handle
' =====
        csysSelection = selections.Item(0)
        selItem = csysSelection.SelItem
        selPath = csysSelection.Path
        selView = csysSelection.SelView2D

        If selView Is Nothing Then
            Throw New Exception("Must select coordinate system from a drawing
view.")
        End If

        drawing = selView.DBParent
' =====
'Extract the csys location (property CoordSys from class CoordSystem)
' =====
        csys = CType(selItem, IpfcCoordSystem)
        csysTransform = csys.CoordSys
        csysTransform.Invert()
        csysName = selItem.GetName
' =====
'Get the root solid, and the transform from the root to the
'component owning the csys
' =====
        asmTransform = Nothing
        rootSolid = selItem.DBParent
        If Not selPath Is Nothing Then
            rootSolid = selPath.Root
            asmTransform = selPath.GetTransform(False)
        End If
' =====
'Get a list of datum points in the model
' =====
        points = rootSolid.ListItems(EpfcModelItemType.EpfcITEM_POINT)
        If (points Is Nothing) Or (points.Count = 0) Then
            Throw New Exception("Nothing Selected")
        End If
' =====
'Set table position
' =====
        location = New CpfcPoint3D
        location.Set(0, 500.0)
        location.Set(1, 500.0)
        location.Set(2, 0.0)
' =====
'Setup the table creation instructions
' =====
        tableInstructions = (New CCpfcTableCreateInstructions).Create(location)

```

```

tableInstructions.SizeType = EpfcTableSizeType.EpfcTABLESIZE_BY_NUM_CHARS

columnInfo = New CpfcColumnCreateOptions

For i = 0 To widths.Length - 1
    column = (New CCpfcColumnCreateOption).Create _
              (EpfcColumnJustification.EpfcCOL_JUSTIFY_LEFT, widths
(i))
        columnInfo.Insert(columnInfo.Count, column)
Next

tableInstructions.ColumnData = columnInfo

rowInfo = New Crealseq
For i = 0 To points.Count + 2
    rowInfo.Insert(rowInfo.Count, 1.0)
Next

tableInstructions.RowHeights = rowInfo
'=====
'Create the table
'Merger the top row cells to form the header
'=====
drawTable = drawing.CreateTable(tableInstructions)

topLeft = (New CCpfcTableCell).Create(1, 1)
bottomRight = (New CCpfcTableCell).Create(1, 4)

drawTable.MergeRegion(topLeft, bottomRight, Nothing)
'=====
'Write Header and add sub headings to columns
'=====
writeTextInCell(drawTable, 1, 1, "Datum Points for " + rootSolid.
FullName + _
                    " w.r.t to csys " + csysName)
writeTextInCell(drawTable, 2, 1, "Point")
writeTextInCell(drawTable, 2, 2, "X")
writeTextInCell(drawTable, 2, 3, "Y")
writeTextInCell(drawTable, 2, 4, "Z")
'=====
'Loop through all datum points
'=====
For p = 0 To points.Count - 1
'=====
'Add the point name to column 1
'=====
geomPoint = points.Item(p)
writeTextInCell(drawTable, p + 3, 1, geomPoint.GetName())
'=====
'Transform location w.r.t. csys
'=====
trfPoint = geomPoint.Point
If Not asmTransform Is Nothing Then
    trfPoint = asmTransform.TransformPoint(trfPoint)
End If
trfPoint = csysTransform.TransformPoint(trfPoint)
'=====
'Adding X, Y, Z values
'=====

```

```

        For i = 0 To 2
            writeTextInCell(drawTable, p + 3, 2 + i, Format(trfPoint.Item
(i), "#,##0.00"))
        Next
    Next

    drawTable.Display()

    Catch ex As Exception
        MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
    End Try
End Sub

Private Sub writeTextInCell(ByRef table As IpfcTable, ByVal row As Integer, _
    ByVal col As Integer, ByVal text As String)
    Dim tableCell As IpfcTableCell
    Dim lines As New Cstringseq

    tableCell = (New CCpfcTableCell).Create(row, col)
    lines.Insert(0, text)

    table.SetText(tableCell, lines)
End Sub

```

## Drawing Table Segments

Drawing tables can be constructed with one or more segments. Each segment can be independently placed. The segments are specified by an integer identifier starting with 0.

Methods and Property Introduced:

- **IpfcSelection.SelTableSegment**
- **IpfcTable.GetSegmentCount()**
- **IpfcTable.GetSegmentSheet()**
- **IpfcTable.MoveSegment()**
- **IpfcTable.GetInfo()**

The property **IpfcSelection.SelTableSegment** returns the value of the segment identifier of the selected table segment. It returns a null value if the selection does not contain a segment identifier.

The method **IpfcTable.GetSegmentCount()** returns the number of segments in the table.

The method **IpfcTable.GetSegmentSheet()** determines the sheet number that contains a specified drawing table segment.

The method **IpfcTable.MoveSegment()** moves a drawing table segment to a new location. Pass the co-ordinates of the target position in the format x, y, z=0.

### Note:

Set the value of the parameter Repaint to true to repaint the drawing with the changes. Set it to false or null to delay the repaint.

To get information about a drawing table pass the value of the segment identifier as input to the method **IpfcTable.GetInfo()**. The method returns the table information including the rotation, row and column information, and the 3D outline.

## Repeat Regions

Methods Introduced:

- **IpfcTable.IsCommentCell()**
- **IpfcTable.GetCellComponentModel()**
- **IpfcTable.GetCellReferenceModel()**
- **IpfcTable.GetCellTopModel()**
- **IpfcTableOwner.UpdateTables()**

The methods **IpfcTable.IsCommentCell()**, **IpfcTable.GetCellComponentModel()**, **IpfcTable.GetCellReferenceModel()**, **IpfcTable.GetCellTopModel()**, and **IpfcTableOwner.UpdateTables()** apply to repeat regions in drawing tables.

The method **IpfcTable.IsCommentCell()** tells you whether a cell in a repeat region contains a comment.

The method **IpfcTable.GetCellComponentModel()** returns the path to the assembly component model that is being referenced by a cell in a repeat region of a drawing table. It does not return a valid path if the cell attribute is set to "NO DUPLICATE" or "NO DUPLICATE/LEVEL".

The method **IpfcTable.GetCellReferenceModel()** returns the reference component that is being referred to by a cell in a repeat region of a drawing table, even if cell attribute is set to "NO DUPLICATE" or "NO DUPLICATE/LEVEL".

The method **IpfcTable.GetCellTopModel()** returns the top model that is being referred to by a cell in a repeat region of a drawing table, even if cell attribute is set to "NO DUPLICATE" or "NO DUPLICATE/LEVEL".

Use the method **IpfcTableOwner.UpdateTables()** to update the repeat regions in all the tables to account for changes to the model. It is equivalent to the command **Table, Repeat Region, Update**.

## Detail Items

The methods described in this section operate on detail items.

In the VB API you can create, delete and modify detail items, control their display, and query what detail items are present in the drawing. The types of detail items available are:

- Draft Entities--Contain graphical items created in Pro/Engineer. The items are as follows:
  - Arc
  - Ellipse
  - Line
  - Point
  - Polygon
  - Spline
- Notes--Textual annotations
- Symbol Definitions--Contained in the drawing's symbol gallery.
- Symbol Instances--Instances of a symbol placed in a drawing.
- Draft Groups--Groups of detail items that contain notes, symbol instances, and draft entities.

- OLE objects--Object Linking and Embedding (OLE) objects embedded in the Pro/ENGINEER drawing file.

## Listing Detail Items

Methods Introduced:

- **IpfcModelItemOwner.ListItems()**
- **IpfcDetailItemOwner.ListDetailItems()**
- **IpfcModelItemOwner.GetItemById()**
- **IpfcDetailItemOwner.CreateDetailItem()**

The method **IpfcModelItemOwner.ListItems()** returns a list of detail items specified by the parameter *Type* or returns null if no detail items of the specified type are found.

The values of the parameter *Type* for detail items are:

- EpfcITEM\_DTL\_ENTITY--Detail Entity
- EpfcITEM\_DTL\_NOTE--Detail Note
- EpfcITEM\_DTL\_GROUP--Draft Group
- EpfcITEM\_DTL\_SYM\_DEFINITION--Detail Symbol Definition
- Epfc ITEM\_DTL\_SYM\_INSTANCE--Detail Symbol Instance
- EpfcITEM\_DTL\_OLE\_OBJECT--Drawing embedded OLE object

If this parameter is set to null, then all the model items in the drawing are listed.

The method **IpfcDetailItemOwner.ListDetailItems()** also lists the detail items in the model. Pass the type of the detail item and the sheet number that contains the specified detail items.

Set the input parameter *Type* to the type of detail item to be listed. Set it to null to return all the detail items. The input parameter *SheetNumber* determines the sheet that contains the specified detail item. Pass null to search all the sheets. This argument is ignored if the parameter *Type* is set to EpfcDETAIL\_SYM\_DEFINITION.

The method returns a sequence of detail items and returns a null if no items matching the input values are found.

The method **IpfcModelItemOwner.GetItemById()** returns a detail item based on the type of the detail item and its integer identifier. The method returns a null if a detail item with the specified attributes is not found.

## Creating a Detail Item

Methods Introduced:

- **IpfcDetailItemOwner.CreateDetailItem()**
- **pfcDetail.pfcDetailGroupInstructions\_Create**

The method **IpfcDetailItemOwner.CreateDetailItem()** creates a new detail item based on the instruction data object that describes the type and content of the new detail item. The instructions data object is returned by the method **pfcDetail.pfcDetailGroupInstructions\_Create**. The method returns the newly created detail item.

## Detail Entities

A detail entity in the VB API is represented by the interface **IpfcDetailEntityItem**. It is a child of the **IpfcDetailItem**.

The interface **IpfcDetailEntityInstructions** contains specific information used to describe a detail entity item.

## Instructions

Methods and Properties Introduced:

- **CCpfcDetailEntityInstructions.Create()**
- **IpfcDetailEntityInstructions.Geometry**
- **IpfcDetailEntityInstructions.IsConstruction**
- **IpfcDetailEntityInstructions.Color**
- **IpfcDetailEntityInstructions.FontName**
- **IpfcDetailEntityInstructions.Width**
- **IpfcDetailEntityInstructions.View**

The method **CCpfcDetailEntityInstructions.Create()** creates an instructions object that describes how to construct a detail entity, for use in the methods **IpfcDetailItemOwner.CreateDetailItem()**, **IpfcDetailSymbolDefItem.CreateDetailItem()**, and **IpfcDetailEntityItem.Modify()**.

The instructions object is created based on the curve geometry and the drawing view associated with the entity. The curve geometry describes the trajectory of the detail entity in world units. The drawing view can be a model view returned by the method **IpfcModel2D.List2DViews()** or a drawing sheet background view returned by the method **IpfcSheetOwner.GetSheetBackgroundView()**. The background view indicates that the entity is not associated with a particular model view.

The method returns the created instructions object.

### Note:

Changes to the values of a **IpfcDetailEntityInstructions** object do not take effect until that instructions object is used to modify the entity using **IpfcDetailEntityItem.Modify()**.

The property **IpfcDetailEntityInstructions.Geometry** returns the geometry of the detail entity item.

For more information refer to [Curve Descriptors](#).

The property **IpfcDetailEntityInstructions.IsConstruction** returns a value that specifies whether the entity is a construction entity.

The property **IpfcDetailEntityInstructions.Color** returns the color of the detail entity item.

The property **IpfcDetailEntityInstructions.FontName** returns the line style used to draw the entity. The method returns a null value if the default line style is used.

The property **IpfcDetailEntityInstructions.Width** returns the value of the width of the entity line. The method returns a null value if the default line width is used.

The property **IpfcDetailEntityInstructions.View** returns the drawing view associated with the entity. The view can either be a model view or a drawing sheet background view.

### Example: 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.

```
Public Sub createLine(ByRef session As IpfcSession)
    Dim model As IpfcModel
    Dim rgbColour As IpfcColorRGB
    Dim drawing As IpfcDrawing
    Dim currSheet As Integer
    Dim view As IpfcView2D
    Dim mouse1 As IpfcMouseStatus
    Dim mouse2 As IpfcMouseStatus
    Dim start As IpfcPoint3D
    Dim finish As IpfcPoint3D
    Dim geom As IpfcLineDescriptor
    Dim lineInstructions As IpfcDetailEntityInstructions

    Try
        '=====  
'Get the current drawing and its background view  
'=====  
        model = session.CurrentModel  
        If model Is Nothing Then  
            Throw New Exception("Model not present")  
        End If  
        If Not model.Type = EpfcModelType.EpfcMDL_DRAWING Then  
            Throw New Exception("Model is not drawing")  
        End If  
        drawing = CType(model, IpfcDrawing)  
  
        currSheet = drawing.CurrentSheetNumber  
        view = drawing.GetSheetBackgroundView(currSheet)  
'=====  
'Set end points of the line  
'=====  
        mouse1 = session.UIGetNextMousePick(EpfcMouseButton.EpfcMOUSE_BTN_LEFT)  
        start = mouse1.Position  
  
        mouse2 = session.UIGetNextMousePick(EpfcMouseButton.EpfcMOUSE_BTN_LEFT)  
        finish = mouse2.Position  
'=====  
'Allocate and initialize curve descriptor  
'=====  
        geom = (New CCpfcLineDescriptor).Create(start, finish)  
  
        rgbColour = session.GetRGBFromStdColor(EpfcStdColor.EpfcCOLOR_QUILT)  
'=====  
'Allocate data for draft entity  
'=====  
        lineInstructions = (New CCpfcDetailEntityInstructions).Create(geom, view)  
        lineInstructions.Color = rgbColour  
'=====  
'Create and display the line  
'=====  
        drawing.CreateDetailItem(lineInstructions)  
        session.CurrentWindow.Repaint()
```

```
Catch ex As Exception
    MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
End Try

End Sub
```

## Detail Entities Information

Method and Property Introduced:

- **IpfcDetailEntityItem.GetInstructions()**
- **IpfcDetailEntityItem.SymbolDef**

The method **IpfcDetailEntityItem.GetInstructions()** returns the instructions data object that is used to construct the detail entity item.

The property **IpfcDetailEntityItem.SymbolDef** returns the symbol definition that contains the entity. This property returns a null value if the entity is not a part of a symbol definition.

## Detail Entities Operations

Methods Introduced:

- **IpfcDetailEntityItem.Draw()**
- **IpfcDetailEntityItem.Erase()**
- **IpfcDetailEntityItem.Modify()**

The method **IpfcDetailEntityItem.Draw()** temporarily draws a detail entity item, so that it is removed during the next draft regeneration.

The method **IpfcDetailEntityItem.Erase()** undraws a detail entity item temporarily, so that it is redrawn during the next draft regeneration.

The method **IpfcDetailEntityItem.Modify()** modifies the definition of an entity item using the specified instructions data object.

## OLE Objects

An object linking and embedding (OLE) object is an external file, such as a document, graphics file, or video file that is created using an external application and which can be inserted into another application, such as Pro/ENGINEER. You can create and insert supported OLE objects into a two-dimensional Pro/ENGINEER file, such as a drawing, report, format file, layout, or diagram. The functions described in this section enable you to identify and access OLE objects embedded in drawings.

Methods and Properties Introduced:

- **IpfcDetailOLEObject.ApplicationType**
- **IpfcDetailOLEObject.Outline**

- **IpfcDetailOLEObject.Path**
- **IpfcDetailOLEObject.Sheet**

The method **IpfcDetailOLEObject.ApplicationType** returns the type of the OLE object as a string, for example, "Microsoft Word Document".

The property **IpfcDetailOLEObject.Outline** returns the extent of the OLE object embedded in the drawing.

The property **IpfcDetailOLEObject.Path** returns the path to the external file for each OLE object, if it is linked to an external file.

The property **IpfcDetailOLEObject.Sheet** returns the sheet number for the OLE object.

## Detail Notes

A detail note in the VB API is represented by the interface **IpfcDetailNoteItem**. It is a child of the **IpfcDetailItem** interface.

The interface **IpfcDetailNoteInstructions** contains specific information that describes a detail note.

## Instructions

Methods and Properties Introduced:

- **CCpfcDetailNoteInstructions.Create()**
- **IpfcDetailNoteInstructions.TextLines**
- **IpfcDetailNoteInstructions.IsDisplayed**
- **IpfcDetailNoteInstructions.IsReadOnly**
- **IpfcDetailNoteInstructions.IsMirrored**
- **IpfcDetailNoteInstructions.Horizontal**
- **IpfcDetailNoteInstructions.Vertical**
- **IpfcDetailNoteInstructions.Color**
- **IpfcDetailNoteInstructions.Leader**
- **IpfcDetailNoteInstructions.TextAngle**

The method **CCpfcDetailNoteInstructions.Create()** creates a data object that describes how a detail note item should be constructed when passed to the methods **IpfcDetailItemOwner.CreateDetailItem()**, **IpfcDetailSymbolDefItem.CreateDetailItem()**, or **IpfcDetailNoteItem.Modify()**. The parameter *inTextLines* specifies the sequence of text line data objects that describe the contents of the note.

### Note:

Changes to the values of a **IpfcDetailNoteInstructions** object do not take effect until that instructions object is used to modify the note using **IpfcDetailNoteItem.Modify**

The property **IpfcDetailNoteInstructions.TextLines** returns the description of text line contents in the note.

The property **IpfcDetailNoteInstructions.IsDisplayed** returns a boolean indicating if the note is currently displayed.

The property **IpfcDetailNoteInstructions.IsReadOnly** determines whether the note can be edited by the user.

The property **IpfcDetailNoteInstructions.IsMirrored** determines whether the note is mirrored.

The property **IpfcDetailNoteInstructions.Horizontal** returns the value of the horizontal justification of the note.

The property **IpfcDetailNoteInstructions.Vertical** returns the value of the vertical justification of the note.

The property **IpfcDetailNoteInstructions.Color** returns the color of the detail note item. The method returns a null value to represent the default drawing color.

The property **IpfcDetailNoteInstructions.Leader** returns the locations of the detail note item and information about the leaders.

The property **IpfcDetailNoteInstructions.TextAngle** returns the value of the angle of the text used in the note. The method returns a null value if the angle is 0.0.

**Example: Create Drawing Note at Specified Location with Leader to Surface and Surface Name**

The following example creates a drawing note at a specified location, with a leader attached to a solid surface, and displays the name of the surface.

```
Public Sub createSurfaceNote(ByRef session As IpfcBaseSession)
    Dim model As IpfcModel
    Dim drawing As IpfcDrawing
    Dim selections As CpfcSelections
    Dim selectionOptions As IpfcSelectionOptions
    Dim selectSurface As IpfcSelection
    Dim item As IpfcModelItem
    Dim name As String
    Dim text As IpfcDetailText
    Dim texts As CpfcDetailTexts
    Dim textLine As IpfcDetailTextLine
    Dim textLines As CpfcDetailTextLines
    Dim drawingView As IpfcView2D
    Dim outline As IpfcOutline3D
    Dim textPosition As IpfcPoint3D
    Dim position As IpfcFreeAttachment
    Dim leadertoSurface As IpfcParametricAttachment
    Dim allAttachments As IpfcDetailLeaders
    Dim attachments As CpfcAttachments
    Dim noteInstructions As IpfcDetailNoteInstructions
    Dim note As IpfcDetailNoteItem

    Try
        '=====
Get the current drawing
'=====
        model = session.CurrentModel
        If model Is Nothing Then
            Throw New Exception("Model not present")
```

```

End If
If Not model.Type = EpfcModelType.EpfcMDL_DRAWING Then
    Throw New Exception("Model is not drawing")
End If
drawing = CType(model, IpfcDrawing)
'=====
Interactively select a surface
'=====
    selectionOptions = (New CCpfcSelectionOptions).Create("surface")
    selectionOptions.MaxNumSels = 1
    selections = session.Select(selectionOptions, Nothing)

    selectSurface = selections.Item(0)
    item = selectSurface.SelItem

    If (Not item.GetName Is Nothing) AndAlso Not
        (item.GetName.ToString = "")
    Then
        name = item.GetName.ToString
    Else
        name = ("Surface Id: " + item.Id.ToString)
    End If
'=====
'Allocate a text item and add it to a new text line
'=====
    text = (New CCpfcDetailText).Create(name)
    texts = New CpfcDetailTexts
    texts.Insert(0, text)
    textLine = (New CCpfcDetailTextLine).Create(texts)
    textLines = New CpfcDetailTextLines
    textLines.Insert(0, textLine)
'=====
'Set location of note text. The note is set to be slightly beyond view
'outline boundary
'=====
    drawingView = selectSurface.SelView2D
    outline = drawingView.Outline
    textPosition = outline.Item(1)

    textPosition.Set(0, textPosition.Item(0) + 0.25 *
        _(textPosition.Item(0) - outline.Item(0).Item(0)))
    textPosition.Set(1, textPosition.Item(1) + 0.25 *
        _textPosition.Item(1) - outline.Item(0).Item(1))
    position = (New CCpfcFreeAttachment).Create(textPosition)
    position.View = drawingView
'=====
Set attachment for the note leader
'=====
    leadertoSurface = (New CCpfcParametricAttachment).Create(selectSurface)
'=====
'Set attachment structure
'=====
    allAttachments = (New CCpfcDetailLeaders).Create()
    allAttachments.ItemAttachment = position

    attachments = New CpfcAttachments
    attachments.Insert(0, leadertoSurface)

    allAttachments.Leaders = attachments

```

```

' =====
'Allocate a note description and set its properties
' =====
        noteInstructions = (New CCpfcDetailNoteInstructions).Create(textLines)
        noteInstructions.Leader = allAttachments
' =====
'Create and display the note
' =====
        note = drawing.CreateDetailItem(noteInstructions)
        note.Show()

    Catch ex As Exception
        MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
        Exit Sub
    End Try
End Sub

```

## Detail Notes Information

Methods and Property Introduced:

- **IpfcDetailNoteItem.GetInstructions()**
- **IpfcDetailNoteItem.SymbolDef**
- **IpfcDetailNoteItem.GetLineEnvelope()**
- **IpfcDetailNoteItem.GetModelReference()**

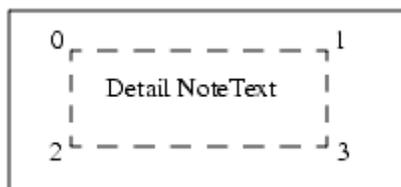
The method **IpfcDetailNoteItem.GetInstructions()** returns an instructions data object that describes how to construct the detail note item. The method takes a Boolean argument, *GiveParametersAsNames*, which identifies whether or not callouts to parameters and drawing properties should be shown in the note text as callouts, or as the displayed text value seen by the user in the note.

### Note:

Pro/ENGINEER does not resolve and replace symbolic callouts for notes which are not displayed. Therefore, if the note is not displayed or is hidden in a layer, the text retrieved may contain symbolic callouts, even when *GiveParametersAsNames* is false.

The property **IpfcDetailNoteItem.SymbolDef** returns the symbol definition that contains the note. The method returns a null value if the note is not a part of a symbol definition.

The method **IpfcDetailNoteItem.GetLineEnvelope()** determines the screen coordinates of the envelope around the detail note. This envelope is defined by four points. The following figure illustrates how the point order is determined.



The ordering of the points is maintained even if the notes are mirrored or are at an angle.

The method **IpfcDetailNoteItem.GetModelReference()** returns the model referenced by the parameterized text in a note. The model is referenced based on the line number and the text index where the parameterized text appears.

## Details Notes Operations

Methods Introduced:

- **IpfcDetailNoteItem.Draw()**
- **IpfcDetailNoteItem.Show()**
- **IpfcDetailNoteItem.Erase()**
- **IpfcDetailNoteItem.Remove()**
- **IpfcDetailNoteItem.Modify()**

The method **IpfcDetailNoteItem.Draw()** temporarily draws a detail note item, so that it is removed during the next draft regeneration.

The method **IpfcDetailNoteItem.Show()** displays the note item, such that it is repainted during the next draft regeneration.

The method **IpfcDetailNoteItem.Erase()** undraws a detail note item temporarily, so that it is redrawn during the next draft regeneration.

The method **IpfcDetailNoteItem.Remove()** undraws a detail note item permanently, so that it is not redrawn during the next draft regeneration.

The method **IpfcDetailNoteItem.Modify()** modifies the definition of an existing detail note item based on the instructions object that describes the new detail note item.

## Detail Groups

A detail group in the VB API is represented by the interface **IpfcDetailGroupItem**. It is a child of the **IpfcDetailItem** interface.

The interface **IpfcDetailGroupInstructions** contains information used to describe a detail group item.

## Instructions

Method and Properties Introduced:

- **CCpfcDetailGroupInstructions.Create()**
- **IpfcDetailGroupInstructions.Name**
- **IpfcDetailGroupInstructions.Elements**
- **IpfcDetailGroupInstructions.IsDisplayed**

The method **CCpfcDetailGroupInstructions.Create()** creates an instruction data object that describes how to construct a detail group for use in **IpfcDetailItemOwner.CreateDetailItem()** and **IpfcDetailGroupItem.Modify()**.

### Note:

Changes to the values of a **IpfcDetailGroupInstructions** object do not take effect until that instructions object is

used to modify the group using `IpfcDetailGroupItem.Modify`.

The property `IpfcDetailGroupInstructions.Name` returns the name of the detail group.

The property `IpfcDetailGroupInstructions.Elements` returns the sequence of the detail items (notes, groups and entities) contained in the group.

The property `IpfcDetailGroupInstructions.IsDisplayed` returns whether the detail group is displayed in the drawing.

## Detail Groups Information

Method Introduced:

- **`IpfcDetailGroupItem.GetInstructions()`**

The method `IpfcDetailGroupItem.GetInstructions()` gets a data object that describes how to construct a detail group item. The method returns the data object describing the detail group item.

## Detail Groups Operations

Methods Introduced:

- **`IpfcDetailGroupItem.Draw()`**
- **`IpfcDetailGroupItem.Erase()`**
- **`IpfcDetailGroupItem.Modify()`**

The method `IpfcDetailGroupItem.Draw()` temporarily draws a detail group item, so that it is removed during the next draft generation.

The method `IpfcDetailGroupItem.Erase()` temporarily undraws a detail group item, so that it is redrawn during the next draft generation.

The method `IpfcDetailGroupItem.Modify()` changes the definition of a detail group item based on the data object that describes how to construct a detail group item.

### Example: Create New Group of Items

The following example creates a group from a set of selected detail items.

```
Public Sub createGroup(ByRef session As IpfcBaseSession, ByVal
                      groupName As String)

    Dim selections As CpfcSelections
    Dim selectionOptions As IpfcSelectionOptions
    Dim items As CpfcDetailItems
    Dim i As Integer
    Dim drawing As IpfcDrawing
    Dim groupInstructions As IpfcDetailGroupInstructions

    Try
'=====
'Select notes, draft entities, symbol instances
```

```

'=====
        selectionOptions = (New CCpfcSelectionOptions).Create("any_note,
draft_ent,dtl_symbol")
        selections = session.Select(selectionOptions, Nothing)

        If selections Is Nothing Or selections.Count = 0 Then
            Throw New Exception("No Detail tem selected")
        End If
'=====
'Allocate and fill a sequence with the detail item handles
'=====
        items = New CpfcDetailItems

        For i = 0 To selections.Count - 1
            items.Insert(items.Count, selections.Item(i).SelItem)
        Next
'=====
'Get the drawing which owns the group
=====
        drawing = items.Item(0).DBParent
'=====
'Allocate group data and set group items
'=====
        groupInstructions = (New CCpfcDetailGroupInstructions).Create(groupName,
items)

        drawing.CreateDetailItem(groupInstructions)
        For i = 0 To selections.Count - 1
            selections.Item(i).UnHighlight()
        Next
        session.CurrentWindow.Repaint()

    Catch ex As Exception
        MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
    End Try
End Sub

```

## Detail Symbols

### Detail Symbol Definitions

A detail symbol definition in the VB API is represented by the interface **IpfcDetailSymbolDefItem**. It is a child of the **IpfcDetailItem** interface.

The interface **IpfcDetailSymbolDefInstructions** contains information that describes a symbol definition. It can be used when creating symbol definition entities or while accessing existing symbol definition entities.

### Instructions

Methods and Properties Introduced:

- **CCpfcDetailSymbolDefInstructions.Create()**
- **IpfcDetailSymbolDefInstructions.SymbolHeight**
- **IpfcDetailSymbolDefInstructions.HasElbow**

- **IpfcDetailSymbolDefInstructions.IsTextAngleFixed**
- **IpfcDetailSymbolDefInstructions.ScaledHeight**
- **IpfcDetailSymbolDefInstructions.Attachments**
- **IpfcDetailSymbolDefInstructions.FullPath**
- **IpfcDetailSymbolDefInstructions.Reference**

The method **CCpfcDetailSymbolDefInstructions.Create()** creates an instruction data object that describes how to create a symbol definition based on the path and name of the symbol definition. The instructions object is passed to the methods **pfcDetailItemOwner.CreateDetailItem** and **pfcDetailSymbolDefItem.Modify**.

**Note:**

Changes to the values of a **IpfcDetailSymbolDefInstructions** object do not take effect until that instructions object is used to modify the definition using the method **pfcDetail.SymbolDefItem.Modify**.

The property **IpfcDetailSymbolDefInstructions.SymbolHeight** returns the value of the height type for the symbol definition. The symbol definition height options are as follows:

- **EpfcSYMDEF\_FIXED**--Symbol height is fixed.
- **EpfcSYMDEF\_VARIABLE**--Symbol height is variable.
- **EpfcSYMDEF\_RELATIVE\_TO\_TEXT**--Symbol height is determined relative to the text height.

The property **IpfcDetailSymbolDefInstructions.HasElbow** determines whether the symbol definition includes an elbow.

The property **IpfcDetailSymbolDefInstructions.IsTextAngleFixed** returns whether the text of the angle is fixed.

The property **IpfcDetailSymbolDefInstructions.ScaledHeight** returns the height of the symbol definition in inches.

The property **IpfcDetailSymbolDefInstructions.Attachments** returns the value of the sequence of the possible instance attachment points for the symbol definition.

The property **IpfcDetailSymbolDefInstructions.FullPath** returns the value of the complete path of the symbol definition file.

The property **IpfcDetailSymbolDefInstructions.Reference** returns the text reference information for the symbol definition. It returns a null value if the text reference is not used. The text reference identifies the text item used for a symbol definition which has a height type of **SYMDEF\_TEXT\_RELATED**.

## Detail Symbol Definitions Information

Methods Introduced:

- **IpfcDetailSymbolDefItem.ListDetailItems()**
- **IpfcDetailSymbolDefItem.GetInstructions()**

The method **IpfcDetailSymbolDefItem.ListDetailItems()** lists the detail items in the symbol definition based on the type of the detail item.

The method **IpfcDetailSymbolDefItem.GetInstructions()** returns an instruction data object that describes how to construct the symbol definition.

## Detail Symbol Definitions Operations

Methods Introduced:

- **IpfcDetailSymbolDefItem.CreateDetailItem()**
- **IpfcDetailSymbolDefItem.Modify()**

The method **IpfcDetailSymbolDefItem.CreateDetailItem()** creates a detail item in the symbol definition based on the instructions data object. The method returns the detail item in the symbol definition.

The method **IpfcDetailSymbolDefItem.Modify()** modifies a symbol definition based on the instructions data object that contains information about the modifications to be made to the symbol definition.

## Retrieving Symbol Definitions

Methods Introduced:

- **IpfcDetailItemOwner.RetrieveSymbolDefinition()**

The method **IpfcDetailItemOwner.RetrieveSymbolDefinition()** retrieves a symbol definition from the disk.

The input parameters of this method are:

- FileName--Name of the symbol definition file
- FilePath--Path to the symbol definition file. It is relative to the path specified by the option "pro\_symbol\_dir" in the configuration file. A null value indicates that the function should search the current directory.
- Version--Numerical version of the symbol definition file. A null value retrieves the latest version.
- UpdateUnconditionally--True if Pro/ENGINEER should update existing instances of this symbol definition, or false to quit the operation if the definition exists in the model.

The method returns the retrieved symbol definition.

### Example : Create Symbol Definition

The following example creates a symbol definition which contains four line entities forming a box, a note at the middle of the box, and a free attachment.

```
Public Sub createBoxSymbolDef(ByRef session As IpfcBaseSession, _
                             ByVal name As String, ByVal text As String)
    Dim model As IpfcModel
    Dim drawing As IpfcDrawing
    Dim symbolInstructions As IpfcDetailSymbolDefInstructions
    Dim origin As CpfcPoint3D
    Dim attachment As IpfcSymbolDefAttachment
    Dim attachments As CpfcSymbolDefAttachments
    Dim symbolDef As IpfcDetailSymbolDefItem
    Dim textHeight As Double
    Dim matrix As IpfcTransform3D
    Dim defHeight As Double
    Dim rgbColour As IpfcColorRGB
    Dim end1 As CpfcPoint3D
    Dim end2 As CpfcPoint3D
```

Try

```
'=====
'Get the current drawing
'=====
    model = session.CurrentModel
    If model Is Nothing Then
        Throw New Exception("Model not present")
    End If
    If Not model.Type = EpfcModelType.EpfcMDL_DRAWING Then
        Throw New Exception("Model is not drawing")
    End If
    drawing = CType(model, IpfcDrawing)
'=====
'Allocate symbol definition description data
'=====
    symbolInstructions = (New CCpfcDetailSymbolDefInstructions).Create(name)
    symbolInstructions.Height = EpfcSymbolDefHeight.EpfcSYMDEF_FIXED
'=====
'Set a free attachment at the origin of the symbol
'=====
    origin = New CpfcPoint3D
    origin.Set(0, 0.0)
    origin.Set(1, 0.0)
    origin.Set(2, 0.0)

    attachment = (New CCpfcSymbolDefAttachment).Create
                (EpfcSymbolDefAttachmentType.EpfcSYMDEFATTACH_FREE, origin)

    attachments = New CpfcSymbolDefAttachments
    attachments.Insert(0, attachment)
    symbolInstructions.Attachments = attachments
'=====
'Create empty symbol
'=====
    symbolDef = drawing.CreateDetailItem(symbolInstructions)
'=====
'Calculate the default text height for the symbol based on the drawing
'text(height And transform)
'=====
    textHeight = drawing.TextHeight
    matrix = drawing.GetSheetTransform(drawing.CurrentSheetNumber)
    defHeight = textHeight / matrix.Matrix.Item(0, 0)

    rgbColour = session.GetRGBFromStdColor(EpfcStdColor.EpfcCOLOR_QUILT)
'=====
'Create four lines to form a box, twice the default text height,
'around the origin
'=====
    end1 = New CpfcPoint3D
    end2 = New CpfcPoint3D
    end1.Set(0, -defHeight)
    end1.Set(1, -defHeight)
    end1.Set(2, 0.0)
    end2.Set(0, defHeight)
    end2.Set(1, -defHeight)
    end2.Set(2, 0.0)

    addLine(symbolDef, end1, end2, rgbColour)
```

```

        end2.Set(0, -defHeight)
        end2.Set(1, defHeight)

        addLine(symbolDef, end1, end2, rgbColour)

        end1.Set(0, defHeight)
        end1.Set(1, defHeight)

        addLine(symbolDef, end1, end2, rgbColour)

        end2.Set(0, defHeight)
        end2.Set(1, -defHeight)

        addLine(symbolDef, end1, end2, rgbColour)
'=====  

'Add a note with the specified text at the origin  

'=====  

        addNote(symbolDef, origin, text)

    Catch ex As Exception
        MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
    Exit Sub
End Try
End Sub

Private Sub addLine(ByRef symDef As IpfcDetailSymbolDefItem,
                    _ByVal start As IpfcPoint3D,
                    ByVal finish As IpfcPoint3D,
                    _ByVal colour As IpfcColorRGB)

    Dim geom As IpfcLineDescriptor
    Dim lineInstructions As IpfcDetailEntityInstructions
'=====  

'Allocate and initialize curve descriptor  

'=====  

        geom = (New CCpfcLineDescriptor).Create(start, finish)
'=====  

'Allocate data for draft entity  

'=====  

        lineInstructions = (New CCpfcDetailEntityInstructions).Create(geom, Nothing)
        lineInstructions.Color = colour
'=====  

'Create and display the line  

'=====  

        symDef.CreateDetailItem(lineInstructions)

End Sub

Private Sub addNote(ByRef symDef As IpfcDetailSymbolDefItem,
                    _ByVal location As IpfcPoint3D,
                    _ByVal message As String)

    Dim text As IpfcDetailText
    Dim texts As CpfcDetailTexts
    Dim textLine As IpfcDetailTextLine
    Dim textLines As CpfcDetailTextLines
    Dim position As IpfcFreeAttachment
    Dim allAttachments As IpfcDetailLeaders

```

```

    Dim noteInstructions As IpfcDetailNoteInstructions
' =====
' Allocate a text item and add it to a new text line
' =====
    text = (New CCpfcDetailText).Create(message)
    texts = New CpfcDetailTexts
    texts.Insert(0, text)
    textLine = (New CCpfcDetailTextLine).Create(texts)
    textLines = New CpfcDetailTextLines
    textLines.Insert(0, textLine)
' =====
' Set the location of the note text
' =====
    position = (New CCpfcFreeAttachment).Create(location)
' =====
' Set the attachment structure
' =====
    allAttachments = (New CCpfcDetailLeaders).Create()
    allAttachments.ItemAttachment = position
' =====
' Allocate note description
' =====
    noteInstructions = (New CCpfcDetailNoteInstructions).Create(textLines)
    noteInstructions.Leader = allAttachments
    noteInstructions.Horizontal = EpfcHorizontalJustification.
EpfcH_JUSTIFY_CENTER
    noteInstructions.Vertical = EpfcVerticalJustification.EpfcV_JUSTIFY_MIDDLE

    symDef.CreateDetailItem(noteInstructions)

End Sub

```

## Detail Symbol Instances

A detail symbol instance in the VB API is represented by the interface **IpfcDetailSymbolInstItem**. It is a child of the **IpfcDetailItem** interface.

The interface **IpfcDetailSymbolInstInstructions** contains information that describes a symbol instance. It can be used when creating symbol instances and while accessing existing groups.

### Instructions

Methods and Properties Introduced:

- **CCpfcDetailSymbolInstInstructions.Create()**
- **IpfcDetailSymbolInstInstructions.IsDisplayed**
- **IpfcDetailSymbolInstInstructions.Color**
- **IpfcDetailSymbolInstInstructions.SymbolDef**
- **IpfcDetailSymbolInstInstructions.AttachOnDefType**
- **IpfcDetailSymbolInstInstructions.DefAttachment**

- **IpfcDetailSymbolInstInstructions.InstAttachment**
- **IpfcDetailSymbolInstInstructions.Angle**
- **IpfcDetailSymbolInstInstructions.ScaledHeight**
- **IpfcDetailSymbolInstInstructions.TextValues**
- **IpfcDetailSymbolInstInstructions.CurrentTransform**
- **IpfcDetailSymbolInstInstructions.SetGroups()**

The method **CCpfcDetailSymbolInstInstructions.Create()** creates a data object that contains information about the placement of a symbol instance.

**Note:**

Changes to the values of a **IpfcDetailSymbolInstInstructions** object do not take effect until that instructions object is used to modify the instance using **IpfcDetailSymbolInstItem.Modify**.

The property **IpfcDetailSymbolInstInstructions.IsDisplayed** returns a value that specifies whether the instance of the symbol is displayed.

The property **IpfcDetailSymbolInstInstructions.Color** returns the color of the detail symbol instance. A null value indicates that the default drawing color is used.

The property **IpfcDetailSymbolInstInstructions.SymbolDef** returns the symbol definition used for the instance.

The property **IpfcDetailSymbolInstInstructions.AttachOnDefType** returns the attachment type of the instance. The method returns a null value if the attachment represents a free attachment. The attachment options are as follows:

- **EpcfSYMDEFATTACH\_FREE**--Attachment on a free point.
- **EpcfSYMDEFATTACH\_LEFT\_LEADER**--Attachment via a leader on the left side of the symbol.
- **EpcfSYMDEFATTACH\_RIGHT\_LEADER**-- Attachment via a leader on the right side of the symbol.
- **EpcfSYMDEFATTACH\_RADIAL\_LEADER**--Attachment via a leader at a radial location.
- **EpcfSYMDEFATTACH\_ON\_ITEM**--Attachment on an item in the symbol definition.
- **EpcfSYMDEFATTACH\_NORMAL\_TO\_ITEM**--Attachment normal to an item in the symbol definition.

The property **IpfcDetailSymbolInstInstructions.DefAttachment** returns the value that represents the way in which the instance is attached to the symbol definition.

The property **IpfcDetailSymbolInstInstructions.InstAttachment** returns the value of the attachment of the instance that includes location and leader information.

The property **IpfcDetailSymbolInstInstructions.Angle** returns the value of the angle at which the instance is placed. The method returns a null value if the value of the angle is 0 degrees.

The property **IpfcDetailSymbolInstInstructions.ScaledHeight** returns the height of the symbol instance in the owner drawing or model coordinates. This value is consistent with the height value shown for a symbol instance in the Properties dialog box in the Pro/ENGINEER User Interface.

**Note:**

The scaled height obtained using the above property is partially based on the properties of the symbol definition assigned using the property **pfcDetail.DetailSymbolInstInstructions.GetSymbolDef**. Changing the symbol definition may change the calculated value for the scaled height.

The property **IpfcDetailSymbolInstInstructions.TextValues** returns the sequence of variant text values used while

placing the symbol instance.

The property **IpfcDetailSymbolInstInstructions.CurrentTransform** returns the coordinate transformation matrix to place the symbol instance.

The method **IpfcDetailSymbolInstInstructions.SetGroups()** sets the *IpfcDetailSymbolGroupOption* argument for displaying symbol groups in the symbol instance. This argument can have the following values:

- **EpfcDETAIL\_SYMBOL\_GROUP\_INTERACTIVE**--Symbol groups are interactively selected for display. This is the default value in the GRAPHICS mode.
- **EpfcDETAIL\_SYMBOL\_GROUP\_ALL**--All non-exclusive symbol groups are included for display.
- **EpfcDETAIL\_SYMBOL\_GROUP\_NONE**--None of the non-exclusive symbol groups are included for display.
- **EpfcDETAIL\_SYMBOL\_GROUP\_CUSTOM**--Symbol groups specified by the application are displayed.

Refer to the section [Detail Symbol Groups](#) for more information on detail symbol groups.

## Detail Symbol Instances Information

Method Introduced:

- **IpfcDetailSymbolInstItem.GetInstructions()**

The method **IpfcDetailSymbolInstItem.GetInstructions()** returns an instructions data object that describes how to construct a symbol instance.

## Detail Symbol Instances Operations

Methods Introduced:

- **IpfcDetailSymbolInstItem.Draw()**
- **IpfcDetailSymbolInstItem.Erase()**
- **IpfcDetailSymbolInstItem.Show()**
- **IpfcDetailSymbolInstItem.Remove()**
- **IpfcDetailSymbolInstItem.Modify()**

The method **IpfcDetailSymbolInstItem.Draw()** draws a symbol instance temporarily to be removed on the next draft regeneration.

The method **IpfcDetailSymbolInstItem.Erase()** undraws a symbol instance temporarily from the display to be redrawn on the next draft generation.

The method **IpfcDetailSymbolInstItem.Show()** displays a symbol instance to be repainted on the next draft regeneration.

The method **IpfcDetailSymbolInstItem.Remove()** deletes a symbol instance permanently.

The method **IpfcDetailSymbolInstItem.Modify()** modifies a symbol instance based on the instructions data object that contains information about the modifications to be made to the symbol instance.

**Example: Create a Free Instance of Symbol Definition**

```
'Place free symbol instance
'=====
'Function      :   placeSymbolInstance
'Purpose       :   This function creates a free instance of a symbol
'               :   definition. A symbol is placed with no leaders at a
'               :   specified location.
'=====
```

```
Public Sub placeSymbolInstance(ByRef session As IpfcSession, _
                               ByVal symbolName As String)
```

```
    Dim model As IpfcModel
    Dim drawing As IpfcDrawing
    Dim symbolDefinition As IpfcDetailSymbolDefItem
    Dim point As CpfcPoint3D
    Dim mouse As IpfcMouseStatus
    Dim symInstructions As IpfcDetailSymbolInstInstructions
    Dim position As IpfcFreeAttachment
    Dim allAttachments As IpfcDetailLeaders
    Dim symItem As IpfcDetailSymbolInstItem
```

```
    Try
```

```
'=====
'Get the current drawing
'=====
```

```
    model = session.CurrentModel
    If model Is Nothing Then
        Throw New Exception("Model not present")
    End If
    If Not model.Type = EpfcModelType.EpfcMDL_DRAWING Then
        Throw New Exception("Model is not drawing")
    End If
    drawing = CType(model, IpfcDrawing)
```

```
'=====
'Retrieve symbol definition from system
'=====
```

```
    symbolDefinition = drawing.RetrieveSymbolDefinition
(symbolName, _"/", _Nothing,
_Nothing)
```

```
'=====
'Select location for symbol
'=====
```

```
    point = New CpfcPoint3D
    mouse = session.UIGetNextMousePick(EpfcMouseButton.EpfcMOUSE_BTN_LEFT)
    point = mouse.Position
```

```
'=====
'Allocate the symbol instance decription
'=====
```

```
    symInstructions = (New CCpfcDetailSymbolInstInstructions).Create
(symbolDefinition)
```

```
'=====
'Set the location of the note text
'=====
```

```
    position = (New CCpfcFreeAttachment).Create(point)
```

```
'=====
'Set the attachment structure
'=====
```

```
    allAttachments = (New CCpfcDetailLeaders).Create()
```

```

        allAttachments.ItemAttachment = position

        symInstructions.InstAttachment = allAttachments
'=====
'Create and display the symbol
'=====
        symItem = drawing.CreateDetailItem(symInstructions)
        symItem.Show()

        Catch ex As Exception
            MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
        End Try
    End Sub

```

**Example: Create a Free Instance of a Symbol Definition with drawing unit heights, variable text and groups**

```

'Place detail symbol instance
'=====
'Function      : placeDetailSymbol
'Purpose      : This function creates a free instance of a symbol
'              : definition with drawing unit heights, variable text and
'              : groups. A symbol is placed with no leaders at a
'              : specified location.
'=====
Public Sub placeDetailSymbol(ByRef session As IpfcSession, ByVal
                            groupName As String, _
                            Optional ByVal variableText As String =
                                Nothing, _
                            Optional ByVal height As Double = 0)

    Dim model As IpfcModel
    Dim drawing As IpfcDrawing
    Dim symbolDefinition As IpfcDetailSymbolDefItem
    Dim point As CpfcPoint3D
    Dim mouse As IpfcMouseStatus
    Dim symInstructions As IpfcDetailSymbolInstInstructions
    Dim position As IpfcFreeAttachment
    Dim allAttachments As IpfcDetailLeaders
    Dim symItem As IpfcDetailSymbolInstItem

    Dim varTexts As IpfcDetailVariantTexts
    Dim varText As IpfcDetailVariantText

    Dim allGroups As IpfcDetailSymbolGroups
    Dim groups As IpfcDetailSymbolGroups
    Dim group As IpfcDetailSymbolGroup

    Try
'=====
        'Get the current drawing
        '=====
        model = session.CurrentModel
        If model Is Nothing Then
            Throw New Exception("Model not present")
        End If
        If Not model.Type = EpfcModelType.EpfcMDL_DRAWING Then

```

```

        Throw New Exception("Model is not drawing")
    End If
    drawing = CType(model, IpfcDrawing)

    '=====
    'Retrieve symbol definition from system
    '=====
    symbolDefinition = drawing.RetrieveSymbolDefinition
                        ("detail_symbol_example", _
                        "./", _
                        Nothing, Nothing)

'=====
    'Select location for symbol
'=====
    point = New CpfcPoint3D
    mouse =
    session.UIGetNextMousePick(EpfcMouseButton.EpfcMOUSE_BTN_LEFT)
    point = mouse.Position

'=====
    'Allocate the symbol instance decription
'=====
    symInstructions = (New
        CCpfcDetailSymbolInstInstructions).Create(symbolDefinition)

'=====
    'Set the new values
'=====
    If height > 0 Then
        symInstructions.ScaledHeight = 15.5
    End If

    If Not variableText Is Nothing Then
        varText = (New CCpfcDetailVariantText).Create("VAR_TEXT",
            variableText)
        varTexts = New CpfcDetailVariantTexts
        varTexts.Append(varText)

        symInstructions.TextValues = varTexts
    End If

    Select Case groupName
        Case "ALL"
            symInstructions.SetGroups(EpfcDetailSymbolGroupOption.
EpfcDETAIL_SYMBOL_
                GROUP_ALL, Nothing)
        Case "NONE"
            symInstructions.SetGroups(EpfcDetailSymbolGroupOption.EpfcDETAIL_SYMBOL_
                GROUP_NONE, Nothing)
        Case Else
            allGroups = symInstructions.SymbolDef.ListSubgroups
            group = getGroup(allGroups, groupName)
            If Not group Is Nothing Then
                groups = New CpfcDetailSymbolGroups
                groups.Append(group)

            symInstructions.SetGroups(EpfcDetailSymbolGroupOption.EpfcDETAIL_SYMBOL_

```

```

        GROUP_CUSTOM, groups)
    End If
End Select

'=====
'Set the location of the note text
'=====
    position = (New CCpfcFreeAttachment).Create(point)

'=====
'Set the attachment structure
'=====
    allAttachments = (New CCpfcDetailLeaders).Create()
    allAttachments.ItemAttachment = position

    symInstructions.InstAttachment = allAttachments

'=====
'Create and display the symbol
'=====
    symItem = drawing.CreateDetailItem(symInstructions)
    symItem.Show()

    Catch ex As Exception
        MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
    End Try

End Sub

Private Function getGroup(ByRef groups As CpfcDetailSymbolGroups,
    ByVal groupName As String) As IpfcDetailSymbolGroup
    Dim group As IpfcDetailSymbolGroup
    Dim groupInstrs As IpfcDetailSymbolGroupInstructions
    Dim i As Integer

    If groups.Count = 0 Then
        Return Nothing
    End If

    For i = 0 To groups.Count - 1
        group = groups.Item(i)
        groupInstrs = group.GetInstructions()

        If groupInstrs.Name = groupName Then
            Return group
        End If
    Next

    Return Nothing
End Function

```

## Detail Symbol Groups

A detail symbol group in the VB API is represented by the interface **IpfcDetailSymbolGroup**. It is a child of the **IpfcObject** interface. A detail symbol group is accessible only as a part of the contents of a detail symbol definition or instance.

The interface **IpfcDetailSymbolGroupInstructions** contains information that describes a symbol group. It can be used when creating new symbol groups, or while accessing or modifying existing groups.

## Instructions

Methods and Properties Introduced:

- **CCpfcDetailSymbolGroupInstructions.Create()**
- **IpfcDetailSymbolGroupInstructions.Items**
- **IpfcDetailSymbolGroupInstructions.Name**

The method **CCpfcDetailSymbolGroupInstructions.Create()** creates the **IpfcDetailSymbolGroupInstructions** data object that stores the name of the symbol group and the list of detail items to be included in the symbol group.

### Note:

Changes to the values of the **IpfcDetailSymbolGroupInstructions** data object do not take effect until this object is used to modify the instance using the method **IpfcDetailSymbolGroup.Modify**.

The property **IpfcDetailSymbolGroupInstructions.Items** returns the list of detail items included in the symbol group.

The property **IpfcDetailSymbolGroupInstructions.Name** returns the name of the symbol group.

## Detail Symbol Group Information

Methods Introduced:

- **IpfcDetailSymbolGroup.GetInstructions()**
- **IpfcDetailSymbolGroup.ParentGroup**
- **IpfcDetailSymbolGroup.ParentDefinition**
- **IpfcDetailSymbolGroup.ListChildren()**
- **IpfcDetailSymbolDefItem.ListSubgroups()**
- **IpfcDetailSymbolDefItem.IsSubgroupLevelExclusive()**
- **IpfcDetailSymbolInstItem.ListGroups()**

The method **IpfcDetailSymbolGroup.GetInstructions()** returns the **IpfcDetailSymbolGroupInstructions** data object that describes how to construct a symbol group.

The method **IpfcDetailSymbolGroup.ParentGroup** returns the parent symbol group to which a given symbol group belongs.

The method **IpfcDetailSymbolGroup.ParentDefinition** returns the symbol definition of a given symbol group.

The method **IpfcDetailSymbolGroup.ListChildren()** lists the subgroups of a given symbol group.

The method **IpfcDetailSymbolDefItem.ListSubgroups()** lists the subgroups of a given symbol group stored in the symbol definition at the indicated level.

The method **IpfcDetailSymbolDefItem.IsSubgroupLevelExclusive()** identifies if the subgroups of a given symbol group stored in the symbol definition at the indicated level are exclusive or independent. If groups are exclusive, only one of the groups at this level can be active in the model at any time. If groups are independent, any number of groups can be active.

The method **IpfcDetailSymbolInstItem.ListGroups()** lists the symbol groups included in a symbol instance. The *IpfcSymbolGroupFilter* argument determines the types of symbol groups that can be listed. It takes the following values:

- **EpcfDTLSYMINST\_ALL\_GROUPS**--Retrieves all groups in the definition of the symbol instance.
- **EpcfDTLSYMINST\_ACTIVE\_GROUPS**--Retrieves only those groups that are actively shown in the symbol instance.
- **EpcfDTLSYMINST\_INACTIVE\_GROUPS**--Retrieves only those groups that are not shown in the symbol instance.

## Detail Symbol Group Operations

Methods Introduced:

- **IpfcDetailSymbolGroup.Delete()**
- **IpfcDetailSymbolGroup.Modify()**
- **IpfcDetailSymbolDefItem.CreateSubgroup()**
- **IpfcDetailSymbolDefItem.SetSubgroupLevelExclusive()**
- **IpfcDetailSymbolDefItem.SetSubgroupLevelIndependent()**

The method **IpfcDetailSymbolGroup.Delete()** deletes the specified symbol group from the symbol definition. This method does not delete the entities contained in the group.

The method **IpfcDetailSymbolGroup.Modify()** modifies the specified symbol group based on the **IpfcDetailSymbolGroupInstructions** data object that contains information about the modifications that can be made to the symbol group.

The method **IpfcDetailSymbolDefItem.CreateSubgroup()** creates a new subgroup in the symbol definition at the indicated level below the parent group.

The method **IpfcDetailSymbolDefItem.SetSubgroupLevelExclusive()** makes the subgroups of a symbol group exclusive at the indicated level in the symbol definition.

### Note:

After you set the subgroups of a symbol group as exclusive, only one of the groups at the indicated level can be active in the model at any time.

The method **IpfcDetailSymbolDefItem.SetSubgroupLevelIndependent()** makes the subgroups of a symbol group independent at the indicated level in the symbol definition.

### Note:

After you set the subgroups of a symbol group as independent, any number of groups at the indicated level can be active in the model at any time.

## Detail Attachments

A detail attachment in VB API is represented by the interface **IpfcAttachment**. It is used for the following tasks:

- The way in which a drawing note or a symbol instance is placed in a drawing.
- The way in which a leader on a drawing note or symbol instance is attached.

Method Introduced:

- **IpfcAttachment.GetType()**

The method **IpfcAttachment.GetType()** returns the **IpfcAttachmentType** object containing the types of detail attachments. The detail attachment types are as follows:

- EpcfATTACH\_FREE--The attachment is at a free point possibly with respect to a given drawing view.
- EpcfATTACH\_PARAMETRIC--The attachment is to a point on a surface or an edge of a solid.
- EpcfATTACH\_OFFSET--The attachment is offset to another drawing view, to a model item, or to a 3D model annotation.
- EpcfATTACH\_TYPE\_UNSUPPORTED--The attachment is to an item that cannot be represented in PFC at the current time. However, you can still retrieve the location of the attachment.

## Free Attachment

The EpcfATTACH\_FREE detail attachment type is represented by the interface **IpfcFreeAttachment**. It is a child of the **IpfcAttachment** interface.

Properties Introduced:

- **IpfcFreeAttachment.AttachmentPoint**
- **IpfcFreeAttachment.View**

The property **IpfcFreeAttachment.AttachmentPoint** returns the attachment point. This location is in screen coordinates for drawing items, symbol instances and surface finishes on flat-to-screen annotation planes, and in model coordinates for symbols and surface finishes on 3D model annotation planes.

The method **IpfcFreeAttachment.View** returns the drawing view to which the attachment is related. The attachment point is relative to the drawing view, that is the attachment point moves when the drawing view is moved. This method returns a NULL value, if the detail attachment is not related to a drawing view, but is placed at the specified location in the drawing sheet, or if the attachment is offset to a model item or to a 3D model annotation.

## Parametric Attachment

The EpcfATTACH\_PARAMETRIC detail attachment type is represented by the interface **IpfcParametricAttachment**. It is a child of the **IpfcAttachment** interface.

Property Introduced:

- **IpfcParametricAttachment.AttachedGeometry**

The property **IpfcParametricAttachment.AttachedGeometry** returns the **IpfcSelection** object representing the item to which the detail attachment is attached. This includes the drawing view in which the attachment is made.

## Offset Attachment

The EpcfATTACH\_OFFSET detail attachment type is represented by the interface **IpfcOffsetAttachment**. It is a child of the **IpfcAttachment** interface.

Properties Introduced:

- **IpfcOffsetAttachment.AttachedGeometry**
- **IpfcOffsetAttachment.AttachmentPoint**

The property **IpfcOffsetAttachment.AttachedGeometry** returns the **IpfcSelection** object representing the item to which the detail attachment is attached. This includes the drawing view where the attachment is made, if the offset reference is in a model.

The property **IpfcOffsetAttachment.AttachmentPoint** returns the attachment point. This location is in screen coordinates for drawing items, symbol instances and surface finishes on flat-to-screen annotation planes, and in model coordinates for symbols and surface finishes on 3D model annotation planes. The distance from the attachment point to the location of the item to which the detail attachment is attached is saved as the offset distance.

## Unsupported Attachment

The EpcfATTACH\_TYPE\_UNSUPPORTED detail attachment type is represented by the interface **IpfcUnsupportedAttachment**. It is a child of the **IpfcAttachment** interface.

Property Introduced:

- **IpfcUnsupportedAttachment.AttachmentPoint**

The property **IpfcUnsupportedAttachment.AttachmentPoint** returns the attachment point. This location is in screen coordinates for drawing items, symbol instances and surface finishes on flat-to-screen annotation planes, and in model coordinates for symbols and surface finishes on 3D model annotation planes.

---

# Solid

---

Most of the objects and methods in the VB API are used with solid models (parts and assemblies). Because solid objects inherit from the interface `IpfcModel`, you can use any of the `IpfcModel` methods on any `IpfcSolid`, `IpfcPart`, or `IpfcAssembly` object.

## Topic

[Getting a Solid Object](#)

[Solid Information](#)

[Solid Operations](#)

[Solid Units](#)

[Mass Properties](#)

[Annotations](#)

[Cross Sections](#)

[Materials](#)

## Getting a Solid Object

Methods and Properties Introduced:

- **`IpfcBaseSession.CreatePart()`**
- **`IpfcBaseSession.CreateAssembly()`**
- **`IpfcComponentPath.Root`**
- **`IpfcComponentPath.Leaf`**
- **`IpfcMFG.GetSolid()`**

The methods **`IpfcBaseSession.CreatePart()`** and **`IpfcBaseSession.CreateAssembly()`** create new solid models with the names you specify.

The properties **`IpfcComponentPath.Root`** and **`IpfcComponentPath.Leaf`** specify the solid objects that make up the component path of an assembly component model. You can get a component path object from any component that has been interactively selected.

The method **`IpfcMFG.GetSolid()`** retrieves the storage solid in which the manufacturing model's features are placed. In order to create a UDF group in the manufacturing model, call the method **`IpfcSolid.CreateUDFGroup()`** on the storage solid.

## Solid Information

Properties Introduced:

- **`IpfcSolid.RelativeAccuracy`**
- **`IpfcSolid.AbsoluteAccuracy`**

You can set the relative and absolute accuracy of any solid model using these methods. Relative accuracy is relative to the

size of the solid. For example, a relative accuracy of .01 specifies that the solid must be accurate to within 1/100 of its size. Absolute accuracy is measured in absolute units (inches, centimeters, and so on).

**Note:**

For a change in accuracy to take effect, you must regenerate the model.

## Solid Operations

Methods and Properties Introduced:

- **IpfcSolid.Regenerate()**
- **CCpfcRegenInstructions.Create()**
- **IpfcRegenInstructions.AllowFixUI**
- **IpfcRegenInstructions.ForceRegen**
- **IpfcRegenInstructions.FromFeat**
- **IpfcRegenInstructions.RefreshModelTree**
- **IpfcRegenInstructions.ResumeExcludedComponents**
- **IpfcRegenInstructions.UpdateAssemblyOnly**
- **IpfcRegenInstructions.UpdateInstances**
- **IpfcSolid.GeomOutline**
- **IpfcSolid.EvalOutline()**
- **IpfcSolid.IsSkeleton**

The method **IpfcSolid.Regenerate()** causes the solid model to regenerate according to the instructions provided in the form of the **IpfcRegenInstructions** object. Passing a null value for the instructions argument causes an automatic regeneration. The **IpfcRegenInstructions** object contains the following input parameters:

- AllowFixUI--Determines whether or not to activate the Fix Model user interface, if there is an error.

Use the property **IpfcRegenInstructions.AllowFixUI** to modify this parameter.

- ForceRegen--Forces the solid model to fully regenerate. All the features in the model are regenerated. If this parameter is false, Pro/ENGINEER determines which features to regenerate. By default, it is false.

Use the property **IpfcRegenInstructions.ForceRegen** to modify this parameter.

- FromFeat--Not currently used. This parameter is reserved for future use.

Use the property **IpfcRegenInstructions.FromFeat** to modify this parameter.

- RefreshModelTree--Refreshes the Pro/ENGINEER Model Tree after regeneration. The model must be active to use this attribute. If this attribute is false, the Model Tree is not refreshed. By default, it is false.

Use the property **IpfcRegenInstructions.RefreshModelTree** to modify this parameter.

- ResumeExcludedComponents--Enables Pro/ENGINEER to resume the available excluded components of the simplified representation during regeneration. This results in a more accurate update of the simplified representation.

Use the property **IpfcRegenInstructions.ResumeExcludedComponents** to modify this parameter.

- UpdateAssemblyOnly--Updates the placements of an assembly and all its sub-assemblies, and regenerates the assembly features and intersected parts. If the affected assembly is retrieved as a simplified representation, then the locations of the components are updated. If this attribute is false, the component locations are not updated, even if the simplified representation is retrieved. By default, it is false.

Use the property **IpfcRegenInstructions.UpdateAssemblyOnly** to modify this parameter.

- UpdateInstances--Updates the instances of the solid model in memory. This may slow down the regeneration process. By default, this attribute is false.

Use the property **IpfcRegenInstructions.UpdateInstances** to modify this parameter.

The property **IpfcSolid.GeomOutline** returns the three-dimensional bounding box for the specified solid. The method **IpfcSolid.EvalOutline()** also returns a three-dimensional bounding box, but you can specify the coordinate system used to compute the extents of the solid object.

The property **IpfcSolid.IsSkeleton** determines whether the part model is a skeleton or a concept model. It returns a true value if the model is a skeleton, else it returns a false.

## Solid Units

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

The following types of quantities govern the definition of units of measurement:

- Basic Quantities--The basic units and dimensions of the system of units. For example, consider the Centimeter Gram Second (CGS) system of unit. The basic quantities for this system of units are:
  - Length--cm
  - Mass--g
  - Force--dyne
  - Time--sec
  - Temperature--K
- Derived Quantities--The derived units are those that are derived from the basic quantities. For example, consider the Centimeter Gram Second (CGS) system of unit. The derived quantities for this system of unit are as follows:
  - Area--cm<sup>2</sup>
  - Volume--cm<sup>3</sup>
  - Velocity--cm/sec

In the VB API, individual units in the model are represented by the interface **pfcUnits.Unit**.

## Types of Unit Systems

The types of systems of units are as follows:

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

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

- Mass Length Time (MLT)--The following systems of units belong to this category:
  - CGS --Centimeter Gram Second
  - MKS--Meter Kilogram Second
  - mmKS--millimeter Kilogram Second
- Force Length Time (FLT)--The following systems of units belong to this category:
  - Pro/ENGINEER Default--Inch lbm Second. This is the default system followed by Pro/ENGINEER.
  - FPS--Foot Pound Second

- IPS--Inch Pound Second
- mmNS--Millimeter Newton Second

In the VB API, the system of units followed by the model is represented by the interface **pfcUnits.UnitSystem**.

## Accessing Individual Units

Methods and Properties Introduced:

- **IpfcSolid.ListUnits()**
- **IpfcSolid.GetUnit()**
- **IpfcUnit.Name**
- **IpfcUnit.Expression**
- **IpfcUnit.Type**
- **IpfcUnit.IsStandard**
- **IpfcUnit.ReferenceUnit**
- **IpfcUnit.ConversionFactor**
- **IpfcUnitConversionFactor.Offset**
- **IpfcUnitConversionFactor.Scale**

The method **IpfcSolid.ListUnits()** returns the list of units available to the specified model.

The method **IpfcSolid.GetUnit()** retrieves the unit, based on its name or expression for the specified model in the form of the **IpfcUnit** object.

The property **IpfcUnit.Name** returns the name of the unit.

The property **IpfcUnit.Expression** returns a user-friendly unit description in the form of the name (for example, ksi) for ordinary units and the expression (for example, N/m<sup>3</sup>) for system-generated units.

The property **IpfcUnit.Type** returns the type of quantity represented by the unit in terms of the **IpfcUnitType** object. The types of units are as follows:

- EpfcUNIT\_LENGTH--Specifies length measurement units.
- EpfcUNIT\_MASS--Specifies mass measurement units.
- EpfcUNIT\_FORCE--Specifies force measurement units.
- EpfcUNIT\_TIME--Specifies time measurement units.
- EpfcUNIT\_TEMPERATURE--Specifies temperature measurement units.
- EpfcUNIT\_ANGLE--Specifies angle measurement units.

The property **IpfcUnit.IsStandard** identifies whether the unit is system-defined (if the property *IsStandard* is set to true) or user-defined (if the property *IsStandard* is set to false).

The property **IpfcUnit.ReferenceUnit** returns a reference unit (one of the available system units) in terms of the **IpfcUnit** object.

The property **IpfcUnit.ConversionFactor** identifies the relation of the unit to its reference unit in terms of the

**IpfcUnitConversionFactor** object. The unit conversion factors are as follows:

- Offset--Specifies the offset value applied to the values in the reference unit.
- Scale--Specifies the scale applied to the values in the reference unit to get the value in the actual unit.

Example - Consider the formula to convert temperature from Centigrade to Fahrenheit  
 $F = a + (C * b)$

where

F is the temperature in Fahrenheit

C is the temperature in Centigrade

a = 32 (constant signifying the offset value)

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

**Note:**

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

Use the properties **IpfcUnitConversionFactor.Offset** and **IpfcUnitConversionFactor.Scale** to retrieve the unit conversion factors listed above.

## Modifying Individual Units

Methods and Properties Introduced:

- **IpfcUnit.Modify()**
- **IpfcUnit.Delete()**

The method **IpfcUnit.Modify()** modifies the definition of a unit by applying a new conversion factor specified by the **IpfcUnitConversionFactor** object and a reference unit.

The method **IpfcUnit.Delete()** deletes the unit.

**Note:**

You can delete only custom units and not standard units.

## Creating a New Unit

Methods Introduced:

- **IpfcSolid.CreateCustomUnit()**
- **CCpfcUnitConversionFactor.Create()**

The method **IpfcSolid.CreateCustomUnit()** creates a custom unit based on the specified name, the conversion factor given by the **IpfcUnitConversionFactor** object, and a reference unit.

The method **CCpfcUnitConversionFactor.Create()** creates the **IpfcUnitConversionFactor** object containing the unit conversion factors.

## Accessing Systems of Units

Methods and Properties Introduced:

- **IpfcSolid.ListUnitSystems()**

- **IpfcSolid.GetPrincipalUnits()**
- **IpfcUnitSystem.GetUnit()**
- **IpfcUnitSystem.Name**
- **IpfcUnitSystem.Type**
- **IpfcUnitSystem.IsStandard**

The method **IpfcSolid.ListUnitSystems()** returns the list of unit systems available to the specified model.

The method **IpfcSolid.GetPrincipalUnits()** returns the system of units assigned to the specified model in the form of the **IpfcUnitSystem** object.

The method **IpfcUnitSystem.GetUnit()** retrieves the unit of a particular type used by the unit system.

The property **IpfcUnitSystem.Name** returns the name of the unit system.

The property **IpfcUnitSystem.Type** returns the type of the unit system in the form of the **IpfcUnitSystemType** object. The types of unit systems are as follows:

- **EpcfUNIT\_SYSTEM\_MASS\_LENGTH\_TIME**--Specifies the Mass Length Time (MLT) unit system.
- **EpcfUNIT\_SYSTEM\_FORCE\_LENGTH\_TIME**--Specifies the Force Length Time (FLT) unit system.

For more information on these unit systems listed above, refer to the section [Types of Unit Systems](#).

The property **IpfcUnitSystem.IsStandard** identifies whether the unit system is system-defined (if the property *IsStandard* is set to true) or user-defined (if the property *IsStandard* is set to false).

## Modifying Systems of Units

Method Introduced:

- **IpfcUnitSystem.Delete()**

The method **IpfcUnitSystem.Delete()** deletes a custom-defined system of units.

### Note:

You can delete only a custom-defined system of units and not a standard system of units.

## Creating a New System of Units

Method Introduced:

- **IpfcSolid.CreateUnitSystem()**

The method **IpfcSolid.CreateUnitSystem()** creates a new system of units in the model based on the specified name, the type of unit system given by the **IpfcUnitSystemType** object, and the types of units specified by the **IpfcUnits** sequence to use for each of the base measurement types (length, force or mass, and temperature).

## Conversion to a New Unit System

Methods and Properties Introduced:

- **IpfcSolid.SetPrincipalUnits()**
- **CCpfcUnitConversionOptions.Create()**
- **IpfcUnitConversionOptions.DimensionOption**
- **IpfcUnitConversionOptions.IgnoreParamUnits**

The method **IpfcSolid.SetPrincipalUnits()** changes the principal system of units assigned to the solid model based on the unit conversion options specified by the **IpfcUnitConversionOptions** object. The method **CCpfcUnitConversionOptions.Create()** creates the **IpfcUnitConversionOptions** object containing the unit conversion options listed below.

The types of unit conversion options are as follows:

- DimensionOption--Use the option while converting the dimensions of the model.

Use the property **IpfcUnitConversionOptions.DimensionOption** to modify this option.

This option can be of the following types:

- EpcfUNITCONVERT\_SAME\_DIMS--Specifies that unit conversion occurs by interpreting the unit value in the new unit system. For example, 1 inch will equal to 1 millimeter.
- EpcfUNITCONVERT\_SAME\_SIZE--Specifies that unit conversion will occur by converting the unit value in the new unit system. For example, 1 inch will equal to 25.4 millimeters.
- IgnoreParamUnits--This boolean attribute determines whether or not ignore the parameter units. If it is null or true, parameter values and units do not change when the unit system is changed. If it is false, parameter units are converted according to the rule.

Use the property **IpfcUnitConversionOptions.IgnoreParamUnits** to modify this attribute.

## Mass Properties

Method Introduced:

- **IpfcSolid.GetMassProperty()**

The function **IpfcSolid.GetMassProperty()** 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 an object containing the following fields:

- The volume.
- The surface area.
- The density. The density value is 1.0, unless a material has been assigned.
- The mass.
- The center of gravity (COG).
- The inertia matrix.
- The inertia tensor.
- The inertia about the COG.
- The principal moments of inertia (the eigen values of the COG inertia).
- The principal axes (the eigenvectors of the COG inertia).

### Example Code: Retrieving a Mass Property Object

This method retrieves a MassProperty object from a specified solid model. The solid's mass, volume, and center of gravity point are then printed.

```
Imports pfcls
```

```
Public Class pfcSolidExamples
```

```
    Public Sub printMassProperties(ByRef session As IpfcBaseSession)
```

```
        Dim model As IpfcModel
```

```
        Dim solid As IpfcSolid
```

```
        Dim solidProperties As IpfcMassProperty
```

```
        Dim gravityCentre As New CpfcPoint3D
```

```
        Try
```

```
'=====
```

```
'Get the current solid
```

```
'=====
```

```
    model = session.CurrentModel
```

```
    If model Is Nothing Then
```

```
        Throw New Exception("Model not present")
```

```
    End If
```

```
    If (Not model.Type = EpfcModelType.EpfcMDL_PART) And
```

```
        _(Not model.Type = EpfcModelType.EpfcMDL_ASSEMBLY) Then
```

```
        Throw New Exception("Model is not a solid")
```

```
    End If
```

```
    solid = CType(model, IpfcSolid)
```

```
'=====
```

```
'Get the solid properties. Optional argument in this method is the name
```

```
'of the coordinate system to use. If null, uses default
```

```
'=====
```

```
    solidProperties = solid.GetMassProperty(Nothing)
```

```
    gravityCentre = solidProperties.GravityCenter
```

```
    MsgBox("The solid mass is: " + solidProperties.Mass.ToString +
```

```
        Chr(13).ToString + _"The solid volume is: " +
```

```
        solidProperties.Volume.ToString +
```

```
        Chr(13).ToString + _"The Centre of Gravity is at: " +
```

```
        Chr(13).ToString + _"X : " +
```

```
        gravityCentre.Item(0).ToString + Chr(13).ToString +
```

```
        _"Y : " + gravityCentre.Item(1).ToString +
```

```
        Chr(13).ToString +
```

```
        _"Z : " + gravityCentre.Item(2).ToString +
```

```
        Chr(13).ToString)
```

```
    Catch ex As Exception
```

```
        MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
```

```
    Exit Sub
```

```
End Try
```

```
End Sub
```

```
End Class
```

## Annotations

Methods and Properties Introduced:

- **IpfcNote.Lines**
- **IpfcNote.URL**
- **IpfcNote.Display()**

- **IpfcNote.Delete()**
- **IpfcNote.GetOwner()**

3D model notes are instance of ModelItem objects. They can be located and accessed using methods that locate model items in solid models, and downcast to the Note interface to use the methods in this section.

The property **IpfcNote.Lines** returns the text contained in the 3D model note.

The property **IpfcNote.URL** returns the URL stored in the 3D model note.

The method **IpfcNote.Display()** forces the display of the model note.

The method **IpfcNote.Delete()** deletes a model note.

The method **IpfcNote.GetOwner()** returns the solid model owner of the note.

## Cross Sections

Methods Introduced:

- **IpfcSolid.ListCrossSections()**
- **IpfcSolid.GetCrossSection()**
- **IpfcXSection.GetName()**
- **IpfcXSection.SetName()**
- **IpfcXSection.GetXSecType()**
- **IpfcXSection.Delete()**
- **IpfcXSection.Display()**
- **IpfcXSection.Regenerate()**

The method **IpfcSolid.ListCrossSections()** returns a sequence of cross section objects represented by the Xsection interface. The method **IpfcSolid.GetCrossSection()** searches for a cross section given its name.

The method **IpfcXSection.GetName()** returns the name of the cross section in Pro/ENGINEER. The method **IpfcXSection.SetName()** modifies the cross section name.

The method **IpfcXSection.GetXSecType()** returns the type of cross section, that is planar or offset, and the type of item intersected by the cross section.

The method **IpfcXSection.Delete()** deletes a cross section.

The method **IpfcXSection.Display()** forces a display of the cross section in the window.

The method **IpfcXSection.Regenerate()** regenerates a cross section.

## Materials

The VB API enables you to programmatically access the material types and properties of parts. Using the methods and

properties described in the following sections, you can perform the following actions:

- Create or delete materials
- Set the current material
- Access and modify the material types and properties

Methods and Properties Introduced:

- **IpfcMaterial.Save()**
- **IpfcMaterial.Delete()**
- **IpfcPart.CurrentMaterial**
- **IpfcPart.ListMaterials()**
- **IpfcPart.CreateMaterial()**
- **IpfcPart.RetrieveMaterial()**

The method **IpfcMaterial.Save()** writes to a material file that can be imported into any Pro/ENGINEER part.

The method **IpfcMaterial.Delete()** removes material from the part.

The property **IpfcPart.CurrentMaterial** returns and sets the material assigned to the part.

**Note:**

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

The method **IpfcPart.ListMaterials()** returns a list of the materials available in the part.

The method **IpfcPart.CreateMaterial()** creates a new empty material in the specified part.

The method **IpfcPart.RetrieveMaterial()** imports a material file into the part. The name of the file read can be as either:

- `<name>.mtl`--Specifies the new material file format.
- `<name>.mat`--Specifies the material file format prior to Pro/ENGINEER Wildfire 3.0.

If the material is not already in the part database, **IpfcPart.RetrieveMaterial()** 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.

## Accessing Material Types

Properties Introduced:

- **IpfcMaterial.StructuralMaterialType**
- **IpfcMaterial.ThermalMaterialType**

- **IpfcMaterial.SubType**
- **IpfcMaterial.PermittedSubTypes**

The property **IpfcMaterial.StructuralMaterialType** sets the material type for the structural properties of the material. The material types are as follows:

- EpfcMTL\_ISOTROPIC--Specifies a material with an infinite number of planes of material symmetry, making the properties equal in all directions.
- EpfcMTL\_ORTHOTROPIC--Specifies a material with symmetry relative to three mutually perpendicular planes.
- EpfcMTL\_TRANSVERSELY\_ISOTROPIC--Specifies a material with rotational symmetry about an axis. The properties are equal for all directions in the plane of isotropy.

The property **IpfcMaterial.ThermalMaterialType** sets the material type for the thermal properties of the material. The material types are as follows:

- EpfcMTL\_ISOTROPIC--Specifies a material with an infinite number of planes of material symmetry, making the properties equal in all directions.
- EpfcMTL\_ORTHOTROPIC--Specifies a material with symmetry relative to three mutually perpendicular planes.
- EpfcMTL\_TRANSVERSELY\_ISOTROPIC--Specifies a material with rotational symmetry about an axis. The properties are equal for all directions in the plane of isotropy.

The property **IpfcMaterial.SubType** returns sets the subtype for the EpfcMTL\_ISOTROPIC material type.

Use the property **IpfcMaterial.PermittedSubTypes** to retrieve a list of the permitted string values for the material subtype.

## Accessing Material Properties

The methods and properties listed in this section enable you to access material properties.

Methods and Properties Introduced:

- **CCpfcMaterialProperty.Create()**
- **IpfcMaterial.GetPropertyValue()**
- **IpfcMaterial.SetPropertyValue()**
- **IpfcMaterial.SetPropertyUnits()**
- **IpfcMaterial.RemoveProperty()**
- **IpfcMaterial.Description**
- **IpfcMaterial.FatigueType**
- **IpfcMaterial.PermittedFatigueTypes**
- **IpfcMaterial.FatigueMaterialType**
- **IpfcMaterial.PermittedFatigueMaterialTypes**
- **IpfcMaterial.FatigueMaterialFinish**
- **IpfcMaterial.PermittedFatigueMaterialFinishes**

- **IpfcMaterial.FailureCriterion**
- **IpfcMaterial.PermittedFailureCriteria**
- **IpfcMaterial.Hardness**
- **IpfcMaterial.HardnessType**
- **IpfcMaterial.Condition**
- **IpfcMaterial.BendTable**
- **IpfcMaterial.CrossHatchFile**
- **IpfcMaterial.MaterialModel**
- **IpfcMaterial.PermittedMaterialModels**
- **IpfcMaterial.ModelDefByTests**

The method **CCpfcMaterialProperty.Create()** creates a new instance of a material property object.

All numerical material properties are accessed using the same set of APIs. You must provide a property type to indicate the property you want to read or modify.

The method **IpfcMaterial.GetPropertyValue()** returns the value and the units of the material property.

Use the method **IpfcMaterial.SetPropertyValue()** to set the value and units of the material property. If the property type does not exist for the material, then this method creates it.

Use the method **IpfcMaterial.SetPropertyUnits()** to set the units of the material property.

Use the method **IpfcMaterial.RemoveProperty()** to remove the material property.

Material properties that are non-numeric can be accessed using the following properties.

The property **IpfcMaterial.Description** sets the description string for the material.

The property **IpfcMaterial.FatigueType** and sets the valid fatigue type for the material.

Use the property **IpfcMaterial.PermittedFatigueTypes** to get a list of the permitted string values for the fatigue type.

The property **IpfcMaterial.FatigueMaterialType** sets the class of material when determining the effect of the fatigue.

Use the property **IpfcMaterial.PermittedFatigueMaterialTypes** to retrieve a list of the permitted string values for the fatigue material type.

The property **IpfcMaterial.FatigueMaterialFinish** sets the type of surface finish for the fatigue material.

Use the property **IpfcMaterial.PermittedFatigueMaterialFinishes** to retrieve a list of permitted string values for the fatigue material finish.

The property **IpfcMaterial.FailureCriterion** sets the reduction factor for the failure strength of the material. This factor is used to reduce the endurance limit of the material to account for unmodeled stress concentrations, such as those found in welds. Use the property **IpfcMaterial.PermittedFailureCriteria** to retrieve a list of permitted string values for the material failure criterion.

The property **IpfcMaterial.Hardness** sets the hardness for the specified material.

The property **IpfcMaterial.HardnessType** sets the hardness type for the specified material.

The property **IpfcMaterial.Condition** sets the condition for the specified material.

The property **IpfcMaterial.BendTable** sets the bend table for the specified material.

The property **IpfcMaterial.CrossHatchFile** sets the file containing the crosshatch pattern for the specified material.

The property **IpfcMaterial.MaterialModel** sets the type of hyperelastic isotropic material model.

Use the property **IpfcMaterial.PermittedMaterialModels** to retrieve a list of the permitted string values for the material model.

The property **IpfcMaterial.ModelDefByTests** determines whether the hyperelastic isotropic material model has been defined using experimental data for stress and strain.

## Accessing User-defined Material Properties

Materials permit assignment of user-defined parameters. These parameters allow you to place non-standard properties on a given material. Therefore `IpfcMaterial` is a child of `IpfcParameterOwner`, which provides access to user-defined parameters and properties of materials through the methods in that interface.

---

# Windows and Views

---

The VB API provides access to Pro/ENGINEER windows and saved views. This section describes the methods that provide this access.

## Topic

[Windows](#)

[Embedded Browser](#)

[Views](#)

[Coordinate Systems and Transformations](#)

## Windows

This section describes the VB API methods that access window objects. The topics are as follows:

- Getting a Window Object
- Window Operations

## Getting a Window Object

Methods and Property Introduced:

- **IpfcBaseSession.CurrentWindow**
- **IpfcBaseSession.CreateModelWindow()**
- **IpfcModel.Display()**
- **IpfcBaseSession.ListWindows()**
- **IpfcBaseSession.GetWindow()**
- **IpfcBaseSession.OpenFile()**
- **IpfcBaseSession.GetModelWindow()**

The property **IpfcBaseSession.CurrentWindow** provides access to the current active window in Pro/ENGINEER.

The method **IpfcBaseSession.CreateModelWindow()** creates a new window that contains the model that was passed as an argument.

### Note:

You must call the method **IpfcModel.Display()** for the model geometry to be displayed in the window.

Use the method **IpfcBaseSession.ListWindows()** to get a list of all the current windows in session.

The method **IpfcBaseSession.GetWindow()** gets the handle to a window given its integer identifier.

The method **IpfcBaseSession.OpenFile()** returns the handle to a newly created window that contains the opened

model.

**Note:**

If a model is already open in a window the method returns a handle to the window.

The method **IpfcBaseSession.GetModelWindow()** returns the handle to the window that contains the opened model, if it is displayed.

## Window Operations

Methods and Properties Introduced:

- **IpfcWindow.Height**
- **IpfcWindow.Width**
- **IpfcWindow.XPos**
- **IpfcWindow.YPos**
- **IpfcWindow.GraphicsAreaHeight**
- **IpfcWindow.GraphicsAreaWidth**
- **IpfcWindow.Clear()**
- **IpfcWindow.Repaint()**
- **IpfcWindow.Refresh()**
- **IpfcWindow.Close()**
- **IpfcWindow.Activate()**

The properties **IpfcWindow.Height**, **IpfcWindow.Width**, **IpfcWindow.XPos**, and **IpfcWindow.YPos** retrieve the height, width, x-position, and y-position of the window respectively. The values of these parameters are normalized from 0 to 1.

The properties **IpfcWindow.GraphicsAreaHeight** and **IpfcWindow.GraphicsAreaWidth** retrieve the height and width of the Pro/ENGINEER graphics area window without the border respectively. The values of these parameters are normalized from 0 to 1. For both the window and graphics area sizes, if the object occupies the whole screen, the window size returned is 1. For example, if the screen is 1024 pixels wide and the graphics area is 512 pixels, then the width of the graphics area window is returned as 0.5.

The method **IpfcWindow.Clear()** removes geometry from the window.

Both **IpfcWindow.Repaint()** and **IpfcWindow.Refresh()** repaint solid geometry. However, the **Refresh** method does not remove highlights from the screen and is used primarily to remove temporary geometry entities from the screen.

Use the method **IpfcWindow.Close()** to close the window. If the current window is the original window created when Pro/ENGINEER started, this method clears the window. Otherwise, it removes the window from the screen.

The method **IpfcWindow.Activate()** activates a window. This function is available only in the asynchronous mode.

# Embedded Browser

Methods Introduced:

- **IpfcWindow.GetURL()**
- **IpfcWindow.SetURL()**
- **IpfcWindow.GetBrowserSize()**
- **IpfcWindow.SetBrowserSize()**

The methods **IpfcWindow.GetURL()** and **IpfcWindow.SetURL()** enables you to find and change the URL displayed in the embedded browser in the Pro/ENGINEER window.

The methods **IpfcWindow.GetBrowserSize()** and **IpfcWindow.SetBrowserSize()** enables you to find and change the size of the embedded browser in the Pro/ENGINEER window.

## Views

This section describes the the VB API methods that access `IpfcView` objects. The topics are as follows:

- Getting a View Object
- View Operations

### Getting a View Object

Methods Introduced:

- **IpfcViewOwner.RetrieveView()**
- **IpfcViewOwner.GetView()**
- **IpfcViewOwner.ListViews()**
- **IpfcViewOwner.GetCurrentView()**

Any solid model inherits from the interface `IpfcViewOwner`. This will enable you to use these methods on any solid object.

The method **IpfcViewOwner.RetrieveView()** sets the current view to the orientation previously saved with a specified name.

Use the method **IpfcViewOwner.GetView()** to get a handle to a named view without making any modifications.

The method **IpfcViewOwner.ListViews()** returns a list of all the views previously saved in the model.

The method **IpfcViewOwner.GetCurrentView()** returns a view handle that represents the current orientation. Although this view does not have a name, you can use this view to find or modify the current orientation.

## View Operations

Methods and Properties Introduced:

- **IpfcView.Name**
- **IpfcView.IsCurrent**
- **IpfcView.Reset()**
- **IpfcViewOwner.SaveView()**

To get the name of a view given its identifier, use the property **IpfcView.Name**.

The property **IpfcView.IsCurrent** determines if the View object represents the current view.

The **IpfcView.Reset()** method restores the current view to the default view.

To store the current view under the specified name, call the method **IpfcViewOwner.SaveView()**.

## Coordinate Systems and Transformations

This section describes the various coordinate systems used by Pro/ENGINEER and accessible from the VB API and how to transform from one coordinate system to another.

### Coordinate Systems

Pro/ENGINEER and the VB API 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 the VB API for all the methods that look at geometry and most of the methods that draw three-dimensional graphics.

#### Screen Coordinate System

The screen coordinate system is 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 nominal pixel counts. 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 methods, the mouse input methods, and all methods 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 take account of zoom and pan. For example, you can find out whether a point on the solid is visible in the window or you can 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 VB API methods and properties that manipulate drawings generally use screen coordinates.

## **Drawing View Coordinate System**

The 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, subassemblies, and the geometry of datum features created in the assembly.

When an assembly is retrieved into memory each member is also loaded 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 the VB API 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.

## **Transformations**

Methods and Properties Introduced:

- **IpfcTransform3D.Invert()**
- **IpfcTransform3D.TransformPoint()**
- **IpfcTransform3D.TransformVector()**
- **IpfcTransform3D.Matrix**
- **IpfcTransform3D.GetOrigin()**
- **IpfcTransform3D.GetXAxis()**
- **IpfcTransform3D.GetYAxis()**
- **IpfcTransform3D.GetZAxis()**

All coordinate systems are treated in the VB API as if they were three-dimensional. Therefore, a point in any of the coordinate systems is always represented by the `IpfcPoint3D` class:

Vectors store the same data but are represented for clarity by the `IpfcVector3D` class.

Screen coordinates contain a z-value whose positive direction is outwards from the screen. The value of z is not generally important when specifying a screen location as an input to a method, but it is useful in other situations. For example, if you select a datum plane, you can find the direction of the plane 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 the `IpfcTransform3D` class. This class contains a 4x4 matrix that combines the conventional 3x3 matrix that describes the relative orientation of the two systems, and the vector that describes the shift between them.

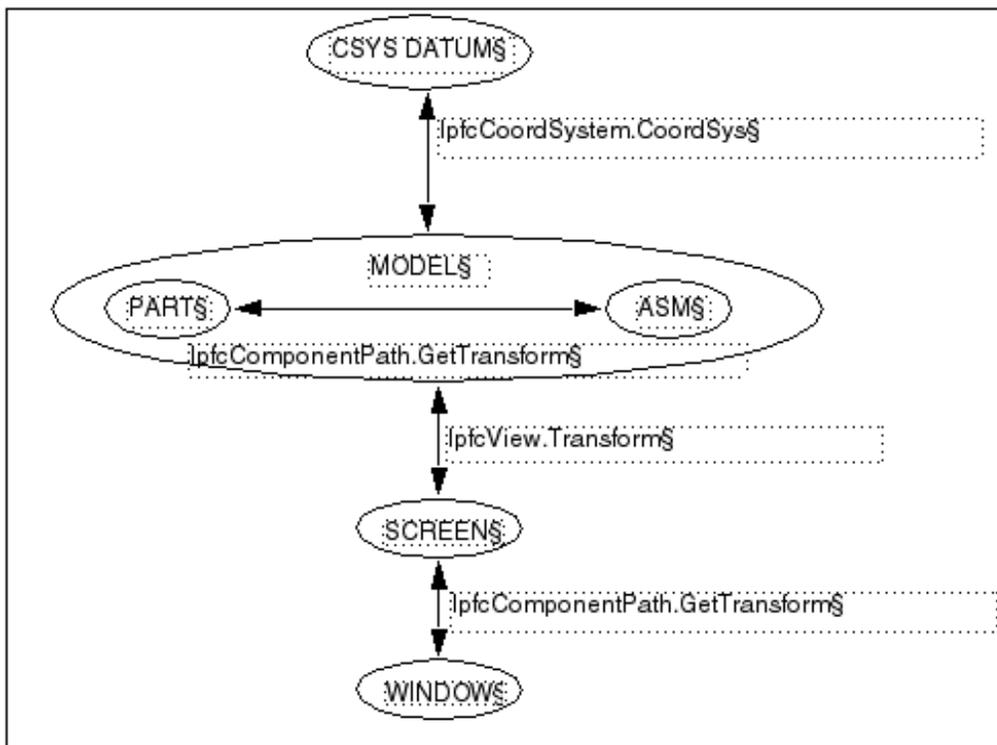
$$\begin{bmatrix} X' & Y' & Z' & 1 \end{bmatrix} = \begin{bmatrix} X & Y & Z & 1 \end{bmatrix} \begin{bmatrix} \dots & \dots & \dots & 0 \\ \dots & \dots & \dots & 0 \\ \dots & \dots & \dots & 0 \\ X_s & Y_s & Z_s & 1 \end{bmatrix}$$

The 4x4 matrix used for transformations is as follows:

The utility method **IpfcTransform3D.Invert()** inverts a transformation matrix so that it can be used to transform points in the opposite direction.

The VB API provides two utilities for performing coordinate transformations. The method **IpfcTransform3D.TransformPoint()** transforms a three-dimensional point and **IpfcTransform3D.TransformVector()** transforms a three-dimensional vector.

The following diagram summarizes the coordinate transformations needed when using the VB API and specifies the the VB API methods that provide the transformation matrix.



## Transforming to Screen Coordinates

Methods and Properties Introduced:

- **IpfcView.Transform**
- **IpfcView.Rotate()**

The view matrix describes the transformation from solid to screen coordinates. The property **IpfcView.Transform** provides the view matrix for the specified view.

The method **IpfcView.Rotate()** rotates a view, relative to the X, Y, or Z axis, in the amount that you specify.

To transform from screen to solid coordinates, invert the transformation matrix using the method **IpfcTransform3D.Invert()**.

## Transforming to Coordinate System Datum Coordinates

Property Introduced:

- **IpfcCoordSystem.CoordSys**

The property **IpfcCoordSystem.CoordSys** provides the location and orientation of the coordinate system datum in the 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.

## Transforming Window Coordinates

Properties Introduced

- **IpfcWindow.ScreenTransform**

- **IpfcScreenTransform.PanX**
- **IpfcScreenTransform.PanY**
- **IpfcScreenTransform.Zoom**

You can alter the pan and zoom of a window by using a Screen Transform object. This object contains three attributes. PanX and PanY represent the horizontal and vertical movement. Every increment of 1.0 moves the view point one screen width or height. Zoom represents a scaling factor for the view. This number must be greater than zero.

## Transforming Coordinates of an Assembly Member

Method Introduced:

- **IpfcComponentPath.GetTransform()**

The method **IpfcComponentPath.GetTransform()** 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.

### Example Code - Normalizing a Coordinate Transformation Matrix

The following example code uses two methods to transfer the view transformation from one view to another. Imports pfcls

```
Public Class pfcViewExamples

    Public Function viewTransfer(ByVal view1 As IpfcView,
                               _ByVal view2 As IpfcView) As IpfcView

        Dim transform As IpfcTransform3D
        Dim matrix As IpfcMatrix3D

        Try
            transform = view1.Transform
            matrix = transform.Matrix
            matrix = matrixNormalize(matrix)
            transform.Matrix = matrix
            view2.Transform = transform
            viewTransfer = view2

        Catch ex As Exception
            MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
            Return Nothing
        End Try

    End Function

    '=====  

    'Function : matrixNormalize  

    'Purpose : This function normalizes a Matrix3D object  

    '=====  

    Private Function matrixNormalize(ByVal matrix As IpfcMatrix3D) As IpfcMatrix3D
        Dim scale As Double
        Dim row, col As Integer
    '=====  

    'Set bottom row to 0.0  

    '=====  

        matrix.Set(3, 0, 0.0)
```

```
matrix.Set(3, 1, 0.0)
matrix.Set(3, 2, 0.0)

scale = Math.Sqrt(matrix.Item(0, 0) * matrix.Item(0, 0) +
    matrix.Item(0, 1) * _matrix.Item(0, 1) +
    matrix.Item(0, 2) * matrix.Item(0, 2))

For row = 0 To 2
    For col = 0 To 2
        matrix.Set(row, col, matrix.Item(row, col) / scale)
    Next
Next

matrixNormalize = matrix

End Function
End Class
```

---

# ModelItem

---

This section describes the the VB API methods that enable you to access and manipulate `ModelItems`.

## Topic

[Solid Geometry Traversal](#)

[Getting ModelItem Objects](#)

[ModelItem Information](#)

[Layer Objects](#)

## Solid Geometry Traversal

Solid models are made up of 11 distinct types of `IpfcModelItem`, as follows:

- `IpfcFeature`
- `IpfcSurface`
- `IpfcEdge`
- `IpfcCurve` (datum curve)
- `IpfcAxis` (datum axis)
- `IpfcPoint` (datum point)
- `IpfcQuilt` (datum quilt)
- `IpfcLayer`
- `IpfcNote`
- `IpfcDimension`
- `IpfcRefDimension`

Each model item is assigned a unique identification number that will never change. In addition, each model item can be assigned a string name. Layers, points, axes, dimensions, and reference dimensions are automatically assigned a name that can be changed.

## Getting ModelItem Objects

Methods and Properties Introduced:

- **IpfcModelItemOwner.ListItems()**
- **IpfcFeature.ListSubItems()**
- **IpfcLayer.ListItems()**
- **IpfcModelItemOwner.GetItemById()**
- **IpfcModelItemOwner.GetItemByName()**
- **IpfcFamColModelItem.RefItem**
- **IpfcSelection.SelItem**

All models inherit from the interface `IpfcModelItemOwner`. The method **IpfcModelItemOwner.ListItems()** returns a sequence of `IpfcModelItems` contained in the model. You can specify which type of `IpfcModelItem` to collect by passing in one of the enumerated `EpfcModelItemType` values, or you can collect all `IpfcModelItems` by passing **null** as the model item type.

The methods **IpfcFeature.ListSubItems()** and **IpfcLayer.ListItems()** produce similar results for specific features and layers. These methods return a list of subitems in the feature or items in the layer.

To access specific model items, call the method **IpfcModelItemOwner.GetItemById()**. This method enables you to access the model item by identifier.

To access specific model items, call the method **IpfcModelItemOwner.GetItemByName()**. This method enables you to access the model item by name.

The property **IpfcFamColModelItem.RefItem** returns the dimension or feature used as a header for a family table.

The property **IpfcSelection.SelItem** returns the item selected interactively by the user.

## ModelItem Information

Methods and Properties Introduced:

- **IpfcModelItem.GetName()**
- **IpfcModelItem.SetName()**
- **IpfcModelItem.Id**
- **IpfcModelItem.Type**

Certain `IpfcModelItem`s also have a string name that can be changed at any time. The methods **GetName** and **SetName** access this name.

The property **Id** returns the unique integer identifier for the `IpfcModelItem`.

The **Type** property returns an enumeration object that indicates the model item type of the specified `IpfcModelItem`. See the section "[Solid Geometry Traversal](#)" for the list of possible model item types.

## Layer Objects

In the VB API, layers are instances of `IpfcModelItem`. The following sections describe how to get layer objects and the operations you can perform on them.

### Getting Layer Objects

Method Introduced:

- **IpfcModel.CreateLayer()**

The method **IpfcModel.CreateLayer()** returns a new layer with the name you specify.

See the section "[Getting ModelItem Objects](#)" for other methods that can return layer objects.

### Layer Operations

Methods and Properties Introduced:

- **IpfcLayer.Status**
- **IpfcLayer.ListItems()**
- **IpfcLayer.AddItem()**
- **IpfcLayer.RemoveItem()**
- **IpfcLayer.Delete()**

The property **IpfcLayer.Status** enables you to access the display status of a layer. The corresponding enumeration class is `EpfcDisplayStatus` and the possible values are `Normal`, `Displayed`, `Blank`, or `Hidden`.

Use the methods **IpfcLayer.ListItems()**, **IpfcLayer.AddItem()**, and **IpfcLayer.RemoveItem()** to control the contents of a layer.

The method **IpfcLayer.Delete()** removes the layer (but not the items it contains) from the model.

---

# Features

---

All Pro/ENGINEER solid models are made up of features. This section describes how to program on the feature level using the VB API.

## Topic

[Access to Features](#)

[Feature Information](#)

[Feature Operations](#)

[Feature Groups and Patterns](#)

[User Defined Features](#)

[Creating Features from UDFs](#)

## Access to Features

Methods and Properties Introduced:

- **IpfcFeature.ListChildren()**
- **IpfcFeature.ListParents()**
- **IpfcFeatureGroup.GroupLeader**
- **IpfcFeaturePattern.PatternLeader**
- **IpfcFeaturePattern.ListMembers()**
- **IpfcSolid.ListFailedFeatures()**
- **IpfcSolid.ListFeaturesByType()**
- **IpfcSolid.GetFeatureById()**

The methods **IpfcFeature.ListChildren()** and **IpfcFeature.ListParents()** return a sequence of features that contain all the children or parents of the specified feature.

To get the first feature in the specified group access the property **IpfcFeatureGroup.GroupLeader**.

The property **IpfcFeaturePattern.PatternLeader** and the method **IpfcFeaturePattern.ListMembers()** return features that make up the specified feature pattern. See [Feature Groups and Patterns](#) for more information on feature patterns.

The method **IpfcSolid.ListFailedFeatures()** returns a sequence that contains all the features that failed regeneration.

The method **IpfcSolid.ListFeaturesByType()** returns a sequence of features contained in the model. You can specify which type of feature to collect by passing in one of the `EpfcFeatureType` enumeration objects, or you can collect all features by passing **void null** as the type. If you list all features, the resulting sequence will include invisible features that Pro/ENGINEER creates internally. Use the method's *VisibleOnly* argument to exclude them.

The method **IpfcSolid.GetFeatureById()** returns the feature object with the corresponding integer identifier.

# Feature Information

Properties Introduced:

- **IpfcFeature.FeatType**
- **IpfcFeature.Status**
- **IpfcFeature.IsVisible**
- **IpfcFeature.IsReadOnly**
- **IpfcFeature.IsEmbedded**
- **IpfcFeature.Number**
- **IpfcFeature.FeatTypeName**
- **IpfcFeature.FeatSubType**
- **IpfcRoundFeat.IsAutoRoundMember**

The enumeration classes `EpfcFeatureType` and `EpfcFeatureStatus` provide information for a specified feature. The following properties specify this information:

- `IpfcFeature.FeatType`--Returns the type of a feature.
- `IpfcFeature.Status`--Returns whether the feature is suppressed, active, or failed regeneration.

The other properties that gather feature information include the following:

- `IpfcFeature.IsVisible`--Identifies whether the specified feature will be visible on the screen.
- `IpfcFeature.IsReadOnly`--Identifies whether the specified feature can be modified.
- `IpfcFeature.GetIsEmbedded`--Specifies whether the specified feature is an embedded datum.
- `IpfcFeature.Number`--Returns the feature regeneration number. This method returns void null if the feature is suppressed.

The property **IpfcFeature.FeatTypeName** returns a string representation of the feature type.

The property **IpfcFeature.FeatSubType** returns a string representation of the feature subtype, for example, "Extrude" for a protrusion feature.

The property **IpfcRoundFeat.IsAutoRoundMember** determines whether the specified round feature is a member of an Auto Round feature.

# Feature Operations

Methods and Properties Introduced:

- **IpfcSolid.ExecuteFeatureOps()**
- **IpfcFeature.CreateSuppressOp()**
- **IpfcSuppressOperation.Clip**

- **IpfcSuppressOperation.AllowGroupMembers**
- **IpfcSuppressOperation.AllowChildGroupMembers**
- **IpfcFeature.CreateDeleteOp()**
- **IpfcDeleteOperation.Clip**
- **IpfcDeleteOperation.AllowGroupMembers**
- **IpfcDeleteOperation.AllowChildGroupMembers**
- **IpfcDeleteOperation.KeepEmbeddedDatums**
- **IpfcFeature.CreateResumeOp()**
- **IpfcResumeOperation.WithParents**
- **IpfcFeature.CreateReorderBeforeOp()**
- **IpfcReorderBeforeOperation.BeforeFeat**
- **IpfcFeature.CreateReorderAfterOp()**
- **IpfcReorderAfterOperation.AfterFeat**

The method **IpfcSolid.ExecuteFeatureOps()** causes a sequence of feature operations to run in order. Feature operations include suppressing, resuming, reordering, and deleting features. The optional *IpfcRegenInstructions* argument specifies whether the user will be allowed to fix the model if a regeneration failure occurs.

You can create an operation that will delete, suppress, reorder, or resume certain features using the methods in the class **IpfcFeature**. Each created operation must be passed as a member of the **IpfcFeatureOperations** object to the method **IpfcSolid.ExecuteFeatureOps()**.

Some of the operations have specific options that you can modify to control the behavior of the operation:

- **Clip**--Specifies whether to delete or suppress all features after the selected feature. By default, this option is false. Use the properties **IpfcDeleteOperation.Clip** and **IpfcSuppressOperation.Clip** to modify this option.
- **AllowGroupMembers**--If this option is set to true and if the feature to be deleted or suppressed is a member of a group, then the feature will be deleted or suppressed out of the group. If this option is set to false, then the entire group containing the feature is deleted or suppressed. By default, this option is false. It can be set to true only if the option **Clip** is set to true. Use the properties **IpfcSuppressOperation.AllowGroupMembers** and **IpfcDeleteOperation.AllowGroupMembers** to modify this option.
- **AllowChildGroupMembers**--If this option is set to true and if the children of the feature to be deleted or suppressed are members of a group, then the children of the feature will be individually deleted or suppressed out of the group. If this option is set to false, then the entire group containing the feature and its children is deleted or suppressed. By default, this option is false. It can be set to true only if the options **Clip** and **AllowGroupMembers** are set to true. Use the properties **IpfcSuppressOperation.AllowChildGroupMembers** and **IpfcDeleteOperation.AllowChildGroupMembers** to modify this option.
- **KeepEmbeddedDatums**--Specifies whether to retain the embedded datums stored in a feature while deleting the feature. By default, this option is false. Use the property **IpfcDeleteOperation.KeepEmbeddedDatums** to modify this option.
- **WithParents**--Specifies whether to resume the parents of the selected feature. Use the property **IpfcResumeOperation.WithParents** to modify this option.

- BeforeFeat--Specifies the feature before which you want to reorder the features. Use the property `IpfcReorderBeforeOperation.BeforeFeat` to modify this option.
- AfterFeat--Specifies the feature after which you want to reorder the features. Use the property `IpfcReorderAfterOperation.AfterFeat` to modify this option.

## Feature Groups and Patterns

Patterns are treated as features in Pro/ENGINEER Wildfire. A feature type, `FEATTYPE_PATTERN_HEAD`, is used for the pattern header feature.

### Note:

The pattern header feature is not treated as a leader or a member of the pattern by the methods described in the following section.

Methods and Properties Introduced:

- **IpfcFeature.Group**
- **IpfcFeature.Pattern**
- **IpfcSolid.CreateLocalGroup()**
- **IpfcFeatureGroup.Pattern**
- **IpfcFeatureGroup.GroupLeader**
- **IpfcFeaturePattern.PatternLeader**
- **IpfcFeaturePattern.ListMembers()**
- **IpfcFeaturePattern.Delete()**

The property **IpfcFeature.Group** returns a handle to the local group that contains the specified feature.

To get the first feature in the specified group call the property **IpfcFeatureGroup.GroupLeader**.

The property **IpfcFeaturePattern.PatternLeader** and the method **IpfcFeaturePattern.ListMembers()** return features that make up the specified feature pattern.

The properties **IpfcFeature.Pattern** and **IpfcFeatureGroup.Pattern** return the `FeaturePattern` object that contains the corresponding `Feature` or `FeatureGroup`. Use the method **IpfcSolid.CreateLocalGroup()** to take a sequence of features and create a local group with the specified name. To delete a `FeaturePattern` object, call the method **IpfcFeaturePattern.Delete()**.

## Notes On Feature Groups

Feature groups have a group header feature, which shows up in the model information and feature list for the model. This feature will be inserted in the regeneration list to a position just before the first feature in the group.

The results of the header feature are as follows:

- Models that contain groups will get one extra feature in the regeneration list, of type `EFeatureType.FEATTYPE_GROUP_HEAD`. This affects the feature numbers of all subsequent features, including those in the group.
- Each group automatically contains the header feature in the list of features returned from `IpfcFeature.FeatureGroup`.

ListMembers.

- Each group automatically gets the group head feature as the leader. This is returned from `IpfcFeature.FeatureGroup.GetGroupLeader`.
- Each group pattern contains a series of groups, and each group in the pattern will be similarly constructed.

## User Defined Features

Groups in Pro/ENGINEER represent sets of contiguous features that act as a single feature for specific operations. Individual features are affected by most operations while some operations apply to an entire group:

- Suppress
- Delete
- Layers
- Patterning

User defined Features (UDFs) are groups of features that are stored in a file. When a UDF is placed in a new model the created features are automatically assigned to a group. A local group is a set of features that have been specifically assigned to a group to make modifications and patterning easier.

### Note:

All methods in this section can be used for UDFs and local groups.

## Read Access to Groups and User Defined Features

Methods and Properties Introduced:

- **IpfcFeatureGroup.UDFName**
- **IpfcFeatureGroup.UDFInstanceName**
- **IpfcFeatureGroup.ListUDFDimensions()**
- **IpfcUDFDimension.UDFDimensionName**

User defined features (UDF's) are groups of features that can be stored in a file and added to a new model. A local group is similar to a UDF except it is available only in the model in which it was created.

The property **IpfcFeatureGroup.UDFName** provides the name of the group for the specified group instance. A particular group definition can be used more than once in a particular model.

If the group is a family table instance, the property **IpfcFeatureGroup.UDFInstanceName** supplies the instance name.

The method **IpfcFeatureGroup.ListUDFDimensions()** traverses the dimensions that belong to the UDF. These dimensions correspond to the dimensions specified as variables when the UDF was created. Dimensions of the original features that were not variables in the UDF are not included unless the UDF was placed using the **Independent** option.

The property **IpfcUDFDimension.UDFDimensionName** provides access to the dimension name specified when the UDF was created, and not the name of the dimension in the current model. This name is required to place the UDF programmatically using the method **IpfcSolid.CreateUDFGroup()**.

## Creating Features from UDFs

Method Introduced:

- **IpfcSolid.CreateUDFGroup()**

The method **IpfcSolid.CreateUDFGroup()** is used to create new features by retrieving and applying the contents of an existing UDF file. It is equivalent to the Pro/ENGINEER command **Feature, Create, User Defined**.

To understand the following explanation of this method, you must have a good knowledge and understanding of the use of UDF's in Pro/ENGINEER. PTC recommends that you read about UDF's in the Pro/ENGINEER on-line help, and practice defining and using UDF's in Pro/ENGINEER before you attempt to use this method.

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 the VB API, you can provide some or all of this information programmatically by filling several compact data classes that are inputs to the method **IpfcSolid.CreateUDFGroup()**.

During the call to **IpfcSolid.CreateUDFGroup()**, Pro/ENGINEER prompts you for the following:

- Information required by the UDF that was not provided in the input data structures.
- 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 UDF's programmatically to provide automatic synthesis of model geometry, you can also use **IpfcSolid.CreateUDFGroup()** to create UDF's 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 UDFs

Creating a UDF requires the following information:

- Name--The name of the UDF you are creating and the instance name if applicable.
- Dependency--Specify if the UDF is independent of the UDF definition or is modified by the changers made to it.
- Scale--How to scale the UDF relative to the placement model.
- Variable Dimension--The new values of the variables 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 that the UDF needs in order to relate the features it contains to the existing models features. The elements correspond to the picks that Pro/ENGINEER prompts you for when you create a UDF interactively using the prompts defined when the UDF was created. You cannot select an embedded datum as the UDF reference.
- Parts Intersection--When a UDF that is being created in an assembly contains features that modify the existing geometry you must define which parts are affected or intersected. You also need to know at what level in an assembly each intersection is going to be visible.
- Orientations--When a UDF contains a feature with a direction that is defined in respect to a datum plane Pro/ENGINEER must know what direction the new feature will point to. When you create such a UDF interactively Pro/ENGINEER prompt you for this information with a flip arrow.
- Quadrants--When a UDF contains a linearly placed feature that references two datum planes to define it's location in the new model Pro/ENGINEER prompts you to pick the location of the new feature. This is determined by which side of each datum plane the feature must lie. This selection is referred to as the quadrant because there are four possible combinations for each linearly placed feature.

To pass all the above values to Pro/ENGINEER, the VB API uses a special class that prepares and sets all the options and passes them to Pro/ENGINEER.

## Creating Interactively Defined UDFs

Method Introduced:

- **CCpfcUDFPromptCreateInstructions.Create()**

This static method is used to create an instructions object that can be used to prompt a user for the required values that will create a UDF interactively.

## Creating a Custom UDF

Method Introduced:

- **CCpfcUDFCustomCreateInstructions.Create()**

This method creates a `UDFCustomCreateInstructions` object with a specified name. To set the UDF creation parameters programmatically you must modify this object as described below. The members of this class relate closely to the prompts Pro/ENGINEER gives you when you create a UDF interactively. PTC recommends that you experiment with creating the UDF interactively using Pro/ENGINEER before you write the the VB API code to fill the structure.

## Setting the Family Table Instance Name

Property Introduced:

- **IpfcUDFCustomCreateInstructions.InstanceName**

If the UDF contains a family table, this field can be used to select the instance in the table. If the UDF does not contain a family table, or if the generic instance is to be selected, the do not set the string.

## Setting Dependency Type

Property Introduced:

- **IpfcUDFCustomCreateInstructions.DependencyType**

The `EpfcUDFDependencyType` object represents the dependency type of the UDF. The choices correspond to the choices available when you create a UDF interactively. This enumerated type takes the following values:

- `EpfcUDFDEP_INDEPENDENT`
- `EpfcUDFDEP_DRIVEN`

**Note:**

`EpfcUDFDEP_INDEPENDENT` is the default value, if this option is not set.

## Setting Scale and Scale Type

Properties Introduced:

- **IpfcUDFCustomCreateInstructions.ScaleType**

- **IpfcUDFCustomCreateInstructions.Scale**

The property *ScaleType* specifies the length units of the UDF in the form of the `EpfcUDFScaleType` object. This enumerated type takes the following values:

- `EpfcUDFSCALE_SAME_SIZE`
- `EpfcUDFSCALE_SAME_DIMS`
- `EpfcUDFSCALE_CUSTOM`
- `EpfcUDFSCALE_nil`

**Note:**

The default value is UDFSCALE\_SAME\_SIZE if this option is not set.

The property Scale specifies the scale factor. If the *ScaleType* is set to EpcfUDFSCALE\_CUSTOM, the property Scale assigns the user defined scale factor. Otherwise, this attribute is ignored.

## Setting the Appearance of the Non UDF Dimensions

Properties Introduced:

- **IpcfUDFCustomCreateInstructions.DimDisplayType**

The EpcfUDFDimensionDisplayType object sets the options in Pro/ENGINEER for determining the appearance in the model of UDF dimensions and pattern parameters that were not variable in the UDF, and therefore cannot be modified in the model. This enumerated type takes the following values:

- EpcfUDFDISPLAY\_NORMAL
- EpcfUDFDISPLAY\_READ\_ONLY
- EpcfUDFDISPLAY\_BLANK

**Note:**

The default value is EpcfUDFDISPLAY\_NORMAL if this option is not set.

## Setting the Variable Dimensions and Parameters

Methods and Properties Introduced:

- **IpcfUDFCustomCreateInstructions.VariantValues**
- **CCpcfUDFVariantDimension.Create()**
- **CCpcfUDFVariantPatternParam.Create()**

IpcfUDFVariantValues class represents an array of variable dimensions and pattern parameters.

**CCpcfUDFVariantDimension.Create()** is a static method creating a IpcfUDFVariantDimension. It accepts the following parameters:

- Name--The symbol that the dimension had when the UDF was originally defined not the prompt that the UDF uses when it is created interactively. To make this name easy to remember, before you define the UDF that you plan to create with the VB API, you should modify the symbols of all the dimensions that you want to select to be variable. If you get the name wrong, IpcfSolid.CreateUDFGroup will not recognize the dimension and prompts the user for the value in the usual way does not modify the value.
- DimensionValue--The new value.

If you do not remember the name, you can find it by creating the UDF interactively in a test model, then using the **IpcfFeatureGroup.ListUDFDimensions()** and **IpcfUDFDimension.UDFDimensionName** to find out the name.

**CCpcfUDFVariantPatternParam.Create()** is a static method which creates a IpcfUDFVariantPatternParam. It accepts the following parameters:

- name--The string name that the pattern parameter had when the UDF was originally defined
- patternparam--The new value.

After the IpcfUDFVariantValues object has been compiled, use **IpcfUDFCustomCreateInstructions**.

**VariantValues** to add the variable dimensions and parameters to the instructions.

## Setting the User Defined References

Methods and Properties Introduced:

- **CCpfcUDFReference.Create()**
- **IpfcUDFReference.IsExternal**
- **IpfcUDFReference.ReferenceItem**
- **IpfcUDFCustomCreateInstructions.References**

The method **CCpfcUDFReference.Create()** is a static method creating a `UDFReference` object. It accepts the following parameters:

- **PromptForReference**--The prompt defined for this reference when the UDF was originally set up. It indicates which reference this structure is providing. If you get the prompt wrong, `IpfcSolid.CreateUDFGroup()` will not recognize it and prompts the user for the reference in the usual way.
- **ReferenceItem**--Specifies the `IpfcSelection` object representing the referenced element. You can set `Selection` programmatically or prompt the user for a selection separately. You cannot set an embedded datum as the UDF reference.

There are two types of reference:

- **Internal**--The referenced element belongs directly to the model that will contain the UDF. For an assembly, this means that the element belongs to the top level.
- **External**--The referenced element belongs to an assembly member other than the placement member.

To set the reference type, use the property **IpfcUDFReference.IsExternal**.

To set the item to be used for reference, use the property **IpfcUDFReference.ReferenceItem**.

After the `UDFReferences` object has been set, use **IpfcUDFCustomCreateInstructions.References** to add the program-defined references.

## Setting the Assembly Intersections

Methods and Properties Introduced:

- **CCpfcUDFAssemblyIntersection.Create()**
- **IpfcUDFAssemblyIntersection.InstanceNames**
- **IpfcUDFCustomCreateInstructions.Intersections**

**CCpfcUDFAssemblyIntersection.Create()** is a static method creating a `IpfcUDFReference` object. It accepts the following parameters:

- **ComponentPath**--Is an `intseq` type object representing the component path of the part to be intersected.
- **Visibility level**--The number that corresponds to the visibility level of the intersected part in the assembly. If the number is equal to the length of the component path the feature is visible in the part that it intersects. If `Visibility level` is 0, the feature is visible at the level of the assembly containing the UDF.

**IpfcUDFAssemblyIntersection.InstanceNames** sets an array of names for the new instances of parts created to

represent the intersection geometry. This property accepts the following parameters:

- instance names--is a `com.ptc.cipjava.stringseq` type object representing the array of new instance names.

After the `IpfcUDFAssemblyIntersections` object has been set, use **IpfcUDFCustomCreateInstructions.Intersections** to add the assembly intersections.

## Setting Orientations

Properties Introduced:

- **IpfcUDFCustomCreateInstructions.Orientations**

`IpfcUDFOrientations` class represents an array of orientations that provide the answers to Pro/ENGINEER prompts that use a flip arrow. Each term is a `EpfcUDFOrientation` object that takes the following values:

- `EpfcUDFORIENT_INTERACTIVE`--Prompt for the orientation using a flip arrow.
- `EpfcUDFORIENT_NO_FLIP`--Accept the default flip orientation.
- `EpfcUDFORIENT_FLIP`--Invert the orientation from the default orientation.

The order of orientations should correspond to the order in which Pro/ENGINEER prompts for them when the UDF is created interactively. If you do not provide an orientation that Pro/ENGINEER needs, it uses the default value `NO_FLIP`.

After the `IpfcUDFOrientations` object has been set use **IpfcUDFCustomCreateInstructions.Orientations** to add the orientations.

## Setting Quadrants

Property Introduced:

- **IpfcUDFCustomCreateInstructions.Quadrants**

The property **IpfcUDFCustomCreateInstructions.Quadrants** sets an array of points, which provide the X, Y, and Z coordinates that correspond to the picks answering the Pro/ENGINEER prompts for the feature positions. The order of quadrants should correspond to the order in which Pro/ENGINEER prompts for them when the UDF is created interactively.

## Setting the External References

Property Introduced:

- **IpfcUDFCustomCreateInstructions.ExtReferences**

The property **IpfcUDFCustomCreateInstructions.ExtReferences** sets an external reference assembly to be used when placing the UDF. This will be required when placing the UDF in the component using references outside of that component. References could be to the top level assembly of another component.

### Example Code

The example code places copies of a node UDF at a particular coordinate system location in a part. The node UDF is a spherical cut centered at the coordinate system whose diameter is driven by the 'diam' argument to the method. The method returns the **FeatureGroup** object created, or null if an error occurred.

```

Public Function createNodeUDFInPart(ByVal placementModel As IpfcSolid, _
                                   ByVal csysName As String, _
                                   ByVal diameter As Double) _
    As IpfcFeatureGroup

    Dim csys As IpfcCoordSystem = Nothing
    Dim cSystems As IpfcModelItems
    Dim i As Integer
    Dim udfInstructions As IpfcUDFCustomCreateInstructions
    Dim csysSelection As IpfcSelection
    Dim csysReference As IpfcUDFReference
    Dim references As CpfcUDFReferences
    Dim variantDims As IpfcUDFVariantDimension
    Dim variantVals As IpfcUDFVariantValues
    Dim group As IpfcFeatureGroup

    Try

        cSystems =
            placementModel.ListItems(EpfcModelItemType.EpfcITEM_COORD_SYS)

        For i = 0 To cSystems.Count - 1
            If (cSystems.Item(i).GetName.ToString = csysName) Then
                csys = cSystems.Item(i)
                Exit For
            End If
        Next

        If csys Is Nothing Then
            Throw New Exception("Coordinate System not found in
                                current Solid")
        End If

' =====
' Instructions for UDF creation
' =====
        udfInstructions =
            (New CCpfcUDFCustomCreateInstructions).Create("node")

' =====
' Make non variant dimensions blank to disable their display
' =====
        udfInstructions.DimDisplayType =
            EpfcUDFDimensionDisplayType.EpfcUDFDISPLAY_BLANK

' =====
' Initialize the UDF reference and assign it to the instructions.
' The string argument is the reference prompt for the particular
' reference.
' =====
        csysSelection =
            (New CMpfcSelect).CreateModelItemSelection(csys, Nothing)

        csysReference = (New CCpfcUDFReference).Create("REF_CSYS",
                                                    csysSelection)

        references = New CpfcUDFReferences
        references.Set(0, csysReference)

        udfInstructions.References = references
    
```

```

' =====
' Initialize the variant dimension and assign it to the instructions.
' The string argument is the dimension symbol for the variant dimension.
' =====
        variantDims = (New CCpfcUDFVariantDimension).Create("d11",
                                                             diameter)

        variantVals = New CpfcUDFVariantValues
        variantVals.Set(0, variantDims)

        udfInstructions.VariantValues = variantVals
' =====
' We need the placement model for the UDF for the call to
' CreateUDFGroup(). If you were placing the UDF in a model other than
' the owner of the coordinate system, the placement would need to be
' provided separately.
' =====

        group = placementModel.CreateUDFGroup(udfInstructions)

        Return group

Catch ex As Exception
    MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
    Return Nothing
End Try

End Function

```

#### Example Code

This function places copies of a hole UDF at a particular location in an assembly. The hole is embedded in a surface of one of the assembly's components, and placed a particular location away from two normal datum planes (the default value for the dimension is used for this example). The UDF creation requires a quadrant determining the location for the UDF (since it could be one of four areas) and intersection instructions for the assembly members (this example makes the hole visible down to the part level). The method returns the FeatureGroup object created.

```

Public Function createHoleUDFInAssembly _
    (ByVal sideRefSurfaceIds() As Integer, _
     ByVal referencePath As IpfcComponentPath, _
     ByVal placementSurfaceId As Integer, _
     ByVal scale As Double, _
     ByVal quadrant As IpfcPoint3D) As IpfcFeatureGroup

    Dim udfInstructions As IpfcUDFCustomCreateInstructions
    Dim referenceModel As IpfcSolid
    Dim placementSurface As IpfcModelItem
    Dim surfaceSelection As IpfcSelection
    Dim datumSelection(2) As IpfcSelection
    Dim references As CpfcUDFReferences
    Dim reference1 As IpfcUDFReference
    Dim reference2 As IpfcUDFReference

```

```

Dim reference3 As IpfcUDFReference
Dim assembly As IpfcSolid
Dim i As Integer
Dim sideReference As IpfcModelItem
Dim quadrants As CpfcPoint3Ds
Dim intersections As CpfcUDFAssemblyIntersections
Dim leafs As IpfcComponentPath()
Dim ids As Cintseq
Dim intersection As IpfcUDFAssemblyIntersection
Dim group As IpfcFeatureGroup = Nothing

Try

    If Not (sideRefSurfaceIds.Length = 2) Then
        Throw New Exception("Improper array size. Both side references must
be given.")
    End If

    udfInstructions = (New CCpfcUDFCustomCreateInstructions).Create
("hole_quadrant")

    If scale = 0 Then
        udfInstructions.ScaleType = EpfcUDFScaleType.EpfcUDFSCALE_SAME_SIZE
    Else
        udfInstructions.ScaleType = EpfcUDFScaleType.EpfcUDFSCALE_CUSTOM
        udfInstructions.Scale = scale
    End If

'=====
'The first UDF reference is a surface from a component model in the
'assembly. This requires using the ComponentPath to initialize the
'Selection, and setting the IsExternal flag to true.
'=====
    referenceModel = referencePath.Leaf

    placementSurface = referenceModel.GetItemById _
        (EpfcModelItemType.EpfcITEM_SURFACE,
placementSurfaceId)

    If Not (TypeOf placementSurface Is IpfcSurface) Then
        Throw New Exception("Input Surface Id " + placementSurfaceId.
ToString _ + " is not surface")
    End If

    surfaceSelection = (New CMpfcSelect).CreateModelItemSelection _
(placementSurface, referencePath)

    references = New CpfcUDFReferences()
    referencel = (New CCpfcUDFReference).Create _
        ("embedding surface?", surfaceSelection)
    referencel.IsExternal = True

    references.Set(0, referencel)

'=====
'The next two UDF references are expected to be Datum Plane features in
'the assembly. The reference is constructed using the Surface object
'contained in the Datum plane feature.
'=====
    assembly = referencePath.Root

```

```

For i = 0 To 1
    sideReference = assembly.GetItemById _
                    (EpfModelItemType.EpfITEM_SURFACE,
sideRefSurfaceIds(i))

    datumSelection(i) = (New CmpfcSelect).CreateModelItemSelection _
                        (sideReference, Nothing)
Next

reference2 = (New CCpfcUDFReference).Create _
            ("right surface", datumSelection(0))
references.Set(1, reference2)

reference3 = (New CCpfcUDFReference).Create _
            ("front surface", datumSelection(1))
references.Set(2, reference3)

udfInstructions.References = references
'=====
'If the UDF and the placement both use two normal datum planes as
'dimensioned references, Pro/ENGINEER prompts the user for a pick to
'define the quadrant where the UDF will be placed.
'=====
    quadrants = New CpfcPoint3Ds
    quadrants.Set(0, quadrant)

    udfInstructions.Quadrants = quadrants
'=====
'This hole UDF should be visible down to the component part level. To
'direct this, the UDFAssemblyIntersection should be created with the
'component ids, and the visibility level argument equal to the number
'of component levels. Alternatively, the visibility level could be 0
'to force the UDF to appear in the assembly only
'=====
    intersections = New CpfcUDFAssemblyIntersections

    leafs = AssemblyUtilities.listEachLeafComponent(assembly)

    For i = 0 To leafs.Length - 1
        If Not leafs(i) Is Nothing Then
            ids = leafs(i).ComponentIds
            intersection = (New CCpfcUDFAssemblyIntersection).Create(ids,
ids.Count)
            intersections.Set(i, intersection)
        End If
    Next

    udfInstructions.Intersections = intersections
'=====
'Create the assembly group
'=====
    group = assembly.CreateUDFGroup(udfInstructions)

    Return group

Catch ex As Exception
    MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
    Return Nothing
End Try

```

```

End Function
'=====
'Class      :   AssemblyUtilities
'Purpose    :   This Class provides utility functions for assembly.
'=====
Private Class AssemblyUtilities
Private Shared asm As IpfcAssembly
Private Shared pathArray As ArrayList
'=====
'Function   :   listEachLeafComponent
'Purpose    :   This function returns an array of all ComponentPath's
'            :   to all component parts ('leaves') in an assembly.
'=====
Public Shared Function listEachLeafComponent(ByVal assembly As IpfcAssembly)
-
As IpfcComponentPath()

Dim startLevel As New Cintseq
Dim i As Integer

asm = assembly
pathArray = New ArrayList

listSubAssemblyComponent(startLevel)
Dim compPaths(pathArray.Count) As IpfcComponentPath

For i = 0 To pathArray.Count - 1
    compPaths(i) = pathArray.Item(i)
Next

Return (compPaths)

End Function
'=====
'Function   :   listEachLeafComponent
'Purpose    :   This function This method is used to recursively visit
'            :   all levels of the assembly structure.
'=====
Private Shared Sub listSubAssemblyComponent(ByVal currentLevel As Cintseq)

Dim currentComponent As IpfcSolid
Dim currentPath As IpfcComponentPath = Nothing
Dim level As Integer
Dim subComponents As IpfcFeatures
Dim i, id As Integer
Dim componentFeat As IpfcFeature

level = currentLevel.Count
'=====
'Special case, level is 0 for the top level assembly.
'=====
If (level > 0) Then
    currentPath = (New CMpfcAssembly).CreateComponentPath(asm,
currentLevel)
    currentComponent = currentPath.Leaf
Else
    currentComponent = asm
End If

If (currentComponent.Type = EpfcModelType.EpfcMDL_PART) And (level > 0)

```

```

Then
    pathArray.Add(currentPath)
Else
'=====
'Find all component features in the current component object.
'Visit each (adjusting the component id paths accordingly).
'=====

    subComponents = currentComponent.ListFeaturesByType _
        (True, EpfcFeatureType.EpfcFEATTYPE_COMPONENT)

    For i = 0 To subComponents.Count - 1
        componentFeat = subComponents.Item(i)
        id = componentFeat.Id

        currentLevel.Set(level, id)

        listSubAssemblyComponent(currentLevel)
    Next
End If
'=====
'Clean up current level of component ids before returning up one level.
'=====
    If Not level = 0 Then
        currentLevel.Remove(level - 1, level)
    End If
    Return

    End Sub

End Class

End Class

```

---

# Geometry Evaluation

---

This section describes geometry representation and discusses how to evaluate geometry using the VB API.

## Topic

[Geometry Traversal](#)

[Curves and Edges](#)

[Contours](#)

[Surfaces](#)

[Axes, Coordinate Systems, and Points](#)

[Interference](#)

## Geometry Traversal

### Note:

- A simple rectangular face has one contour and four edges.
- A contour will traverse a boundary so 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. The original surface of the part is represented as one surface with a cut through it.

## Geometry Terms

Following are definitions for some geometric terms:

- 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 and 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 and to two contours. If the edge is the external boundary of the datum surface it will belong to that surface alone and to a single contour.

## Traversing the Geometry of a Solid Block

Methods Introduced:

- `IpfcModelItemOwner.ListItems()`

- **IpfcSurface.ListContours()**
- **IpfcContour.ListElements()**

To traverse the geometry, follow these steps:

1. Starting at the top-level model, use `IpfcModelItemOwner.ListItems()` with an argument of `ModelItemType.ITEM_SURFACE`.
2. Use **IpfcSurface.ListContours()** to list the contours contained in a specified surface.
3. Use **IpfcContour.ListElements()** to list the edges contained in the contour.

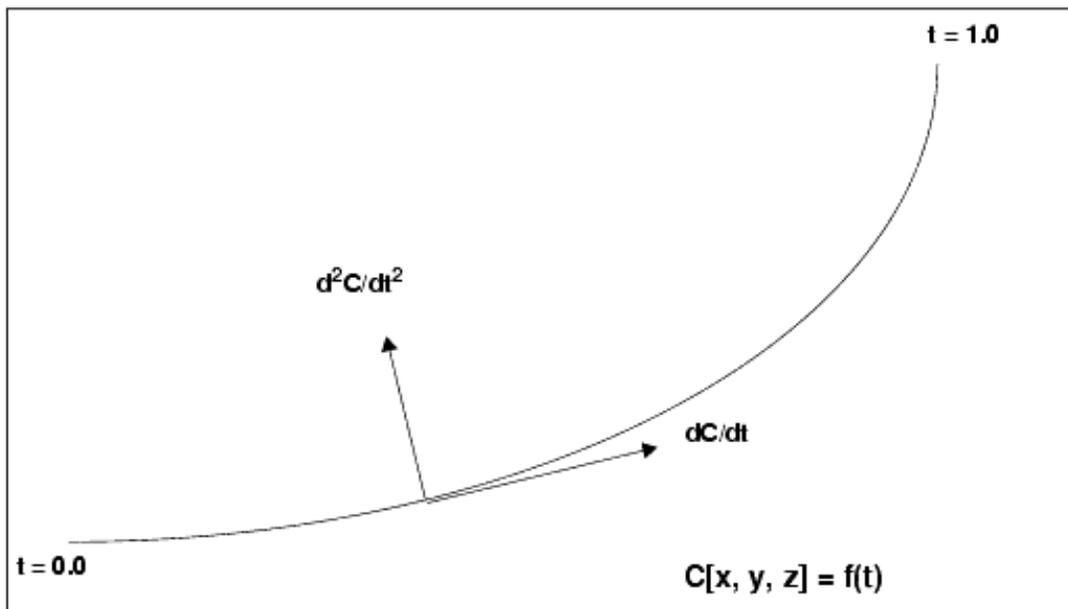
## Curves and Edges

Datum curves, surface edges, and solid edges are represented in the same way in the VB API. You can get edges through geometry traversal or get a list of edges using the methods presented in section "[ModelItem](#)".

### The t Parameter

The geometry of each edge or curve is represented as a set of three parametric equations that represent the values of  $x$ ,  $y$ , and  $z$  as functions of an independent parameter,  $t$ . The  $t$  parameter varies from 0.0 at the start of the curve to 1.0 at the end of it.

The following figure illustrates curve and edge parameterization.



### Curve and Edge Types

Solid edges and datum curves can be any of the following types:

- LINE--A straight line represented by the classinterface `IpfcLine`.
- ARC--A circular curve represented by the classinterface `IpfcArc`.

- **SPLINE**--A nonuniform cubic spline, represented by the classinterface `IpfcSpline`.
- **B-SPLINE**--A nonuniform rational B-spline curve or edge, represented by the classinterface `IpfcBSpline`.
- **COMPOSITE CURVE**--A combination of two or more curves, represented by the classinterface `IpfcCompositeCurve`. This is used for datum curves only.

See the section, [Geometry Representations](#), for the parameterization of each curve type. To determine what type of curve a `IpfcEdge` or `IpfcCurve` object represents, use the **instanceof** operator.

Because each curve class inherits from `IpfcGeomCurve`, you can use all the evaluation methods in `IpfcGeomCurve` on any edge or curve.

The following curve types are not used in solid geometry and are reserved for future expansion:

- **CIRCLE** (Circle)
- **ELLIPSE** (Ellipse)
- **POLYGON** (Polygon)
- **ARROW** (Arrow)
- **TEXT** (Text)

## Evaluation of Curves and Edges

Methods Introduced:

- **`IpfcGeomCurve.Eval3DData()`**
- **`IpfcGeomCurve.EvalFromLength()`**
- **`IpfcGeomCurve.EvalParameter()`**
- **`IpfcGeomCurve.EvalLength()`**
- **`IpfcGeomCurve.EvalLengthBetween()`**

The methods in `IpfcGeomCurve` provide information about any curve or edge.

The method **`IpfcGeomCurve.Eval3DData()`** returns a `IpfcCurveXYZData` object with information on the point represented by the input parameter  $t$ . The method **`IpfcGeomCurve.EvalFromLength()`** returns a similar object with information on the point that is a specified distance from the starting point.

The method **`IpfcGeomCurve.EvalParameter()`** returns the  $t$  parameter that represents the input `IpfcPoint3D` object.

Both **`IpfcGeomCurve.EvalLength()`** and **`IpfcGeomCurve.EvalLengthBetween()`** return numerical values for the length of the curve or edge.

## Solid Edge Geometry

Methods and Properties Introduced:

- **`IpfcEdge.Surface1`**
- **`IpfcEdge.Surface2`**

- **IpfcEdge.Edge1**
- **IpfcEdge.Edge2**
- **IpfcEdge.EvalUV()**
- **IpfcEdge.GetDirection()**

**Note:**

The methods in the interface **IpfcEdge** provide information only for solid or surface edges.

The properties **IpfcEdge.Surface1** and **IpfcEdge.Surface2** return the surfaces bounded by this edge. The properties **IpfcEdge.Edge1** and **IpfcEdge.Edge2** return the next edges in the two contours that contain this edge.

The method **IpfcEdge.EvalUV()** evaluates geometry information based on the UV parameters of one of the bounding surfaces.

The method **IpfcEdge.GetDirection()** returns a positive 1 if the edge is parameterized in the same direction as the containing contour, and -1 if the edge is parameterized opposite to the containing contour.

## Curve Descriptors

A curve descriptor is a data object that describes the geometry of a curve or edge. A curve descriptor describes the geometry of a curve without being a part of a specific model.

Methods Introduced:

- **IpfcGeomCurve.GetCurveDescriptor()**
- **IpfcGeomCurve.GetNURBSRepresentation()**

**Note:**

To get geometric information for an edge, access the **IpfcCurveDescriptor** object for one edge using **IpfcGetCurveDescriptor**.

The method **IpfcGeomCurve.GetCurveDescriptor()** returns a curve's geometry as a data object.

The method **IpfcGeomCurve.GetNURBSRepresentation()** returns a Non-Uniform Rational B-Spline Representation of a curve.

## Contours

Methods and Properties Introduced:

- **IpfcSurface.ListContours()**
- **IpfcContour.InternalTraversal**
- **IpfcContour.FindContainingContour()**

- **IpfcContour.EvalArea()**
- **IpfcContour.EvalOutline()**
- **IpfcContour.VerifyUV()**

Contours are a series of edges that completely bound a surface. A contour is *not* a `IpfcModelItem`. You cannot get contours using the methods that get different types of `ModelItem`. Use the method **IpfcSurface.ListContours()** to get contours from their containing surfaces.

The property **IpfcContour.InternalTraversal** returns a `EpfContourTraversal` enumerated type that identifies whether a given contour is on the outside or inside of a containing surface.

Use the method **IpfcContour.FindContainingContour()** to find the contour that entirely encloses the specified contour.

The method **IpfcContour.EvalArea()** provides the area enclosed by the contour.

The method **IpfcContour.EvalOutline()** returns the points that make up the bounding rectangle of the contour.

Use the method **IpfcContour.VerifyUV()** to determine whether the given `IpfcUVParams` argument lies inside the contour, on the boundary, or outside the contour.

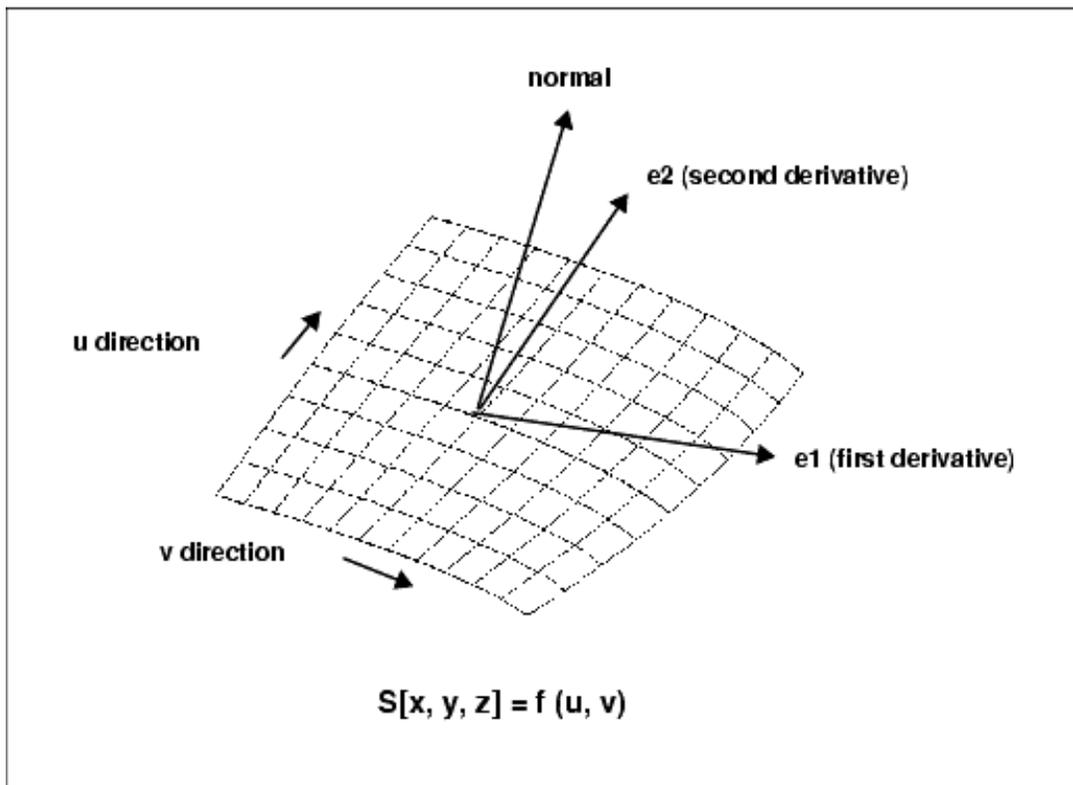
## Surfaces

Using the VB API you access datum and solid surfaces in the same way.

## UV Parameterization

A surface in Pro/ENGINEER is described as a series of parametric equations where two parameters,  $u$  and  $v$ , determine the  $x$ ,  $y$ , and  $z$  coordinates. Unlike the edge parameter,  $t$ , these parameters need not start at 0.0, nor are they limited to 1.0.

The figure on the following page illustrates surface parameterization.



## Surface Types

Surfaces within Pro/ENGINEER can be any of the following types:

- PLANE--A planar surface represented by the classinterface `IpfcPlane`.
- CYLINDER--A cylindrical surface represented by the classinterface `IpfcCylinder`.
- CONE--A conic surface region represented by the classinterface `IpfcCone`.
- TORUS--A toroidal surface region represented by the classinterface `IpfcTorus`.
- REVOLVED SURFACE--Generated by revolving a curve about an axis. This is represented by the classinterface `IpfcRevSurface`.
- RULED SURFACE--Generated by interpolating linearly between two curve entities. This is represented by the classinterface `IpfcRuledSurface`.
- TABULATED CYLINDER--Generated by extruding a curve linearly. This is represented by the classinterface `IpfcTabulatedCylinder`.
- QUILT--A combination of two or more surfaces. This is represented by the classinterface `IpfcQuilt`.

### Note:

This is used only for datum surfaces.

- COONS PATCH--A coons patch is used to blend surfaces together. It is represented by the classinterface `IpfcCoonsPatch`
- FILLET SURFACE--A filleted surface is found where a round or fillet is placed on a curved edge or an edge with a non-constant arc radii. On a straight edge a cylinder is used to represent a fillet. This is represented by the classinterface `IpfcFilletedSurface`.
- SPLINE SURFACE-- A nonuniform bicubic spline surface that passes through a grid with tangent vectors given at each point. This is represented by the classinterface `IpfcSplineSurface`.
- NURBS SURFACE--A NURBS surface is defined by basic functions (in  $u$  and  $v$ ), expandable arrays of knots, weights, and control points. This is represented by the classinterface `IpfcNURBSSurface`.

- **CYLINDRICAL SPLINE SURFACE**-- A cylindrical spline surface is a nonuniform bicubic spline surface that passes through a grid with tangent vectors given at each point. This is represented by the class `IpfcCylindricalSplineSurface`.

To determine which type of surface a `IpfcSurface` object represents, access the surface type using `IpfcGetSurfaceType`.

## Surface Information

Methods Introduced:

- `IpfcSurface.GetSurfaceType()`
- `IpfcSurface.GetXYZExtents()`
- `IpfcSurface.GetUVExtents()`
- `IpfcSurface.GetOrientation()`

## Evaluation of Surfaces

Surface methods allow you to use multiple surface information to calculate, evaluate, determine, and examine surface functions and problems.

Methods and Properties Introduced:

- `IpfcSurface.OwnerQuilt`
- `IpfcSurface.EvalClosestPoint()`
- `IpfcSurface.EvalClosestPointOnSurface()`
- `IpfcSurface.Eval3DData()`
- `IpfcSurface.EvalParameters()`
- `IpfcSurface.EvalArea()`
- `IpfcSurface.EvalDiameter()`
- `IpfcSurface.EvalPrincipalCurv()`
- `IpfcSurface.VerifyUV()`
- `IpfcSurface.EvalMaximum()`
- `IpfcSurface.EvalMinimum()`
- `IpfcSurface.ListSameSurfaces()`

The property `IpfcSurface.OwnerQuilt` returns the `Quilt` object that contains the datum surface.

The method **IpfcSurface.EvalClosestPoint()** projects a three-dimensional point onto the surface. Use the method **IpfcSurface.EvalClosestPointOnSurface()** to determine whether the specified three-dimensional point is on the surface, within the accuracy of the part. If it is, the method returns the point that is exactly on the surface. Otherwise the method returns null.

The method **IpfcSurface.Eval3DData()** returns a `IpfcSurfXYZData` object that contains information about the surface at the specified  $u$  and  $v$  parameters. The method **IpfcSurface.EvalParameters()** returns the  $u$  and  $v$  parameters that correspond to the specified three-dimensional point.

The method **IpfcSurface.EvalArea()** returns the area of the surface, whereas **IpfcSurface.EvalDiameter()** returns the diameter of the surface. If the diameter varies the optional `IpfcUVParams` argument identifies where the diameter should be evaluated.

The method **IpfcSurface.EvalPrincipalCurv()** returns a `IpfcCurvatureData` object with information regarding the curvature of the surface at the specified  $u$  and  $v$  parameters.

Use the method **IpfcSurface.VerifyUV()** to determine whether the `IpfcUVParams` are actually within the boundary of the surface.

The methods **IpfcSurface.EvalMaximum()** and **IpfcSurface.EvalMinimum()** return the three-dimensional point on the surface that is the furthest in the direction of (or away from) the specified vector.

The method **IpfcSurface.ListSameSurfaces()** identifies other surfaces that are tangent and connect to the given surface.

## Surface Descriptors

A surface descriptor is a data object that describes the shape and geometry of a specified surface. A surface descriptor allows you to describe a surface in 3D without an owner ID.

Methods Introduced:

- **IpfcSurface.GetSurfaceDescriptor()**
- **IpfcSurface.GetNURBSRepresentation()**

The method **IpfcSurface.GetSurfaceDescriptor()** returns a surface's geometry as a data object.

The method **IpfcSurface.GetNURBSRepresentation()** returns a Non-Uniform Rational B-Spline Representation of a surface.

## Axes, Coordinate Systems, and Points

Coordinate axes, datum points, and coordinate systems are all model items. Use the methods that return `IpfcModelItems` to get one of these geometry objects. Refer to section "[ModelItem](#)" for additional information.

## Evaluation of ModelItems

Properties Introduced:

- **IpfcAxis.Surf**

- **IpfcCoordSystem.CoordSys**
- **IpfcPoint.Point**

The **IpfcAxis.Surf** returns the revolved surface that uses the axis.

The property **IpfcCoordSystem.CoordSys** returns the `Transform3D` object (which includes the origin and x-, y-, and z- axes) that defines the coordinate system.

The property **IpfcPoint.Point** returns the xyz coordinates of the datum point.

## Interference

Pro/ENGINEER assemblies can contain interferences between components when constraint by certain rules defined by the user. The `pfcInterference` module allows the user to detect and analyze any interferences within the assembly. The analysis of this functionality should be looked at from two standpoints: global and selection based analysis.

Methods and Properties Introduced:

- **CMpfcInterference.CreateGlobalEvaluator()**
- **IpfcGlobalEvaluator.ComputeGlobalInterference()**
- **IpfcGlobalEvaluator.Assem**
- **IpfcGlobalEvaluator.Assem**
- **IpfcGlobalInterference.Volume**
- **IpfcGlobalInterference.SelParts**

To compute all the interferences within an Assembly one has to call **CMpfcInterference.CreateGlobalEvaluator()** with a `IpfcAssembly` object as an argument. This call returns a `IpfcGlobalEvaluator` object.

The property **IpfcGlobalEvaluator.Assem** accesses the assembly to be evaluated.

The method **IpfcGlobalEvaluator.ComputeGlobalInterference()** determines the set of all the interferences within the assembly.

This method will return a sequence of `IpfcGlobalInterference` objects or null if there are no interfering parts. Each object contains a pair of intersecting parts and an object representing the interference volume, which can be extracted by using **IpfcGlobalInterference.SelParts** and **IpfcGlobalInterference.Volume** respectively.

## Analyzing Interference Information

Methods and Properties Introduced:

- **CCpfcSelectionPair.Create()**

- **CMpfcInterference.CreateSelectionEvaluator()**
- **IpfcSelectionEvaluator.Selections**
- **IpfcSelectionEvaluator.ComputeInterference()**
- **IpfcSelectionEvaluator.ComputeClearance()**
- **IpfcSelectionEvaluator.ComputeNearestCriticalDistance()**

The method **CCpfcSelectionPair.Create()** creates a `IpfcSelectionPair` object using two `IpfcSelection` objects as arguments.

A return from this method will serve as an argument to **CMpfcInterference.CreateSelectionEvaluator()**, which will provide a way to determine the interference data between the two selections.

**IpfcSelectionEvaluator.Selections** will extract and set the object to be evaluated respectively.

**IpfcSelectionEvaluator.ComputeInterference()** determines the interfering information about the provided selections. This method will return the `IpfcInterferenceVolume` object or null if the selections do not interfere.

**IpfcSelectionEvaluator.ComputeClearance()** computes the clearance data for the two selection. This method returns a `IpfcClearanceData` object, which can be used to obtain and set clearance distance, nearest points between selections, and a boolean `IsInterfering` variable.

**IpfcSelectionEvaluator.ComputeNearestCriticalDistance()** finds a critical point of the distance function between two selections.

This method returns a `IpfcCriticalDistanceData` object, which is used to determine and set critical points, surface parameters, and critical distance between points.

## Analyzing Interference Volume

Methods and Properties Introduced:

- **IpfcInterferenceVolume.ComputeVolume()**
- **IpfcInterferenceVolume.Highlight()**
- **IpfcInterferenceVolume.Boundaries**

The method **IpfcInterferenceVolume.ComputeVolume()** will calculate a value for interfering volume.

The method **IpfcInterferenceVolume.Highlight()** will highlight the interfering volume with the color provided in the argument to the function.

The property **IpfcInterferenceVolume.Boundaries** will return a set of boundary surface descriptors for the interference volume.

### Example Code

This application finds the interference in an assembly, highlights the interfering surfaces, and highlights

calculates the interference volume.

```
Imports pfcls
```

```
Public Class pfcGeometryExamples
```

```
    Public Sub showInterferences(ByRef session As IpfcBaseSession)
```

```
        Dim model As IpfcModel
        Dim assembly As IpfcAssembly
        Dim globalEval As IpfcGlobalEvaluator
        Dim globalInterferences As IpfcGlobalInterferences
        Dim globalInterference As IpfcGlobalInterference
        Dim selectionPair As IpfcSelectionPair
        Dim selection1, selection2 As IpfcSelection
        Dim interVolume As IpfcInterferenceVolume
        Dim totalVolume As Double
        Dim noInterferences As Integer
        Dim i As Integer
```

```
        Try
```

```
        '=====
```

```
        'Get the current solid
```

```
        '=====
```

```
            model = session.CurrentModel
            If model Is Nothing Then
                Throw New Exception("Model not present")
            End If
            If (Not model.Type = EpfcModelType.EpfcMDL_ASSEMBLY) Then
                Throw New Exception("Model is not an assembly")
            End If
            assembly = CType(model, IpfcAssembly)

            globalEval = (New CMpfcInterference).CreateGlobalEvaluator(assembly)
```

```
        '=====
```

```
        'Select the list of interferences in the assembly
```

```
        'Setting parameter to true will select only solid geometry
```

```
        'Setting it to false will through an exception
```

```
        '=====
```

```
            globalInterferences = globalEval.ComputeGlobalInterference(True)
```

```
            If globalInterferences Is Nothing Then
```

```
                Throw New Exception("No interference detected in assembly : " +  
assembly.FullName)
```

```
                Exit Sub
```

```
            End If
```

```
        '=====
```

```
        'For each interference display interfering surfaces and calculate the
```

```
        'interfering volume
```

```
        '=====
```

```
            noInterferences = globalInterferences.Count
```

```
            For i = 0 To noInterferences - 1
```

```
                globalInterference = globalInterferences.Item(i)
```

```
        selectionPair = globalInterference.SelParts
        selection1 = selectionPair.Sel1
        selection2 = selectionPair.Sel2
        selection1.Highlight(EpfcStdColor.EpfcCOLOR_HIGHLIGHT)
        selection2.Highlight(EpfcStdColor.EpfcCOLOR_HIGHLIGHT)

        interVolume = globalInterference.Volume
        totalVolume = interVolume.ComputeVolume()

        MsgBox("Interference " + i.ToString + " Volume : " + totalVolume.
ToString)
        interVolume.Highlight(EpfcStdColor.EpfcCOLOR_ERROR)

    Next

    Catch ex As Exception
        MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
        Exit Sub
    End Try
End Sub

End Class
```

---

# Dimensions and Parameters

---

This section describes the VB API methods and classes that affect dimensions and parameters.

## Topic

[Overview](#)

[The ParamValue Object](#)

[Parameter Objects](#)

[Dimension Objects](#)

## Overview

Dimensions and parameters in Pro/ENGINEER have similar characteristics but also have significant differences. In the VB API, the similarities between dimensions and parameters are contained in the `IpfcBaseParameter` interface. This interface allows access to the parameter or dimension value and to information regarding a parameter's designation and modification. The differences between parameters and dimensions are recognizable because `IpfcDimension` inherits from the interface `IpfcModelItem`, and can be assigned tolerances, whereas parameters are not `IpfcModelItems` and cannot have tolerances.

## The ParamValue Object

Both parameters and dimension objects contain an object of type `IpfcParamValue`. This object contains the integer, real, string, or Boolean value of the parameter or dimension. Because of the different possible value types that can be associated with a `IpfcParamValue` object there are different methods used to access each value type and some methods will not be applicable for some `IpfcParamValue` objects. If you try to use an incorrect method an exception will be thrown.

## Accessing a ParamValue Object

Methods and Property Introduced:

- **`CMpfcModelItem.CreateIntParamValue()`**
- **`CMpfcModelItem.CreateDoubleParamValue()`**
- **`CMpfcModelItem.CreateStringParamValue()`**
- **`CMpfcModelItem.CreateBoolParamValue()`**
- **`CMpfcModelItem.CreateNoteParamValue()`**
- **`IpfcBaseParameter.Value`**

The `CMpfcModelItem` utility class contains methods for creating each type of `IpfcParamValue` object. Once you have established the value type in the object, you can change it. The property **`IpfcBaseParameter.Value`** returns the `IpfcParamValue` associated with a particular parameter or dimension.

A `NoteIpfcParamValue` is an integer value that refers to the ID of a specified note. To create a parameter of this type the identified note must already exist in the model.

## Accessing the ParamValue Value

Properties Introduced:

- **`IpfcParamValue.discr`**
- **`IpfcParamValue.IntValue`**
- **`IpfcParamValue.DoubleValue`**
- **`IpfcParamValue.StringValue`**
- **`IpfcParamValue.BoolValue`**
- **`IpfcParamValue.NoteId`**

The property **`IpfcParamValue.discr`** returns an enumeration object that identifies the type of value contained in the `IpfcParamValue` object. Use this information with the specified properties to access the value. If you use an incorrect property an exception of type `pfCXBadGetParamValue` will be thrown.

## Parameter Objects

The following sections describe the VB API methods that access parameters. The topics are as follows:

- Creating and Accessing Parameters
- Parameter Selection Options
- Parameter Information
- Parameter Restrictions

## Creating and Accessing Parameters

Methods and Property Introduced:

- **`IpfcParameterOwner.CreateParam()`**
- **`IpfcParameterOwner.CreateParamWithUnits()`**
- **`IpfcParameterOwner.GetParam()`**
- **`IpfcParameterOwner.ListParams()`**

- **IpfcParameterOwner.SelectParam()**
- **IpfcParameterOwner.SelectParameters()**
- **IpfcFamColParam.RefParam**

In the VB API, models, features, surfaces, and edges inherit from the **IpfcParameterOwner** interface, because each of the objects can be assigned parameters in Pro/ENGINEER.

The method **IpfcParameterOwner.GetParam()** gets a parameter given its name.

The method **IpfcParameterOwner.ListParams()** returns a sequence of all parameters assigned to the object.

To create a new parameter with a name and a specific value, call the method **IpfcParameterOwner.CreateParam()**.

To create a new parameter with a name, a specific value, and units, call the method **IpfcParameterOwner.CreateParamWithUnits()**.

The method **IpfcParameterOwner.SelectParam()** allows you to select a parameter from the Pro/ENGINEER user interface. The top model from which the parameters are selected must be displayed in the current window.

The method **IpfcParameterOwner.SelectParameters()** allows you to interactively select parameters from the Pro/ENGINEER Parameter dialog box based on the parameter selection options specified by the **IpfcParameterSelectionOptions** object. The top model from which the parameters are selected must be displayed in the current window. Refer to the section [Parameter Selection Options](#) for more information.

The property **IpfcFamColParam.RefParam** returns the reference parameter from the parameter column in a family table.

## Parameter Selection Options

Parameter selection options in the VB API are represented by the **IpfcParameterSelectionOptions** interface.

Methods and Properties Introduced:

- **CCpfcParameterSelectionOptions.Create()**
- **IpfcParameterSelectionOptions.AllowContextSelection**
- **IpfcParameterSelectionOptions.Contexts**
- **IpfcParameterSelectionOptions.AllowMultipleSelections**
- **IpfcParameterSelectionOptions.SelectButtonLabel**

The method **CCpfcParameterSelectionOptions.Create()** creates a new instance of the

**IpfcParameterSelectionOptions** object that is used by the method **IpfcParameterOwner.SelectParameters()**.

The parameter selection options are as follows:

- AllowContextSelection--This boolean attribute indicates whether to allow parameter selection from multiple contexts, or from the invoking parameter owner. By default, it is false and allows selection only from the invoking parameter owner. If it is true and if specific selection contexts are not yet assigned, then you can select the parameters from any context.  
Use the property `IpfcParameterSelectionOptions.AllowContextSelection` to modify the value of this attribute.
- Contexts--The permitted parameter selection contexts in the form of the `IpfcParameterSelectionContexts` object. Use the property `IpfcParameterSelectionOptions.Contexts` to assign the parameter selection context. By default, you can select parameters from any context.

The types of parameter selection contexts are as follows:

- `EpfParamSelectModel`--Specifies that the top level model parameters can be selected.
  - `EpfParamSelectPart`--Specifies that any part's parameters (at any level of the top model) can be selected.
  - `EpfParamSelectAsm`--Specifies that any assembly's parameters (at any level of the top model) can be selected.
  - `EpfParamSelectFeature`--Specifies that any feature's parameters can be selected.
  - `EpfParamSelectEdge`--Specifies that any edge's parameters can be selected.
  - `EpfParamSelectSurface`--Specifies that any surface's parameters can be selected.
  - `EpfParamSelectQuilt`--Specifies that any quilt's parameters can be selected.
  - `EpfParamSelectCurve`--Specifies that any curve's parameters can be selected.
  - `EpfParamSelectCompositeCurve`--Specifies that any composite curve's parameters can be selected.
  - `EpfParamSelectInherited`--Specifies that any inheritance feature's parameters can be selected.
  - `EpfParamSelectSkeleton`--Specifies that any skeleton's parameters can be selected.
  - `EpfParamSelectComponent`--Specifies that any component's parameters can be selected.
- AllowMultipleSelections--This boolean attribute indicates whether or not to allow multiple parameters to be selected from the dialog box, or only a single parameter. By default, it is true and allows selection of multiple parameters.  
Use the property `IpfcParameterSelectionOptions.AllowMultipleSelections` to modify this attribute.
  - SelectButtonLabel--The visible label for the select button in the dialog box.  
Use the property `IpfcParameterSelectionOptions.SelectButtonLabel` to set the label. If not set, the default label in the language of the active Pro/ENGINEER session is displayed.

## Parameter Information

Methods and Properties Introduced:

- **IpfcBaseParameter.Value**
- **IpfcParameter.GetScaledValue()**
- **IpfcParameter.SetScaledValue()**
- **IpfcParameter.Units**

- **IpfcBaseParameter.IsDesignated**
- **IpfcBaseParameter.IsModified**
- **IpfcBaseParameter.ResetFromBackup()**
- **IpfcParameter.Description**
- **IpfcParameter.GetRestriction()**
- **IpfcParameter.GetDriverType()**
- **IpfcParameter.Reorder()**
- **IpfcParameter.Delete()**
- **IpfcNamedModelItem.Name**

Parameters inherit methods from the **IpfcBaseParameter**, **IpfcParameter**, and **IpfcNamedModelItem** interfaces.

The property **IpfcBaseParameter.Value** returns the value of the parameter or dimension.

The method **IpfcParameter.GetScaledValue()** returns the parameter value in the units of the parameter, instead of the units of the owner model as returned by **IpfcBaseParameter.Value**.

The method **IpfcParameter.SetScaledValue()** assigns the parameter value in the units provided, instead of using the units of the owner model as assumed by **IpfcBaseParameter.Value**.

The method **IpfcParameter.Units** returns the units assigned to the parameter.

You can access the designation status of the parameter using the property **IpfcBaseParameter.IsDesignated**.

The property **IpfcBaseParameter.IsModified** and the method **IpfcBaseParameter.ResetFromBackup()** enable you to identify a modified parameter or dimension, and reset it to the last stored value. A parameter is said to be "modified" when the value has been changed but the parameter's owner has not yet been regenerated.

The property **IpfcParameter.Description** returns the parameter description, or null, if no description is assigned.

The property **IpfcParameter.Description** assigns the parameter description.

The property **IpfcParameter.GetRestriction()** identifies if the parameter's value is restricted to a certain range or enumeration. It returns the **IpfcParameterRestriction** object. Refer to the section [Parameter Restrictions](#) for more information.

The property **IpfcParameter.GetDriverType()** returns the driver type for a material parameter. The driver types are as follows:

- **EpcfPARAMDRIVER\_PARAM**--Specifies that the parameter value is driven by another parameter.
- **EpcfPARAMDRIVER\_FUNCTION**--Specifies that the parameter value is driven by a function.
- **EpcfPARAMDRIVER\_RELATION**--Specifies that the parameter value is driven by a relation. This is equivalent to the value obtained using **IpfcBaseParameter.IsRelationDriven** for a parameter object type.

The method **IpfcParameter.Reorder()** reorders the given parameter to come immediately after the indicated parameter in the Parameter dialog box and information files generated by Pro/ENGINEER.

The method **IpfcParameter.Delete()** permanently removes a specified parameter.

The property **IpfcNamedModelItem.Name** accesses the name of the specified parameter.

## Parameter Restrictions

Pro/ENGINEER allows users to assign specified limitations to the value allowed for a given parameter (wherever the parameter appears in the model). You can only read the details of the permitted restrictions from the VB API, but not modify the permitted values or range of values. Parameter restrictions in the VB API are represented by the interface **IpfcParameterRestriction**.

Method Introduced:

- **IpfcParameterRestriction.Type**

The method **IpfcParameterRestriction.Type** returns the **IpfcRestrictionType** object containing the types of parameter restrictions. The parameter restrictions are of the following types:

- **EpcfPARAMSELECT\_ENUMERATION**--Specifies that the parameter is restricted to a list of permitted values.
- **EpcfPARAMSELECT\_RANGE**--Specifies that the parameter is limited to a specified range of numeric values.

## Enumeration Restriction

The **EpcfPARAMSELECT\_ENUMERATION** type of parameter restriction is represented by the interface **IpfcParameterEnumeration**. It is a child of the **IpfcParameterRestriction** interface.

Property Introduced:

- **IpfcParameterEnumeration.PermittedValues**

The property **IpfcParameterEnumeration.PermittedValues** returns a list of permitted parameter values allowed by this restriction in the form of a sequence of the **IpfcParamValue** objects.

## Range Restriction

The **EpcfPARAMSELECT\_RANGE** type of parameter restriction is represented by the interface **IpfcParameterRange**. It is a child of the **IpfcParameterRestriction** interface.

Properties Introduced:

- **IpfcParameterRange.Maximum**
- **IpfcParameterRange.Minimum**
- **IpfcParameterLimit.Type**
- **IpfcParameterLimit.Value**

The property **IpfcParameterRange.Maximum** returns the maximum value limit for the parameter in the form of the **IpfcParameterLimit** object.

The property **IpfcParameterRange.Minimum** returns the minimum value limit for the parameter in the form of the **IpfcParameterLimit** object.

The property **IpfcParameterLimit.Type** returns the **IpfcParameterLimitType** containing the types of parameter limits. The parameter limits are of the following types:

- **EpfcPARAMLIMIT\_LESS\_THAN**--Specifies that the parameter must be less than the indicated value.
- **EpfcPARAMLIMIT\_LESS\_THAN\_OR\_EQUAL**--Specifies that the parameter must be less than or equal to the indicated value.
- **EpfcPARAMLIMIT\_GREATER\_THAN**--Specifies that the parameter must be greater than the indicated value.
- **EpfcPARAMLIMIT\_GREATER\_THAN\_OR\_EQUAL**--Specifies that the parameter must be greater than or equal to the indicated value.

The property **IpfcParameterLimit.Value** retruns the boundary value of the parameter limit in the form of the **IpfcParamValue** object.

#### **Example Code: Updating Model Parameters**

The following example code contains a method that reads a "properties" file and creates or updates model parameters for each property which exists in the file. Since each property value is returned as a String, a utility method parses the String into int, double, or boolean values if possible

```
Imports pfcls

Public Class pfcDimensionAndParameterExamples
    Public Sub createParametersFromProperties(ByRef pOwner As
IpfcParameterOwner, _
                                           ByVal propertiesFile As String)
        Dim file As IO.StreamReader = Nothing
        Dim s As String
        Dim split() As String
        Dim pv As IpfcParamValue
        Dim p As IpfcParameter

        Try
'=====
'Use a stream reader to read the properties file
'=====
```

```

        file = New IO.StreamReader(propertiesFile)
'=====
'Read and parse line into key - value pairs. These are separated by
':"". Any line starting with # is ignored as comments
'=====
        While Not file.EndOfStream
            s = file.ReadLine()
            If Not (s.Substring(0, 1) = "#") Then
                split = s.Split(":")
'=====
'Invalid key - value pairs are ignored
'=====
                If split.Length = 2 Then

                    pv = createParamValueFromString(split(1).ToString)
                    p = pOwner.GetParam(split(0).ToString)
                    If p Is Nothing Then
                        pOwner.CreateParam(split(0).ToString, pv)
                    Else
                        CType(p, IpfcBaseParameter).Value = pv
                    End If
                End If
            End If

        End While

    Catch ex As Exception
        MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
    Finally
        If Not file Is Nothing Then
            file.Close()
        End If
    End Try
End Sub

'Create Parameters from string
'=====
'Function    : createParamValueFromString
'Purpose    : This method parses a string into a ParamValue object.
'            Useful for reading ParamValues from file or from UI text.
'            This method checks if the value is a proper integer,
'            double, or boolean, and if so, returns a value of that
'            type. If the value is not a number or boolean, the
'            method returns a String ParamValue.
'=====
Private Function createParamValueFromString(ByVal s As String) _
    As IpfcParamValue

    Try
        If (s.Equals("Y", StringComparison.OrdinalIgnoreCase)) Or
            _(s.Equals("true", StringComparison.OrdinalIgnoreCase))
        Then
            Return ((New CMpfcModelItem).CreateBoolParamValue(True))

        ElseIf (s.Equals("N", StringComparison.OrdinalIgnoreCase)) Or

```

```

        _s.Equals("false", StringComparison.OrdinalIgnoreCase))
Then
    Return ((New CMpfcModelItem).CreateBoolParamValue(False))

ElseIf IsDouble(s) Then
    Return ((New CMpfcModelItem).CreateDoubleParamValue
            (CType(s, Double)))

ElseIf IsNumeric(s) Then
    Return ((New CMpfcModelItem).CreateIntParamValue(CType(s,
                                                    Integer)))

Else
    Return ((New CMpfcModelItem).CreateStringParamValue(s))

End If

Catch ex As Exception
    MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
Return Nothing
End Try

End Function
'=====
'Function    : IsDouble
'Purpose     : Helper function to check if string is decimal
'=====
Private Function IsDouble(ByVal s As String) As Boolean
    Dim i As Integer

    If IsNumeric(s) Then
        For i = 0 To s.Length - 2
            If s.Substring(i, 1) = "." Then
                Return True
            End If
        Next
    End If

    Return False

End Function

End Class

```

## Dimension Objects

Dimension objects include standard Pro/ENGINEER dimensions as well as reference dimensions. Dimension objects enable you to access dimension tolerances and enable you to set the value for the dimension. Reference dimensions allow neither of these actions.

## Getting Dimensions

Dimensions and reference dimensions are Pro/ENGINEER model items. See for methods that can return

IpfcDimension and IpfcRefDimension objects.

## Dimension Information

Methods and Properties Introduced:

- **IpfcBaseParameter.Value**
- **IpfcBaseDimension.DimValue**
- **IpfcBaseParameter.IsDesignated**
- **IpfcBaseParameter.IsModified**
- **IpfcBaseParameter.ResetFromBackup()**
- **IpfcBaseParameter.IsRelationDriven**
- **IpfcBaseDimension.DimType**
- **IpfcBaseDimension.Symbol**
- **IpfcBaseDimension.Texts**

All the `IpfcBaseParameter` methods are accessible to `Dimensions` as well as `Parameters`. See "[Parameter Objects](#)" for brief descriptions.

### Note:

You cannot set the value or designation status of reference dimension objects.

The property **IpfcBaseDimension.DimValue** accesses the dimension value as a double. This property provides a shortcut for accessing the dimensions' values without using a `ParamValue` object.

The **IpfcBaseParameter.IsRelationDriven** property identifies whether the part or assembly relations control a dimension.

The property **IpfcBaseDimension.DimType** returns an enumeration object that identifies whether a dimension is linear, radial, angular, or diametrical.

The property **IpfcBaseDimension.Symbol** returns the dimension or reference dimension symbol (that is, "d#" or "rd#").

The property **IpfcBaseDimension.Texts** allows access to the text strings that precede or follow the dimension value.

## Dimension Tolerances

Methods and Properties Introduced:



```
ByVal range As Double) _  
As IpfcDimension
```

```
Dim paramValue As IpfcParamValue  
Dim limits As IpfcDimTolLimits  
Dim dimValue As Double  
Dim upper, lower As Double
```

```
Try
```

```
  If (dimension.DimType = EpfcDimensionType.EpfcDIM_ANGULAR) Then  
    paramValue = dimension.Value  
    dimValue = paramValue.DoubleValue()
```

```
    upper = dimValue + (range / 2)  
    lower = dimValue - (range / 2)
```

```
    limits = (New CCpfcDimTolLimits).Create(upper, lower)
```

```
    dimension.Tolerance = limits
```

```
  End If
```

```
  setAngularToleranceToLimits = dimension
```

```
Catch ex As Exception
```

```
  MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
```

```
  Return Nothing
```

```
End Try
```

```
End Function
```

---

# Relations

---

This section describes how to access relations on all models and model items in Pro/ENGINEER using the methods provided in the VB API.

## Topic

[Accessing Relations](#)

[Adding a Customized Function to the Relations Dialog Box in Pro/ENGINEER](#)

## Accessing Relations

In the VB API, the set of relations on any model or model item is represented by the **IpfcRelationOwner** interface. Models, features, surfaces, and edges inherit from this interface, because each object can be assigned relations in Pro/ENGINEER.

Methods and Properties Introduced:

- **IpfcRelationOwner.RegenerateRelations()**
- **IpfcRelationOwner.DeleteRelations()**
- **IpfcRelationOwner.Relations**
- **IpfcRelationOwner.EvaluateExpression()**

The method **IpfcRelationOwner.RegenerateRelations()** regenerates the relations assigned to the owner item. It also determines whether the specified relation set is valid.

The method **IpfcRelationOwner.DeleteRelations()** deletes all the relations assigned to the owner item.

The property **IpfcRelationOwner.Relations** returns the list of actual relations assigned to the owner item as a sequence of strings.

The method **IpfcRelationOwner.EvaluateExpression()** evaluates the given relations-based expression, and returns the resulting value in the form of the **IpfcParamValue** object. Refer to the section, [The ParamValue Object](#) in the chapter, [Dimensions and Parameters](#) for more information on this object.

### Example 1: Adding Relations between Parameters in a Solid Model

```
Public Class pfcRelationsExamples2
    '=====
    'Function      :   createParamDimRelation
    'Purpose      :   This function creates parameters for all dimensions in
    '              :   all features of a part model and adds relation between
    '              :   them.
    '=====
    Public Sub createParamDimRelation(ByRef features As IpfcFeatures)
```

```

Dim items As IpfcModelItems
Dim item As IpfcModelItem
Dim feature As IpfcFeature
Dim i, j As Integer
Dim paramName As String
Dim dimName As String
Dim dimValue As Double
Dim relations As Cstringseq
Dim paramValue As IpfcParamValue
Dim param As IpfcParameter
Dim paramAdded As Boolean

```

```
Try
```

```
For i = 0 To features.Count - 1
    feature = features.Item(i)
```

```

'=====
'Get the dimensions in the current feature
'=====
    items = feature.ListSubItems(EpfcModelItemType.EpfcITEM_DIMENSION)
    If items Is Nothing OrElse items.Count = 0 Then
        Continue For
    End If

    relations = New Cstringseq
'=====
'Loop through all the dimensions and create relations
'=====
    For j = 0 To items.Count - 1
        item = items.Item(j)
        dimName = item.GetName()
        paramName = "PARAM_" + dimName
        dimValue = CType(item, IpfcBaseDimension).DimValue

        param = feature.GetParam(paramName)
        paramAdded = False
        If param Is Nothing Then
            paramValue = (New
                CmpfcModelItem).CreateDoubleParamValue(dimValue)
            feature.CreateParam(paramName, paramValue)
            paramAdded = True
        Else
            If param.Value.discr = EpfcParamValueType.EpfcPARAM_DOUBLE
                Then
                    paramValue = (New
                        CmpfcModelItem).CreateDoubleParamValue(dimValue)
                    CType(param, IpfcBaseParameter).Value = paramValue
                    paramAdded = True
                End If
            End If

            If paramAdded = True Then
                relations.Append(dimName + " = " + paramName)
            End If
            param = Nothing
        End If
    Next j

```

```

        Next
        CType(feature, IpfcRelationOwner).Relations = relations
    Next
Catch ex As Exception
    MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
End Try
End Sub
End Class

```

## Adding a Customized Function to the Relations Dialog Box in Pro/ENGINEER

Methods Introduced:

- **IpfcBaseSession.RegisterRelationFunction()**

The method **IpfcBaseSession.RegisterRelationFunction()** registers a custom function that is included in the function list of the Relations dialog box in Pro/ENGINEER. You can add the custom function to relations that are added to models, features, or other relation owners. The registration method takes the following input arguments:

- Name--The name of the custom function.
- IpfcRelationFunctionOptions--This object contains the options that determine the behavior of the custom relation function. Refer to the section 'Relation Function Options' for more information.
- IpfcRelationFunctionListener--This object contains the action listener methods for the implementation of the custom function. Refer to the section 'Relation Function Listeners' for more information.

**Note:**

the VB API relation functions are valid only when the custom function has been registered by the application. If the application is not running or not present, models that contain user-defined relations cannot evaluate these relations. In this situation, the relations are marked as errors. However, these errors can be commented until needed at a later time when the relations functions are reactivated in a Pro/ENGINEER session.

## Relation Function Options

Methods and Properties Introduced:

- **CCpfcRelationFunctionOptions.Create()**
- **IpfcRelationFunctionOptions.ArgumentTypes**
- **CCpfcRelationFunctionArgument.Create()**
- **IpfcRelationFunctionArgument.Type**
- **IpfcRelationFunctionArgument.IsOptional**
- **IpfcRelationFunctionOptions.EnableTypeChecking**
- **IpfcRelationFunctionOptions.EnableArgumentCheckMethod**
- **IpfcRelationFunctionOptions.EnableExpressionEvaluationMethod**

- **IpfcRelationFunctionOptions.EnableValueAssignmentMethod**

Use the method **CCpfcRelationFunctionOptions.Create()** to create the **IpfcRelationFunctionOptions** object containing the options to enable or disable various relation function related features. Use the methods listed above to access and modify the options. These options are as follows:

- **ArgumentTypes**--The types of arguments in the form of the **IpfcRelationFunctionArgument** object. By default, this parameter is null, indicating that no arguments are permitted.

Use the method **CCpfcRelationFunctionArgument.Create()** to create the **IpfcRelationFunctionArgument** object containing the attributes of the arguments passed to the custom relation function.

These attributes are as follows:

- **Type**--The type of argument value such as double, integer, and so on in the form of the **IpfcParamValueType** object.
- **IsOptional**--This boolean attribute specifies whether the argument is optional, indicating that it can be skipped when a call to the custom relation function is made. The optional arguments must fall at the end of the argument list. By default, this attribute is false.
- **EnableTypeChecking**--This boolean attribute determines whether or not to check the argument types internally. By default, it is false. If this attribute is set to false, Pro/ENGINEER does not need to know the contents of the arguments array. The custom function must handle all user errors in such a situation.
- **EnableArgumentCheckMethod**--This boolean attribute determines whether or not to enable the arguments check listener function. By default, it is false.
- **EnableExpressionEvaluationMethod**--This boolean attribute determines whether or not to enable the evaluate listener function. By default, it is true.
- **EnableValueAssignmentMethod**--This boolean attribute determines whether or not to enable the value assignment listener function. By default, it is false.

## Relation Function Listeners

The interface **IpfcRelationFunctionListener** provides the method signatures to implement a custom relation function.

Methods Introduced:

- **IpfcRelationFunctionListener.CheckArguments()**
- **IpfcRelationFunctionListener.AssignValue()**
- **IpfcRelationFunctionListener.EvaluateFunction()**

The method **IpfcRelationFunctionListener.CheckArguments()** checks the validity of the arguments passed to the custom function. This listener method takes the following input arguments:

- The owner of the relation being evaluated
- The custom function name
- A sequence of arguments passed to the custom function

If the implementation of this method determines that the arguments are not valid for the custom function, then the listener method returns false. Otherwise, it returns true.

The method **IpfcRelationFunctionListener.EvaluateFunction()** evaluates a custom relation function invoked on the right hand side of a relation. This listener method takes the following input arguments:

- The owner of the relation being evaluated

- o The custom function name
- o A sequence of arguments passed to the custom function

You must return the computed result of the custom relation function.

The method **IpfcRelationFunctionListener.AssignValue()** evaluates a custom relation function invoked on the left hand side of a relation. It allows you to initialize properties to be stored and used by your application. This listener method takes the following input arguments:

- o The owner of the relation being evaluated
- o The custom function name
- o A sequence of arguments passed to the custom function
- o The value obtained by Pro/ENGINEER from evaluating the right hand side of the relation

## Example 2: Adding and Implementing a New Custom Relation Function

The addRelation function in this example code, which defines the options for a new custom relation function and registers it in the current session. The RelationListener class contains the CheckArguments, AssignValue and EvaluateFunction listener methods that are called when the custom relation function is used.

```
Public Class pfcrRelationsExamples1
    Implements IpfcAsyncActionListener
    Implements ICIPClientObject
    Implements IpfcActionListener

    Dim WithEvents eventTimer As Timers.Timer
    Dim exitFlag As Boolean = False
    Dim aC As pfcls.IpfcAsyncConnection

    Public Sub New(ByRef asyncConnection As pfcls.IpfcAsyncConnection)
        aC = asyncConnection
    End Sub

    Public Function GetClientInterfaceName() As String Implements
        pfcls.ICIPClientObject.GetClientInterfaceName
        GetClientInterfaceName = "IpfcAsyncActionListener"
    End Function

    Public Sub OnTerminate(ByVal _Status As Integer) Implements
        pfcls.IpfcAsyncActionListener.OnTerminate
        aC.InterruptEventProcessing()
        exitFlag = True
    End Sub

' =====
'Function      :   timeElapsed
'Purpose      :   This function handels the time elapsed event of timer
'               which is fired at regular intervals
' =====
Private Sub timeElapsed(ByVal sender As Object, ByVal e As
    System.Timers.ElapsedEventArgs)
    If exitFlag = False Then
        aC.EventProcess()
    Else

```

```

        eventTimer.Enabled = False
    End If
End Sub

```

```

'=====
'Function      :   addRelation
'Purpose       :   This function adds new custom relation functions.
'=====

```

```

Public Sub addRelation()
    Dim listenerObj As RelationListener

    Dim setOptions As IpfcRelationFunctionOptions
    Dim getOptions As IpfcRelationFunctionOptions
    Dim getArgs As IpfcRelationFunctionArguments
    Dim getArg As IpfcRelationFunctionArgument

```

```

    Try
        listenerObj = New RelationListener()

```

```

'=====
'Start the timer to call EventProcess at regular intervals
'=====

```

```

    eventTimer = New Timers.Timer(50)
    eventTimer.Enabled = True
    AddHandler eventTimer.Elapsed, AddressOf Me.timeElapsed

```

```

    setOptions = (New CCpfcRelationFunctionOptions).Create()
    setOptions.EnableArgumentCheckMethod = False
    setOptions.EnableExpressionEvaluationMethod = False
    setOptions.EnableTypeChecking = False
    setOptions.EnableValueAssignmentMethod = True

```

```

    aC.Session.RegisterRelationFunction("SET_A", listenerObj,
                                        setOptions)
    aC.Session.RegisterRelationFunction("SET_B", listenerObj,
                                        setOptions)

```

```

    getArgs = New CpfcRelationFunctionArguments
    getArg = (New

```

```

CCpfcRelationFunctionArgument).Create(EpfcParamValueType.EpfcPARAM_DOUBL)
    getArg.IsOptional = False

```

```

    getArgs.Append(getArg)

```

```

    getOptions = (New CCpfcRelationFunctionOptions).Create()
    getOptions.EnableTypeChecking = False
    getOptions.ArgumentTypes = getArgs

```

```

    aC.Session.RegisterRelationFunction("EVAL_AX_B", listenerObj,
                                        getOptions)

```

```

    aC.AddActionListener(Me)

```

```

    Catch ex As Exception
        MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
    End Try

```

```

End Sub

```

```

'=====
'Class      : RelationListener
'Purpose    : This class implements the IpfcRelationFunctionListener
'            Interface along with the correct client interface name.
'            The implemented method will be called when the custom
'            relation function is used.
'=====
Private Class RelationListener
    Implements IpfcRelationFunctionListener
    Implements ICIPClientObject
    Implements IpfcActionListener

    Dim aValue As Double = 1
    Dim bValue As Double = 0

    Public Function GetClientInterfaceName() As String Implements
        pfcls.ICIPClientObject.GetClientInterfaceName
        GetClientInterfaceName = "IpfcRelationFunctionListener"
    End Function

'=====
'Function   : AssignValue
'Purpose    : Function called when value is assigned to custom
'            relation function.
'=====
    Public Sub AssignValue(ByVal _Owner As pfcls.IpfcRelationOwner,
        ByVal _FunctionName As String, ByVal _Arguments As pfcls.CpfcParamValues,
        ByVal _Assignment As pfcls.IpfcParamValue) Implements
        pfcls.IpfcRelationFunctionListener.AssignValue
        If Not _Assignment.discr = EpfcParamValueType.EpfcPARAM_DOUBLE
            Then
                Throw New Exception("Incorrect type")
            End If
            If _FunctionName = "SET_A" Then
                aValue = _Assignment.DoubleValue
            End If
            If _FunctionName = "SET_B" Then
                bValue = _Assignment.DoubleValue
            End If
        End Sub

'=====
'Function   : CheckArguments
'Purpose    : Function called to check arguments supplied
'            to custom relation function.
'=====
    Public Function CheckArguments(ByVal _Owner As
        pfcls.IpfcRelationOwner, ByVal _FunctionName As String, ByVal _Arguments
        As pfcls.CpfcParamValues) As Boolean Implements
        pfcls.IpfcRelationFunctionListener.CheckArguments
    End Function

'=====
'Function   : EvaluateFunction
'Purpose    : Function called when value is to be returned from
'            custom relation function.
'=====
    Public Function EvaluateFunction(ByVal _Owner As

```

```
pfcls.IpfcRelationOwner, ByVal _FunctionName As String, ByVal _Arguments  
As pfcls.CpfcParamValues) As pfcls.IpfcParamValue Implements  
pfcls.IpfcRelationFunctionListener.EvaluateFunction
```

```
    Dim paramValue As IpfcParamValue  
    Dim ret As Double  
  
    If _FunctionName = "EVAL_AX_B" Then  
        ret = (aValue * (_Arguments.Item(0).DoubleValue)) + bValue  
        paramValue = (New  
                        CMpfcModelItem).CreateDoubleParamValue(ret)  
        Return paramValue  
    Else  
        Return Nothing  
    End If
```

```
End Function
```

```
End Class
```

```
End Class
```

---

# Assemblies and Components

---

This section describes the the VB API functions that access the functions of a Pro/ENGINEER assembly. You must be familiar with the following before you read this section:

- The Selection Object
- Coordinate Systems
- The Geometry section

## Topic

[Structure of Assemblies and Assembly Objects](#)

[Assembling Components](#)

[Redefining and Rerouting Assembly Components](#)

[Exploded Assemblies](#)

[Skeleton Models](#)

## Structure of Assemblies and Assembly Objects

The object `IpfcAssembly` is an instance of `IpfcSolid`. The `IpfcAssembly` object can therefore be used as input to any of the `IpfcSolid` and `IpfcModel` methods applicable to assemblies. However assemblies do not contain solid geometry items. The only geometry in the assembly is datums (points, planes, axes, coordinate systems, curves, and surfaces). Therefore solid assembly features such as holes and slots will not contain active surfaces or edges in the assembly model.

The solid geometry of an assembly is contained in its components. A component is a feature of type `IpfcComponentFeat`, which is a reference to a part or another assembly, and a set of parametric constraints for determining its geometrical location within the parent assembly.

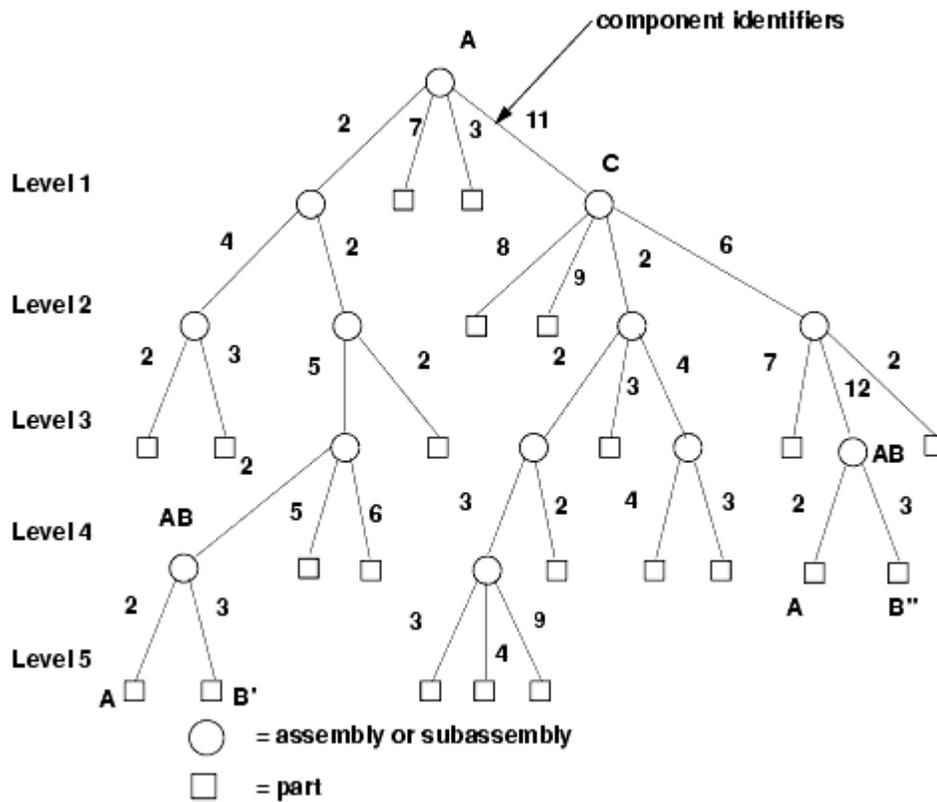
Assembly features that are solid, such as holes and slots, and therefore affect the solid geometry of parts in the assembly hierarchy, do not themselves contain the geometry items that describe those modifications. These items are always contained in the parts whose geometry is modified, within local features created for that purpose.

The important functions for assemblies are those that operate on the components of an assembly. The object `IpfcComponentFeat`, which is an instance of `IpfcFeature` 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 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 sufficient to have the integer identifier of the item and the handle to its owning part or assembly, as would be provided by its `IpfcFeature` 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. This is the purpose of the object `IComponentPath`, which is used as the input to the VB API assembly functions.

The following figure shows an assembly hierarchy with two examples of the contents of a `IpfcComponentPath` object.



In the assembly shown in the figure, 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''

```

ComponentIds.Item(0) = 2   ComponentIds.Item(1) = 11
ComponentIds.Item(1) = 2   ComponentIds.Item(2) = 6
ComponentIds.Item(2) = 5   ComponentIds.Item(3) = 12
ComponentIds.Item(3) = 2   ComponentIds.Item(4) = 3
ComponentIds.Item(4) = 3

```

The object `IpfComponentPath` is one of the main portions of the `IpfSelection` object.

## Assembly Components

Methods and Properties Introduced:

- `IpfComponentFeat.IsBulkitem`
- `IpfComponentFeat.IsSubstitute`
- `IpfComponentFeat.CompType`
- `IpfComponentFeat.ModelDescr`
- `IpfComponentFeat.IsPlaced`
- `IpfComponentFeat.IsPackaged`

- **IpfcComponentFeat.IsUnderconstrained**
- **IpfcComponentFeat.IsFrozen**
- **IpfcComponentFeat.Position**
- **IpfcComponentFeat.CopyTemplateContents()**
- **IpfcComponentFeat.CreateReplaceOp()**

The property **IpfcComponentFeat.IsBulkitem** identifies 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.

The property **IpfcComponentFeat.IsSubstitute** returns a true value if the component is substituted, else it returns a false. When you substitute a component in a simplified representation, you temporarily exclude the substituted component and superimpose the substituting component in its place.

The property **IpfcComponentFeat.CompType** enables you to set the type of the assembly component. The component type identifies the purpose of the component in a manufacturing assembly.

The property **IpfcComponentFeat.ModelDescr** returns the model descriptor of the component part or subassembly.

The property **IpfcComponentFeat.IsPlaced** forces the component to be considered placed. The value of this parameter is important in assembly Bill of Materials.

**Note:**

Once a component is constrained or packaged, it cannot be made unplaced again.

A component of an assembly that is either partially constrained or unconstrained is known as a packaged component. Use the property **IpfcComponentFeat.IsPackaged** to determine if the specified component is packaged.

The property **IpfcComponentFeat.IsUnderconstrained** determines if the specified component is underconstrained, that is, it possesses some constraints but is not fully constrained.

The property **IpfcComponentFeat.IsFrozen** determines if the specified component is frozen. The frozen component behaves similar to the packaged component and does not follow the constraints that you specify.

The property **IpfcComponentFeat.Position** retrieves the component's initial position before constraints and movements have been applied. If the component is packaged this position is the same as the constraint's actual position. This property modifies the assembly component data but does not regenerate the assembly component. To regenerate the component, use the method **IpfcComponentFeat.Regenerate()**.

The method **IpfcComponentFeat.CopyTemplateContents()** copies the template model into the model of the specified component.

The method **IpfcComponentFeat.CreateReplaceOp()** creates a replacement operation used to swap a component automatically with a related component. The replacement operation can be used as an argument to **IpfcSolid.ExecuteFeatureOps()**.

**Example Code: Replacing Instances**

The following example code contains a single static utility method. This method takes an assembly for an argument. It searches through the assembly for all components that are instances of the model "bolt". It then replaces all such occurrences with a different instance of bolt.

Imports pfcls

Public Class pfcAssembliesExamples

```
Public Sub replaceInstance(ByRef session As IpfcBaseSession, _  
                          ByVal modelName As String, _  
                          ByVal oldInstance As String, _  
                          ByVal newInstance As String)
```

```
Dim model As IpfcModel  
Dim assembly As IpfcAssembly  
Dim oldModel As IpfcSolid  
Dim newInstanceFamilyRow As IpfcFamilyTableRow  
Dim newModel As IpfcSolid  
Dim components As IpfcFeatures  
Dim component As IpfcComponentFeat  
Dim modelDesc As IpfcModelDescriptor  
Dim replace As IpfcCompModelReplace  
Dim replaceOperations As CpfcFeatureOperations  
Dim i As Integer
```

Try

```
'=====
```

```
'Get the current assembly
```

```
'=====
```

```
    model = session.CurrentModel  
    If model Is Nothing Then  
        Throw New Exception("Model not present")  
    End If  
    If (Not model.Type = EpfcModelType.EpfcMDL_ASSEMBLY) Then  
        Throw New Exception("Model is not an assembly")  
    End If  
    assembly = CType(model, IpfcAssembly)
```

```
'=====
```

```
'Get the model to be replaced
```

```
'=====
```

```
    oldModel = session.GetModel(modelName,  
                                EpfcModelType.EpfcMDL_PART)
```

```
'=====
```

```
'Create instance of new model
```

```
'=====
```

```
    newInstanceFamilyRow = oldModel.GetRow(newInstance)  
    newModel = newInstanceFamilyRow.CreateInstance()
```

```
    replaceOperations = New CpfcFeatureOperations
```

```
'=====
```

```
'Loop through all the components and create replace operations for any  
'instance of the model found
```

```
'=====
```

```
    components = assembly.ListFeaturesByType(False, EpfcFeatureType.  
EpfcFEATTYPE_COMPONENT)  
    For i = 0 To components.Count - 1  
        component = components.Item(i)  
        modelDesc = component.ModelDescr  
        If modelDesc.InstanceName = oldInstance Then  
            replace = component.CreateReplaceOp(newModel)
```

```

                replaceOperations.Insert(0, replace)
            End If
        Next
' =====
'Replace the model
' =====

        assembly.ExecuteFeatureOps(replaceOperations, Nothing)

    Catch ex As Exception
        MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
        Exit Sub
    End Try
End Sub
End Class

```

## Regenerating an Assembly Component

Method Introduced:

- **IpfcComponentFeat.Regenerate()**

The method **IpfcComponentFeat.Regenerate()** regenerates an assembly component. The method regenerates the assembly component just as in an interactive Pro/ENGINEER session.

## Creating a Component Path

Methods Introduced

- **CMpfcAssembly.CreateComponentPath()**

The method **CMpfcAssembly.CreateComponentPath()** returns a component path object, given the Assembly model and the integer id path to the desired component.

## Component Path Information

Methods and Properties Introduced:

- **IpfcComponentPath.Root**
- **IpfcComponentPath.ComponentIds**
- **IpfcComponentPath.Leaf**
- **IpfcComponentPath.GetTransform()**
- **IpfcComponentPath.SetTransform()**
- **IpfcComponentPath.GetIsVisible()**

The property **IpfcComponentPath.Root** returns the assembly at the head of the component path object.

The property **IpfcComponentPath.ComponentIds** returns the sequence of ids which is the path to the particular component.

The property **IpfcComponentPath.Leaf** returns the solid model at the end of the component path.

The method **IpfcComponentPath.GetTransform()** returns the coordinate system transformation between the assembly and the particular component. It has an option to provide the transformation from bottom to top, or from top to bottom. This method describes the current position and the orientation of the assembly component in the root assembly.

The method **IpfcComponentPath.SetTransform()** applies a temporary transformation to the assembly component, similar to the transformation that takes place in an exploded state. The transformation will only be applied if the assembly is using DynamicPositioning.

The method **IpfcComponentPath.GetIsVisible()** identifies if a particular component is visible in any simplified representation.

## Assembling Components

Methods Introduced:

- **IpfcAssembly.AssembleComponent()**
- **IpfcAssembly.AssembleByCopy()**
- **IpfcComponentFeat.GetConstraints()**
- **IpfcComponentFeat.SetConstraints()**

The method **IpfcAssembly.AssembleComponent()** adds a specified component model to the assembly at the specified initial position. The position is specified in the format defined by the interface **IpfcTransform3D**. 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 method **IpfcAssembly.AssembleByCopy()** creates a new component in the specified assembly by copying from the specified component. If no model is specified, then the new component is created empty. The input parameters for this method are:

- LeaveUnplaced--If true the component is unplaced. If false the component is placed at a default location in the assembly. Unplaced components belong to an assembly without being assembled or packaged. These components appear in the model tree, but not in the graphic window. Unplaced components can be constrained or packaged by selecting them from the model tree for redefinition. When its parent assembly is retrieved into memory, an unplaced component is also retrieved.
- ModelToCopy--Specify the model to be copied into the assembly
- NewModelName--Specify a name for the copied model

The method **IpfcComponentFeat.GetConstraints()** retrieves the constraints for a given assembly component.

The method **IpfcComponentFeat.SetConstraints()** allows you to set the constraints for a specified assembly component. The input parameters for this method are:

- Constraints--Constraints for the assembly component. These constraints are explained in detail in the later sections.
- ReferenceAssembly--The path to the owner assembly, if the constraints have external references to other members of the top level assembly. If the constraints are applied only to the assembly component then the value of this parameter should be null.

This method modifies the component feature data but does not regenerate the assembly component. To regenerate the assembly use the method **IpfcSolid.Regenerate()**.

## Constraint Attributes

Methods and Properties Introduced:

- **CCpfcConstraintAttributes.Create()**
- **IpfcConstraintAttributes.Force**
- **IpfcConstraintAttributes.Ignore**

The method **CCpfcConstraintAttributes.Create()** returns the constraint attributes object based on the values of the following input parameters:

- Ignore--Constraint is ignored during regeneration. Use this capability to store extra constraints on the component, which allows you to quickly toggle between different constraints.
- Force--Constraint has to be forced for line and point alignment.
- None--No constraint attributes. This is the default value.

## Assembling a Component Parametrically

You can position a component relative to its neighbors (components or assembly features) so that its position is updated as its neighbors move or change. This is called parametric assembly. Pro/ENGINEER allows you to specify constraints to determine how and where the component relates to the assembly. You can add as many constraints as you need to make sure that the assembly meets the design intent.

Methods and Properties Introduced:

- **CCpfcComponentConstraint.Create()**
- **IpfcComponentConstraint.Type**
- **IpfcComponentConstraint.AssemblyReference**
- **IpfcComponentConstraint.AssemblyDatumSide**
- **IpfcComponentConstraint.ComponentReference**
- **IpfcComponentConstraint.ComponentDatumSide**
- **IpfcComponentConstraint.Offset**
- **IpfcComponentConstraint.Attributes**
- **IpfcComponentConstraint.UserDefinedData**

The method **CCpfcComponentConstraint.Create()** returns the component constraint object having the following parameters:

- ComponentConstraintType--Using the TYPE options, you can specify the placement constraint types. They are as follows:
  - EpfcASM\_CONSTRAINT\_MATE--Use this option to make two surfaces touch one another, that is coincident and facing each other.
  - EpfcASM\_CONSTRAINT\_MATE\_OFF--Use this option to make two planar surfaces parallel and facing each other.

- EpfcASM\_CONSTRAINT\_ALIGN--Use this option to make two planes coplanar, two axes coaxial and two points coincident. You can also align revolved surfaces or edges.
- EpfcASM\_CONSTRAINT\_ALIGN\_OFF--Use this option to align two planar surfaces at an offset.
- EpfcASM\_CONSTRAINT\_INSERT--Use this option to insert a ``male" revolved surface into a ``female" revolved surface, making their respective axes coaxial.
- EpfcASM\_CONSTRAINT\_ORIENT--Use this option to make two planar surfaces to be parallel in the same direction.
- EpfcASM\_CONSTRAINT\_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.
- EpfcASM\_CONSTRAINT\_TANGENT----Use this option to control the contact of two surfaces at their tangents.
- EpfcASM\_CONSTRAINT\_PNT\_ON\_SRF--Use this option to control the contact of a surface with a point.
- EpfcASM\_CONSTRAINT\_EDGE\_ON\_SRF--Use this option to control the contact of a surface with a straight edge.
- EpfcASM\_CONSTRAINT\_DEF\_PLACEMENT--Use this option to align the default coordinate system of the component to the default coordinate system of the assembly.
- EpfcASM\_CONSTRAINT\_SUBSTITUTE--Use this option in simplified representations when a component has been substituted with some other model
- EpfcASM\_CONSTRAINT\_PNT\_ON\_LINE--Use this option to control the contact of a line with a point.
- EpfcASM\_CONSTRAINT\_FIX--Use this option to force the component to remain in its current packaged position.
- EpfcASM\_CONSTRAINT\_AUTO--Use this option in the user interface to allow an automatic choice of constraint type based upon the references.
- o AssemblyReference--A reference in the assembly.
- o AssemblyDatumSide--Orientation of the assembly. This can have the following values:
  - Yellow--The primary side of the datum plane which is the default direction of the arrow.
  - Red--The secondary side of the datum plane which is the direction opposite to that of the arrow.
- o ComponentReference--A reference on the placed component.
- o ComponentDatumSide--Orientation of the assembly component. This can have the following values:
  - Yellow--The primary side of the datum plane which is the default direction of the arrow.
  - Red--The secondary side of the datum plane which is the direction opposite to that of the arrow.
- o Offset--The mate or align offset value from the reference.
- o Attributes--Constraint attributes for a given constraint
- o UserDefinedData--A string that specifies user data for the given constraint.

Use the properties listed above to access the parameters of the component constraint object.

## Redefining and Rerouting Assembly Components

These functions enable you to reroute previously assembled components, just as in an interactive Pro/ENGINEER session.

Methods Introduced:

- **IpfcComponentFeat.RedefineThroughUI()**
- **IpfcComponentFeat.MoveThroughUI()**

The method **IpfcComponentFeat.RedefineThroughUI()** must be used in interactive VB applications. This method displays the Pro/ENGINEER Constraint dialog box. This enables the end user to redefine the constraints interactively. The control returns to the VB API application when the user selects **OK** or **Cancel** and the dialog box is closed.

The method **IpfcComponentFeat.MoveThroughUI()** invokes a dialog box that prompts the user to interactively reposition the components. This interface enables the user to specify the translation and rotation values. The control returns to the VB API application when the user selects **OK** or **Cancel** and the dialog box is closed.

**Example: Component Constraints**

This function displays each constraint of the component visually on the screen, and includes a text explanation for each constraint.

```
Public Sub highlightConstraints(ByRef session As IpfcBaseSession)

    Dim model As IpfcModel
    Dim assembly As IpfcAssembly
    Dim options As IpfcSelectionOptions
    Dim selections As IpfcSelections
    Dim item As IpfcModelItem
    Dim feature As IpfcFeature
    Dim asmComp As IpfcComponentFeat
    Dim compConstraints As CpfcComponentConstraints
    Dim i As Integer
    Dim compConstraint As IpfcComponentConstraint
    Dim asmReference As IpfcSelection
    Dim compReference As IpfcSelection
    Dim offset As String
    Dim constraintType As String

    Try

'=====
'Get the current assembly
'=====
        model = session.CurrentModel
        If model Is Nothing Then
            Throw New Exception("Model not present")
        End If
        If (Not model.Type = EpfcModelType.EpfcMDL_ASSEMBLY) Then
            Throw New Exception("Model is not an assembly")
        End If
        assembly = CType(model, IpfcAssembly)
'=====
'Get constraints for the component
'=====
        options = (New CCpfcSelectionOptions).Create("membfeat")
        options.MaxNumSels = 1

        selections = session.Select(options, Nothing)
        If selections Is Nothing OrElse selections.Count = 0 Then
            Throw New Exception("Nothing Selected")
        End If

        item = selections.Item(0).SelItem
        feature = CType(item, IpfcFeature)

        If Not feature.FeatType = EpfcFeatureType.EpfcFEATTYPE_COMPONENT Then
            Throw New Exception("Component not Selected")
        End If

        asmComp = CType(item, IpfcComponentFeat)
        compConstraints = asmComp.GetConstraints()
        If compConstraints Is Nothing OrElse compConstraints.Count = 0 Then
            Throw New Exception("No Constraints to display")
        End If
'=====
    End Try
End Sub
```

```

'Loop through all the constraints
'=====
    For i = 0 To compConstraints.Count - 1

        compConstraint = compConstraints.Item(i)
'=====
'Highlight the assembly reference geometry
'=====
        asmReference = compConstraint.AssemblyReference
        If Not asmReference Is Nothing Then
            asmReference.Highlight(EpfcStdColor.EpfcCOLOR_ERROR)
        End If
'=====
'Highlight the component reference geometry
'=====
        compReference = compConstraint.ComponentReference
        If Not asmReference Is Nothing Then
            compReference.Highlight(EpfcStdColor.EpfcCOLOR_WARNING)
        End If
'=====
'Prepare and display the message text
'=====
        offset = ""
        If Not compConstraint.Offset Is Nothing Then
            offset = ", offset of " + compConstraint.Offset.ToString
        End If
        constraintType = constraintTypeToString(compConstraint.Type)
        MsgBox("Showing constraint " + (i + 1).ToString + " of " +
            _compConstraints.Count.ToString + Chr(13).ToString
            + _ constraintType + offset)
'=====
'Clean up the UI for the next constraint
'=====
        If Not asmReference Is Nothing Then
            asmReference.UnHighlight()
        End If

        If Not compReference Is Nothing Then
            compReference.UnHighlight()
        End If
    Next

    Catch ex As Exception
        MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
    Exit Sub
End Try

End Sub
'=====
'Function      :   constraintTypeToString
'Purpose      :   This function converts constraint type to string.
'=====
Private Function constraintTypeToString(ByVal type As Integer) As String

    Select Case (type)
        Case EpfcComponentConstraintType.EpfcASM_CONSTRAINT_MATE
            Return ("(Mate)")
        Case EpfcComponentConstraintType.EpfcASM_CONSTRAINT_MATE_OFF
            Return ("(Mate Offset)")
    End Select
End Function

```

```

Case EpfcComponentConstraintType.EpfcASM_CONSTRAINT_ALIGN
    Return ("Align")
Case EpfcComponentConstraintType.EpfcASM_CONSTRAINT_ALIGN_OFF
    Return ("Align Offset")
Case EpfcComponentConstraintType.EpfcASM_CONSTRAINT_INSERT
    Return ("Insert")
Case EpfcComponentConstraintType.EpfcASM_CONSTRAINT_ORIENT
    Return ("Orient")
Case EpfcComponentConstraintType.EpfcASM_CONSTRAINT_CSYS
    Return ("Csys")
Case EpfcComponentConstraintType.EpfcASM_CONSTRAINT_TANGENT
    Return ("Tangent")
Case EpfcComponentConstraintType.EpfcASM_CONSTRAINT_PNT_ON_SRF
    Return ("Point on Surf")
Case EpfcComponentConstraintType.EpfcASM_CONSTRAINT_EDGE_ON_SRF
    Return ("Edge on Surf")
Case EpfcComponentConstraintType.EpfcASM_CONSTRAINT_DEF_PLACEMENT
    Return ("Default")
Case EpfcComponentConstraintType.EpfcASM_CONSTRAINT_SUBSTITUTE
    Return ("Substitute")
Case EpfcComponentConstraintType.EpfcASM_CONSTRAINT_PNT_ON_LINE
    Return ("Point on Line")
Case EpfcComponentConstraintType.EpfcASM_CONSTRAINT_FIX
    Return ("Fix")
Case EpfcComponentConstraintType.EpfcASM_CONSTRAINT_AUTO
    Return ("Auto")
End Select
Return ("Unrecognized Type")
End Function

```

#### **Example: 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.

```

Public Sub assembleByDatums(ByRef session As IpfcBaseSession, _
                            ByVal componentFileName As String, _
                            ByVal assemblyDatums() As String, _
                            ByVal componentDatums() As String)

    Dim model As IpfcModel
    Dim assembly As IpfcAssembly
    Dim modelDesc As IpfcModelDescriptor
    Dim componentModel As IpfcSolid
    Dim asmcomp As IpfcComponentFeat
    Dim constraints As IpfcComponentConstraints
    Dim i As Integer
    Dim asmItem As IpfcModelItem
    Dim compItem As IpfcModelItem
    Dim ids As Cintseq
    Dim path As IpfcComponentPath
    Dim asmSelect As IpfcSelection

```

```

Dim compSelect As IpfcSelection
Dim constraint As IpfcComponentConstraint
Dim errorCount As Integer

Try
'=====
'Get the current assembly and new component
'=====
    model = session.CurrentModel
    If model Is Nothing Then
        Throw New Exception("Model not present")
    End If
    If (Not model.Type = EpfcModelType.EpfcMDL_ASSEMBLY) Then
        Throw New Exception("Model is not an assembly")
    End If
    assembly = CType(model, IpfcAssembly)

    modelDesc = (New CCpfcModelDescriptor).CreateFromFileName
(componentFileName)
    componentModel = session.GetModelFromDescr(modelDesc)
    If componentModel Is Nothing Then
        componentModel = session.RetrieveModel(modelDesc)
    End If
'=====
'Package the component initially
'=====
    asmcomp = assembly.AssembleComponent(componentModel,
                                         nothing)
'=====
'Prepare the constraints array
'=====
    errorCount = 0
    constraints = New CpfcComponentConstraints
    For i = 0 To 2
'=====
'Find the assembly datum
'=====
        asmItem = assembly.GetItemByName(EpfcModelItemType.EpfcITEM_SURFACE,
                                         _assemblyDatums(i))
        If asmItem Is Nothing Then
            errorCount = errorCount + 1
            Continue For
        End If
'=====
'Find the component datum
'=====
        compItem = componentModel.GetItemByName(EpfcModelItemType.
EpfcITEM_SURFACE,
                                         _componentDatums(i))

        If compItem Is Nothing Then
            errorCount = errorCount + 1
            Continue For
        End If
'=====
'For the assembly reference, initialize a component path.
'This is necessary even if the reference geometry is in the assembly
'=====
    ids = New Cintseq

```

```

        path = (New CMpfcAssembly).CreateComponentPath(assembly,
            ids)
'=====  

'Allocate the references  

'=====  

        asmSelect = (New CMpfcSelect).CreateModelItemSelection(asmItem, path)  

        compSelect = (New CMpfcSelect).CreateModelItemSelection(compItem,  

Nothing)  

'=====  

'Allocate and fill the constraint  

'=====  

        constraint = (New CCpfcComponentConstraint).Create  

        _(EpfComponentConstraintType.EpfcASM_CONSTRAINT_ALIGN)  

        constraint.AssemblyReference = asmSelect  

        constraint.ComponentReference = compSelect  

        constraints.Insert(constraints.Count, constraint)

        Next
'=====  

'Set the assembly component constraints and regenerate the assembly if  

'at least one constraint has been defined properly  

'=====  

        If errorCount < 2 Then  

            asmcomp.SetConstraints(constraints, Nothing)  

            assembly.Regenerate(Nothing)  

            session.GetModelWindow(assembly).Repaint()  

        End If
'=====  

'If any of the expect datums was not found, prompt the user to constrain  

'the new component  

'=====  

        If errorCount > 0 Then  

            MsgBox("Unable to locate all required datum references." +  

                -  

                "New component is packaged")  

            asmcomp.RedefineThroughUI()  

        End If

        Catch ex As Exception  

            MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)  

            Exit Sub  

        End Try
    End Sub

```

## Exploded Assemblies

These methods enable you to determine and change the explode status of the assembly object.

Methods and Properties Introduced:

- **IpfcAssembly.IsExploded**
- **IpfcAssembly.Explode()**
- **IpfcAssembly.UnExplode()**
- **IpfcAssembly.GetActiveExplodedState()**

- **IpfcAssembly.GetDefaultExplodedState()**
- **IpfcExplodedState.Activate()**

The methods **IpfcAssembly.Explode()** and **IpfcAssembly.UnExplode()** enable you to determine and change the explode status of the assembly object.

The property **IpfcAssembly.IsExploded** reports whether the specified assembly is currently exploded.

The method **IpfcAssembly.GetActiveExplodedState()** returns the current active explode state.

The method **IpfcAssembly.GetDefaultExplodedState()** returns the default explode state.

The method **IpfcExplodedState.Activate()** activates the specified explode state representation.

## 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.

Methods and Properties Introduced:

- **IpfcAssembly.AssembleSkeleton()**
- **IpfcAssembly.AssembleSkeletonByCopy()**
- **IpfcAssembly.GetSkeleton()**
- **IpfcAssembly.DeleteSkeleton()**
- **IpfcSolid.IsSkeleton**

The method **IpfcAssembly.AssembleSkeleton()** adds an existing skeleton model to the specified assembly.

The method **IpfcAssembly.GetSkeleton()** returns the skeleton model of the specified assembly.

The method **IpfcAssembly.DeleteSkeleton()** deletes a skeleton model component from the specified assembly.

The method **IpfcAssembly.AssembleSkeletonByCopy()** adds a specified skeleton model to the assembly. The input parameters for this method are:

- SkeletonToCopy--Specify the skeleton model to be copied into the assembly
- NewSkeletonName--Specify a name for the copied skeleton model

The property **IpfcSolid.IsSkeleton** determines if the specified part model is a skeleton model or a concept model. It returns a true if the model is a skeleton else it returns a false.

---

# Family Tables

---

This section describes how to use the VB API classes and methods to access and manipulate family table information.

## Topic

[Working with Family Tables](#)

[Creating Family Table Instances](#)

[Creating Family Table Columns](#)

## Working with Family Tables

The VB API provides several methods for accessing family table information. Because every model inherits from the interface `IpfcFamilyMember`, every model can have a family table associated with it.

## Accessing Instances

Methods and Properties Introduced:

- **IpfcFamilyMember.Parent**
- **IpfcFamilyMember.GetImmediateGenericInfo()**
- **IpfcFamilyTableRow.CreateInstance()**
- **IpfcFamilyMember.ListRows()**
- **IpfcFamilyMember.GetRow()**
- **IpfcFamilyMember.RemoveRow()**
- **IpfcFamilyTableRow.InstanceName**
- **IpfcFamilyTableRow.IsLocked**

To get the generic model for an instance call the property **IpfcFamilyMember.Parent**.

From Pro/ENGINEER Wildfire 4.0 onwards, the behavior of the method **IpfcFamilyMember.Parent** has changed as a result of performance improvement in family table retrieval mechanism. When you now call the method **pfcFamily.FamilyMember.GetParent**, it throws an exception `pfcExceptions.XToolkitCantOpen`, if the immediate generic of a model instance in a nested family table is currently not in session. Handle this exception and use the method **IpfcFamilyMember.GetImmediateGenericInfo()** to get the model descriptor of the immediate generic model. This information can be used to retrieve the immediate generic model.

If you wish to switch off the above behavior and continue to run legacy applications in the pre-Wildfire 4.0 mode, set the configuration option `retrieve_instance_dependencies` to "instance\_and\_generic\_deps".

The method **IpfcFamilyTableRow.CreateInstance()** returns an instance model created from the information stored

in the `IpfcFamilyTableRow` object.

The method **`IpfcFamilyMember.ListRows()`** returns a sequence of all rows in the family table, whereas **`IpfcFamilyMember.GetRow()`** gets the row object with the name you specify.

Use the method **`IpfcFamilyMember.RemoveRow()`** to permanently delete the row from the family table.

The property **`IpfcFamilyTableRow.InstanceName`** returns the name that corresponds to the invoking row object.

To control whether the instance can be changed or removed, call the property **`IpfcFamilyTableRow.IsLocked`**.

## Accessing Columns

Methods and Properties Introduced:

- **`IpfcFamilyMember.ListColumns()`**
- **`IpfcFamilyMember.GetColumn()`**
- **`IpfcFamilyMember.RemoveColumn()`**
- **`IpfcFamilyTableColumn.Symbol`**
- **`IpfcFamilyTableColumn.Type`**
- **`IpfcFamColModelItem.RefItem`**
- **`IpfcFamColParam.RefParam`**

The method **`IpfcFamilyMember.ListColumns()`** returns a sequence of all columns in the family table.

The method **`IpfcFamilyMember.GetColumn()`** returns a family table column, given its symbolic name.

To permanently delete the column from the family table and all changed values in all instances, call the method **`IpfcFamilyMember.RemoveColumn()`**.

The property **`IpfcFamilyTableColumn.Symbol`** returns the string symbol at the top of the column, such as *D4* or *F5*.

The property **`IpfcFamilyTableColumn.Type`** returns an enumerated value indicating the type of parameter governed by the column in the family table.

The property **`IpfcFamColModelItem.RefItem`** returns the `IModelItem` (`Feature` or `Dimension`) controlled by the column, whereas **`IpfcFamColParam.RefParam`** returns the `Parameter` controlled by the column.

## Accessing Cell Information

Methods and Properties Introduced:

- **`IpfcFamilyMember.GetCell()`**
- **`IpfcFamilyMember.GetCellsDefault()`**

- **IpfcFamilyMember.SetCell()**
- **IpfcParamValue.StringValue**
- **IpfcParamValue.IntValue**
- **IpfcParamValue.DoubleValue**
- **IpfcParamValue.BoolValue**

The method **IpfcFamilyMember.GetCell()** returns a string `IPParamValue` that corresponds to the cell at the intersection of the row and column arguments. Use the method **IpfcFamilyMember.GetCellIsDefault()** to check if the value of the specified cell is the default value, which is the value of the specified cell in the generic model.

The method **IpfcFamilyMember.SetCell()** assigns a value to a column in a particular family table instance.

The **IpfcParamValue.StringValue**, **IpfcParamValue.IntValue**, **IpfcParamValue.DoubleValue**, and **IpfcParamValue.BoolValue** properties are used to get the different types of parameter values.

## Creating Family Table Instances

Methods Introduced:

- **IpfcFamilyMember.AddRow()**
- **CMpfcModelItem.CreateStringParamValue()**
- **CMpfcModelItem.CreateIntParamValue()**
- **CMpfcModelItem.CreateDoubleParamValue()**
- **CMpfcModelItem.CreateBoolParamValue()**

Use the method **IpfcFamilyMember.AddRow()** to create a new instance with the specified name, and, optionally, the specified values for each column. If you do not pass in a set of values, the value "\*" will be assigned to each column. This value indicates that the instance uses the generic value.

## Creating Family Table Columns

Methods Introduced:

- **IpfcFamilyMember.CreateDimensionColumn()**
- **IpfcFamilyMember.CreateParamColumn()**
- **IpfcFamilyMember.CreateFeatureColumn()**
- **IpfcFamilyMember.CreateComponentColumn()**
- **IpfcFamilyMember.CreateCompModelColumn()**
- **IpfcFamilyMember.CreateGroupColumn()**

- **IpfcFamilyMember.CreateMergePartColumn()**
- **IpfcFamilyMember.CreateColumn()**
- **IpfcFamilyMember.AddColumn()**
- **CMpfcModelItem.CreateStringParamValue()**

The **IpfcFamilyMember.CreateParamColumn()** methods initialize a column based on the input argument. These methods assign the proper symbol to the column header.

The method **IpfcFamilyMember.CreateColumn()** creates a new column given a properly defined symbol and column type. The results of this call should be passed to the method **IpfcFamilyMember.AddColumn()** to add the column to the model's family table.

The method **IpfcFamilyMember.AddColumn()** adds the column to the family table. You can specify the values; if you pass nothing for the values, the method assigns the value "\*" to each instance to accept the column's default value.

#### **Example Code: Adding Dimensions to a Family Table**

This function adds all the dimensions to a family table. The program lists the dependencies of the assembly and loops through each dependency, assigning the model to a new FamColDimension column object. All the dimensions, parameters, features, and components could be added to the family table using a similar method.

```
Imports pfcls
Public Class pfcFamilyTablesExamples

    Public Sub addHoleDiameterColumns(ByRef session As IpfcBaseSession)

        Dim model As IpfcModel
        Dim solid As IpfcSolid
        Dim holeFeatures As IpfcFeatures
        Dim holeFeature As IpfcFeature
        Dim dimensions As IpfcModelItems
        Dim dimension As IpfcDimension
        Dim dimensionColumn As IpfcFamColDimension
        Dim i, j As Integer

        Try
            '=====
            'Get the current assembly and new component
            '=====
            model = session.CurrentModel
            If model Is Nothing Then
                Throw New Exception("Model not present")
            End If
            If (Not model.Type = EpfcModelType.EpfcMDL_PART) And _
                (Not model.Type = EpfcModelType.EpfcMDL_ASSEMBLY) Then
                Throw New Exception("Model is not a solid")
            End If
            solid = CType(model, IpfcSolid)
```

```

'=====  

'List all holes in the solid model  

'=====  

holeFeatures = solid.ListFeaturesByType _  

                (True, EpfcFeatureType.EpfcFEATTYPE_HOLE)  

For i = 0 To holeFeatures.Count - 1  

    holeFeature = holeFeatures.Item(i)  

    '=====  

    'List all dimensions in the feature  

    '=====  

    dimensions = holeFeature.ListSubItems _  

                (EpfcModelItemType.EpfcITEM_DIMENSION)  

    For j = 0 To dimensions.Count - 1  

        dimension = dimensions.Item(j)  

        '=====  

        'Determine if dimension is diameter type  

        '=====  

        If dimension.DimType = EpfcDimensionType.EpfcDIM_DIAMETER Then  

            '=====  

            'Create family table column  

            '=====  

            dimensionColumn = solid.CreateDimensionColumn(dimension)  

            '=====  

            'Add the column to the Solid.  

            'Instead of null, any array of ParamValues can be passed  

            'for the initial column values  

            '=====  

            solid.AddColumn(dimensionColumn, Nothing)  

            End If  

        Next  

    Next  

    Catch ex As Exception  

        MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)  

        Exit Sub  

    End Try  

End Sub  

End Class

```

---

# Action Listeners

---

This section describes the VB API methods that enable you to use action listeners.

## Topic

[The VB API Action Listeners](#)

[Action Sources](#)

[Types of Action Listeners](#)

[Cancelling an ActionListener Operation](#)

## The VB API Action Listeners

An `ActionListener` is a class that is assigned to respond to certain events. In the VB API, you can assign action listeners to respond to events involving the following tasks:

- Changing windows
- Changing working directories
- Model operations
- Regenerating
- Creating, deleting, and redefining features
- Checking for regeneration failures

All action listeners in the VB API are defined by these classes:

- Interface--Named `<Object>ActionListener`. This interface defines the methods that can respond to various events.
- Default class--Named `Default<Object>ActionListener`. This class has every available method overridden by an empty implementation. You create your own action listeners by extending the default class and overriding the methods for events that interest you.

## Action Sources

Methods introduced:

- **IpfcActionSource.AddActionListener()**
- **IpfcActionSource.RemoveActionListener()**

Many VB API classes inherit the `IpfcActionSource` interface, but only the following classes currently make calls to the methods of registered `IpfcActionListeners`:

- `IpfcSession`
  - Session Action Listener
  - Model Action Listener
  - Solid Action Listener
  - Model Event Action Listener
  - Feature Action Listener
- `IpfcUICommand`
  - UI Action Listener
- `IpfcModel` (and its subclasses)
  - Model Action Listener
  - Parameter Action Listener
- `IpfcSolid` (and its subclasses)
  - Solid Action Listener
  - Feature Action Listener
- `IpfcFeature` (and its subclasses)
  - Feature Action Listener

**Note:**

Assigning an action listener to a source not related to it will not cause an error but the listener method will never be called.

## Types of Action Listeners

The following sections describe the different kinds of action listeners: session, UI command, solid, and feature.

### Session Level Action Listeners

Methods introduced:

- **IpfcSessionActionListener.OnAfterDirectoryChange()**

- **IpfcSessionActionListener.OnAfterWindowChange()**
- **IpfcSessionActionListener.OnAfterModelDisplay()**
- **IpfcSessionActionListener.OnBeforeModelErase()**
- **IpfcSessionActionListener.OnBeforeModelDelete()**
- **IpfcSessionActionListener.OnBeforeModelRename()**
- **IpfcSessionActionListener.OnBeforeModelSave()**
- **IpfcSessionActionListener.OnBeforeModelPurge()**
- **IpfcSessionActionListener.OnBeforeModelCopy()**
- **IpfcSessionActionListener.OnAfterModelPurge()**

The **IpfcSessionActionListener.OnAfterDirectoryChange()** method activates after the user changes the working directory. This method takes the new directory path as an argument.

The **IpfcSessionActionListener.OnAfterWindowChange()** method activates when the user activates a window other than the current one. Pass the new window to the method as an argument.

The **IpfcSessionActionListener.OnAfterModelDisplay()** method activates every time a model is displayed in a window.

**Note:**

Model display events happen when windows are moved, opened and closed, repainted, or the model is regenerated. The event can occur more than once in succession.

The methods **pfcSession.SessionActionListener.OnBeforeModelErase**, **pfcSession.SessionActionListener.OnBeforeModelRename**, **pfcSession.SessionActionListener.OnBeforeModelSave**, and **IpfcSessionActionListener.OnBeforeModelCopy()** take special arguments. They are designed to allow you to fill in the arguments and pass this data back to Pro/ENGINEER. The model names placed in the descriptors will be used by Pro/ENGINEER as the default names in the user interface.

# UI Command Action Listeners

Methods introduced:

- **IpfcSession.UICreateCommand()**
- **IpfcUICommandActionListener.OnCommand()**

The **IpfcSession.UICreateCommand()** method takes a `IpfcUICommandActionListener` argument and returns a `IpfcUICommand` action source with that action listener already registered. This `UICommand` object is subsequently passed as an argument to the **Session.AddUIButton** method that adds a command button to a Pro/ENGINEER menu. The **IpfcUICommandActionListener.OnCommand()** method of the registered `IpfcUICommandActionListener` is called whenever the command button is clicked.

# Model Level Action listeners

Methods introduced:

- **IpfcModelActionListener.OnAfterModelSave()**
- **IpfcModelEventActionListener.OnAfterModelCopy()**
- **IpfcModelEventActionListener.OnAfterModelRename()**
- **IpfcModelEventActionListener.OnAfterModelErase()**
- **IpfcModelEventActionListener.OnAfterModelDelete()**
- **IpfcModelActionListener.OnAfterModelRetrieve()**
- **IpfcModelActionListener.OnBeforeModelDisplay()**
- **IpfcModelActionListener.OnAfterModelCreate()**

- **IpfcModelActionListener.OnAfterModelSaveAll()**
- **IpfcModelEventActionListener.OnAfterModelCopyAll()**
- **IpfcModelActionListener.OnAfterModelEraseAll()**
- **IpfcModelActionListener.OnAfterModelDeleteAll()**
- **IpfcModelActionListener.OnAfterModelRetrieveAll()**

Methods ending in All are called after any event of the specified type. The call is made even if the user did not explicitly request that the action take place. Methods that **do not** end in All are only called when the user specifically requests that the event occurs.

The method `IpfcModelActionListener.OnAfterModelSave()` is called after successfully saving a model.

The method `IpfcModelEventActionListener.OnAfterModelCopy()` is called after successfully copying a model.

The method `IpfcModelEventActionListener.OnAfterModelRename()` is called after successfully renaming a model.

The method `IpfcModelEventActionListener.OnAfterModelErase()` is called after successfully erasing a model.

The method `IpfcModelEventActionListener.OnAfterModelDelete()` is called after successfully deleting a model.

The method `IpfcModelActionListener.OnAfterModelRetrieve()` is called after successfully retrieving a model.

The method `IpfcModelActionListener.OnBeforeModelDisplay()` is called before displaying a model.

The method **`IpfcModelActionListener.OnAfterModelCreate()`** is called after the successful creation of a model.

## **Solid Level Action Listeners**

Methods introduced:

- **IpfcSolidActionListener.OnBeforeRegen()**
- **IpfcSolidActionListener.OnAfterRegen()**
- **IpfcSolidActionListener.OnBeforeUnitConvert()**
- **IpfcSolidActionListener.OnAfterUnitConvert()**
- **IpfcSolidActionListener.OnBeforeFeatureCreate()**
- **IpfcSolidActionListener.OnAfterFeatureCreate()**
- **IpfcSolidActionListener.OnAfterFeatureDelete()**

The **IpfcSolidActionListener.OnBeforeRegen()** and **IpfcSolidActionListener.OnAfterRegen()** methods occur when the user regenerates a solid object within the `IpfcActionSource` to which the listener is assigned. These methods take the first feature to be regenerated and a handle to the `IpfcSolid` object as arguments. In addition, the method **IpfcSolidActionListener.OnAfterRegenerate** includes a Boolean argument that indicates whether regeneration was successful.

**Note:**

- It is not recommended to modify geometry or dimensions using the `IpfcSolidActionListener.OnBeforeRegenerate` method call.
- A regeneration that did not take place because nothing was modified is identified as a regeneration failure.

The **IpfcSolidActionListener.OnBeforeUnitConvert()** and **IpfcSolidActionListener.OnAfterUnitConvert()** methods activate when a user modifies the unit scheme (by selecting the Pro/ENGINEER command **Set Up, Units**). The methods receive the `Solid` object to be converted and a Boolean flag that identifies whether the conversion changed the dimension values to keep the object the same size.

**Note:**

`IpfcSolidActionListeners` can be registered with the session object so that its

methods are called when these events occur for any solid model that is in session.

The **`IpfcSolidActionListener.OnBeforeFeatureCreate()`** method activates when the user starts to create a feature that requires the Feature Creation dialog box. Because this event occurs only after the dialog box is displayed, it will not occur at all for datums and other features that do not use this dialog box. This method takes two arguments: the solid model that will contain the feature and the `IpfcModelItem` identifier.

The **`IpfcSolidActionListener.OnAfterFeatureCreate()`** method activates after any feature, including datums, has been created. This method takes the new `IpfcFeature` object as an argument.

The **`IpfcSolidActionListener.OnAfterFeatureDelete()`** method activates after any feature has been deleted. The method receives the solid that contained the feature and the (now defunct) `IpfcModelItem` identifier.

## Feature Level Action Listeners

Methods introduced:

- **`IpfcFeatureActionListener.OnBeforeDelete()`**
- **`IpfcFeatureActionListener.OnBeforeSuppress()`**
- **`IpfcFeatureActionListener.OnAfterSuppress()`**
- **`IpfcFeatureActionListener.OnBeforeRegen()`**
- **`IpfcFeatureActionListener.OnAfterRegen()`**
- **`IpfcFeatureActionListener.OnRegenFailure()`**
- **`IpfcFeatureActionListener.OnBeforeRedefine()`**
- **`IpfcFeatureActionListener.OnAfterCopy()`**
- **`IpfcFeatureActionListener.OnBeforeParameterDelete()`**

Each method in `IpfcFeatureActionListener` takes as an argument the feature that triggered the event.

`IpfcFeatureActionListeners` can be registered with the `Session` object so that the action listener's methods are called whenever these events occur for any feature that is in session or with a solid model to react to changes only in that model.

The method **`IpfcFeatureActionListener.OnBeforeDelete()`** is called before a feature is deleted.

The method **`IpfcFeatureActionListener.OnBeforeSuppress()`** is called before a feature is suppressed.

The method **`IpfcFeatureActionListener.OnAfterSuppress()`** is called after a successful feature suppression.

The method **`IpfcFeatureActionListener.OnBeforeRegen()`** is called before a feature is regenerated.

The method **`IpfcFeatureActionListener.OnAfterRegen()`** is called after a successful feature regeneration.

The method **`IpfcFeatureActionListener.OnRegenFailure()`** is called when a feature fails regeneration.

The method **`IpfcFeatureActionListener.OnBeforeRedefine()`** is called before a feature is redefined.

The method **`IpfcFeatureActionListener.OnAfterCopy()`** is called after a feature has been successfully copied.

The method **`IpfcFeatureActionListener.OnBeforeParameterDelete()`** is called before a feature parameter is deleted.

## Cancelling an ActionListener Operation

The VB API allows you to cancel certain notification events, registered by the action listeners.

Methods Introduced:

## • **CCpfcXCancelProEAction.Throw()**

The static method **CCpfcXCancelProEAction.Throw()** must be called from the body of an action listener to cancel the impending Pro/ENGINEER operation. This method will throw a The VB API exception signalling to Pro/ENGINEER to cancel the listener event.

Note: Your application should not catch the The VB API exception, or should rethrow it if caught, so that Pro/ENGINEER is forced to handle it.

The following events can be cancelled using this technique:

- IpfcSessionActionListener.OnBeforeModelErase()
  - IpfcSessionActionListener.OnBeforeModelDelete()
  - IpfcSessionActionListener.OnBeforeModelRename()
  - IpfcSessionActionListener.OnBeforeModelSave()
  - IpfcSessionActionListener.OnBeforeModelPurge()
  - IpfcSessionActionListener.OnBeforeModelCopy()
  - IpfcModelActionListener.OnBeforeParameterCreate()
  - IpfcModelActionListener.OnBeforeParameterDelete()
  - IpfcModelActionListener.OnBeforeParameterModify()
  - IpfcFeatureActionListener.OnBeforeDelete()
  - IpfcFeatureActionListener.OnBeforeSuppress()
  - IpfcFeatureActionListener.OnBeforeParameterDelete()
  - IpfcFeatureActionListener.OnBeforeParameterCreate()
  - IpfcFeatureActionListener.OnBeforeRedefine()
-

# Interface

---

This section describes various methods of importing and exporting files in the VB API.

## Topic

[Exporting Files](#)

[Exporting 3D Geometry](#)

[Shrinkwrap Export](#)

[Importing Files](#)

[Importing 3D Geometry](#)

[Plotting Files](#)

[Solid Operations](#)

[Window Operations](#)

## Exporting Files

Method Introduced:

- **IpfcModel.Export()**

The method **IpfcModel.Export()** exports a file from Pro/ENGINEER onto a disk. The input parameters are:

- filename--Output file name including extensions
- exportdata--An export instructions object that controls the export operation.

There are four general categories of files to which you can export models:

- File types whose instructions inherit from IpfcGeomExportInstructions.

These instructions export files that contain precise geometric information used by other CAD systems.

- File types whose instructions inherit from IpfcCoordSysExportInstructions.

These instructions export files that contain coordinate information describing faceted, solid models (without datums and surfaces).

- File types whose instructions inherit from IpfcFeatIdExportInstructions.

These instructions export information about a specific feature.

- General file types that inherit only from IpfcExportInstructions.

These instructions provide conversions to file types such as BOM (bill of materials).

For information on exporting to a specific format, see the theVB API browser and online help for the Pro/ENGINEER interface.

## Export Instructions

Methods Introduced:

- **CCpfcRelationExportInstructions.Create()**

- CCpfcModelInfoExportInstructions.Create()
- CCpfcProgramExportInstructions.Create()
- CCpfcIGESFileExportInstructions.Create()
- CCpfcDXFExportInstructions.Create()
- CCpfcRenderExportInstructions.Create()
- CCpfcSTLASCIIExportInstructions.Create()
- CCpfcSTLBinaryExportInstructions.Create()
- CCpfcBOMExportInstructions.Create()
- CCpfcDWGSetupExportInstructions.Create()
- CCpfcFeatInfoExportInstructions.Create()
- CCpfcMFGFeatCLExportInstructions.Create()
- CCpfcMFGOperCLExportInstructions.Create()
- CCpfcMaterialExportInstructions.Create()
- CCpfcCGMFILEExportInstructions.Create()
- CCpfcInventorExportInstructions.Create()
- CCpfcFIATExportInstructions.Create()
- CCpfcConnectorParamExportInstructions.Create()
- CCpfcCableParamsFileInstructions.Create()
- CCpfcCADDSExportInstructions.Create()
- CCpfcSTEP3DExportInstructions.Create()
- CCpfcNEUTRALFileExportInstructions.Create()
- IpfcBaseSession.ExportDirectVRML()

### Export Instructions Table

Interface	Used to Export
IpfcRelationExportInstructions	A list of the relations and parameters in a part or assembly

IpfcModelInfoExportInstructions	Information about a model, including units information, features, and children
IpfcProgramExportInstructions	A program file for a part or assembly that can be edited to change the model
IpfcIGESEExportInstructions	A drawing in IGES format
IpfcDXFExportInstructions	A drawing in DXF format
IpfcRenderExportInstructions	A part or assembly in RENDER format
IpfcSTLASCIIExportInstructions	A part or assembly to an ASCII STL file
IpfcSTLBinaryExportInstructions	A part or assembly in a binary STL file
IpfcBOMExportInstructions	A BOM for an assembly
IpfcDWGSetupExportInstructions	A drawing setup file
IpfcFeatInfoExportInstructions	Information about one feature in a part or assembly
IpfcMfgFeatCLExportInstructions	A cutter location (CL) file for one NC sequence in a manufacturing assembly
IpfcMfgOperCLExportInstructions	A cutter location (CL) file for all the NC sequences in a manufacturing assembly
IpfcMaterialExportInstructions	A material from a part
IpfcCGMFILEEExportInstructions	A drawing in CGM format
IpfcInventorExportInstructions	A part or assembly in Inventor format
IpfcFIATExportInstructions	A part or assembly in FIAT format
IpfcConnectorParamExportInstructions	The parameters of a connector to a text file
IpfcCableParamsFileInstructions	Cable parameters from an assembly
IpfcCATIAFacetsExportInstructions	A part or assembly in CATIA format (as a faceted model)
IpfcVRMLModelExportInstructions	A part or assembly in VRML format

IpfcCADDSExportInstructions	A CADD5 solid model
IpfcSTEP2DExportInstructions	A two-dimensional STEP format file
IpfcNEUTRALFileExportInstructions	A Pro/ENGINEER part to neutral format

The **Instruction Classes** are as follows:

- IpfcIGES3DNewExportInstructions
- IpfcSTEP3DExportInstructions
- IpfcVDA3DExportInstructions
- IpfcSET3DExportInstructions
- IpfcCATIA3DExportInstructions

## Exporting 3D Geometry

theVB API allows you to export three dimensional geometry to various formats. Pass the instructions object containing information about the desired export file to the method **IpfcModel.Export()**.

## Export Instructions

Methods and Properties Introduced:

- **IpfcExport3DInstructions.Configuration**
- **IpfcExport3DInstructions.ReferenceSystem**
- **IpfcExport3DInstructions.Geometry**
- **IpfcExport3DInstructions.IncludedEntities**
- **IpfcExport3DInstructions.LayerOptions**
- **CCpfcGeometryFlags.Create()**
- **CCpfcInclusionFlags.Create()**
- **CCpfcLayerExportOptions.Create()**
- **CCpfcSTEP3DExportInstructions.Create()**
- **CCpfcSET3DExportInstructions.Create()**
- **CCpfcVDA3DExportInstructions.Create()**
- **CCpfcIGES3DNewExportInstructions.Create()**
- **CCpfcCATIA3DExportInstructions.Create()**
- **CCpfcCATIAModel3DExportInstructions.Create()**

- **CCpfcPDGS3DExportInstructions.Create()**
- **CCpfcACIS3DExportInstructions.Create()**

The interface **IpfcExport3DInstructions** contains data to export a part or an assembly to a specified 3D format. The fields of this interface are:

- Configuration--While exporting an assembly you can specify the structure and contents of the output files. The options are:
  - EXPORT\_ASM\_FLAT\_FILE--Exports all the geometry of the assembly to a single file as if it were a part.
  - EXPORT\_ASM\_SINGLE\_FILE--Exports an assembly structure to a file with external references to component files. This file contains only top-level geometry.
  - EXPORT\_ASM\_MULTI\_FILE--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.
  - EXPORT\_ASM\_ASSEMBLY\_FILE--Exports an assembly as multiple files containing geometry information of its components and assembly features.
- ReferenceSystem--The reference coordinate system used for export. If this value is null, the system uses the default coordinate system.
- Geometry--The object describing the type of geometry to export. The CCpfcGeometryFlags.Create() returns this instruction object. The types of geometry supported by the export operation are:
  - Wireframe--Export edges only.
  - Solid--Export surfaces along with topology.
  - Surfaces--Export all model surfaces.
  - Quilts--Export as quilt.
- IncludedEntities--The object returned by the method CCpfcInclusionFlags.Create() that determines whether to include certain entities. The entities are:
  - Datums--Determines whether datum curves are included when exporting files. If true the datum curve information is included during export. The default value is false.
  - Blanked--Determines whether entities on blanked layers are exported. If true entities on blanked layers are exported. The default value is false.
- LayerOptions--The instructions object returned by the method CCpfcLayerExportOptions.Create() that describes how to export layers. To export layers you can specify the following:
  - UseAutoId--Enables you to set or remove an interface layer ID. A layer is recognized with this ID when exporting the file to a specified output format. If true, automatically assigns interface IDs to layers not assigned IDs and exports them. The default value is false.
  - LayerSetupFile--Specifies the name and complete path of the layer setup file. This file contains the layer assignment information which includes the name of the layer, its display status, the interface ID and number of sub layers.

## Export 3D Instructions Table

Interface	Used to Export
IpfcSTEP3DExportInstructions	A part or assembly in STEP format
IpfcVDA3DExportInstructions	A part or assembly in VDA format
IpfcSET3DExportInstructions	A class that defines a ruled surface
IpfcIGES3DNewExportInstructions	A part or assembly in IGES format

IpfcCATIA3DExportInstructions	A part or assembly in CATIA format (as precise geometry)
IpfcCATIAModel3DExportInstructions	A part or assembly in CATIA MODEL format
IpfcACIS3DExportInstructions	A part or assembly in ACIS format
IpfcPDGS3DExportInstructions	A part or assembly in PDGS format

## Export Utilities

Methods Introduced:

- **IpfcBaseSession.IsConfigurationSupported()**
- **IpfcBaseSession.IsGeometryRepSupported()**

The method **IpfcBaseSession.IsConfigurationSupported()** checks whether the specified assembly configuration is valid for a particular model and the specified export format. The input parameters for this method are:

- Configuration--Specifies the structure and content of the output files.
- Type--Specifies the output file type to create.

The method returns a true value if the configuration is supported for the specified export type.

The method **IpfcBaseSession.IsGeometryRepSupported()** checks whether the specified geometric representation is valid for a particular export format. The input parameters are :

- Flags--The type of geometry supported by the export operation.
- Type--The output file type to create.

The method returns a true value if the geometry combination is valid for the specified model and export type.

The methods **IpfcBaseSession.IsConfigurationSupported()** and **IpfcBaseSession.IsGeometryRepSupported()** must be called before exporting an assembly to the specified export formats except for the CADDs and STEP2D formats. The return values of both the methods must be true for the export operation to be successful.

Use the method **IpfcModel.Export()** to export the assembly to the specified output format.

## Shrinkwrap Export

To improve performance in a 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.

Method Introduced:

- **IpfcSolid.ExportShrinkwrap()**

You can export the specified solid model as a shrinkwrap model using the method **IpfcSolid.ExportShrinkwrap()**. This method takes the **ShrinkwrapExportInstruction** object as an argument.

Use the appropriate interface given in the following table to create the required type of shrinkwrap. All the interfaces have their own static method to create an object of the specified type. The object created by these interfaces can be used as an object of type **ShrinkwrapExportInstructions** or **ShrinkwrapModelExportInstructions**.

Type of Shrinkwrap Model	Interface to Use
Surface Subset	IpfcShrinkwrapSurfaceSubsetInstructions
Faceted Part	IpfcShrinkwrapFacetedPartInstructions
Faceted VRML	IpfcShrinkwrapFacetedVRMLInstructions
Faceted STL	IpfcShrinkwrapFacetedSTLInstructions
Merged Solid	IpfcShrinkwrapMergedSolidInstructions

## Setting Shrinkwrap Options

The interface **IpfcShrinkwrapModelExportInstructions** contains the general methods available for all the types of shrinkwrap models. The object created by any of the interfaces specified in the preceding table can be used with these methods.

Properties Introduced:

- **IpfcShrinkwrapModelExportInstructions.Method**
- **IpfcShrinkwrapModelExportInstructions.Quality**
- **IpfcShrinkwrapModelExportInstructions.AutoHoleFilling**
- **IpfcShrinkwrapModelExportInstructions.IgnoreSkeleton**
- **IpfcShrinkwrapModelExportInstructions.IgnoreQuilts**
- **IpfcShrinkwrapModelExportInstructions.AssignMassProperties**
- **IpfcShrinkwrapModelExportInstructions.IgnoreSmallSurfaces**
- **IpfcShrinkwrapModelExportInstructions.SmallSurfPercentage**
- **IpfcShrinkwrapModelExportInstructions.DatumReferences**

The property **IpfcShrinkwrapModelExportInstructions.Method** returns the method used to create the shrinkwrap. The types of shrinkwrap methods are:

- SWCREATE\_SURF\_SUBSET--Surface Subset
- SWCREATE\_FACETED\_SOLID--Faceted Solid
- SWCREATE\_MERGED\_SOLID--Merged Solid

The property **IpfcShrinkwrapModelExportInstructions.Quality** specifies the quality level for the system to use when identifying surfaces or components that 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 1.

The property **IpfcShrinkwrapModelExportInstructions.AutoHoleFilling** 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 property **IpfcShrinkwrapModelExportInstructions.IgnoreSkeleton** determine whether the skeleton model geometry must be included in the shrinkwrap model.

The property **IpfcShrinkwrapModelExportInstructions.IgnoreQuilts** determines whether external quilts must be included in the shrinkwrap model.

The property **IpfcShrinkwrapModelExportInstructions.AssignMassProperties** 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 must assign a value for the mass properties.

The property **IpfcShrinkwrapModelExportInstructions.IgnoreSmallSurfaces** sets a flag that forces Pro/ENGINEER to skip surfaces smaller than a certain size. The default value is false. The size of the surface is specified as a percentage of the model's size. This size can be modified using the property **IpfcShrinkwrapModelExportInstructions.SmallSurfPercentage**.

The property **IpfcShrinkwrapModelExportInstructions.DatumReferences** specifies and selects the datum planes, points, curves, axes, and coordinate system references to be included in the shrinkwrap model.

## Surface Subset Options

Methods and Properties Introduced:

- **CCpfcShrinkwrapSurfaceSubsetInstructions.Create()**
- **IpfcShrinkwrapSurfaceSubsetInstructions.AdditionalSurfaces**
- **IpfcShrinkwrapSurfaceSubsetInstructions.OutputModel**

The static method **CCpfcShrinkwrapSurfaceSubsetInstructions.Create()** returns an object used to create a shrinkwrap model of surface subset type. Specify the name of the output model in which the shrinkwrap is to be created as an input to this method.

The property **IpfcShrinkwrapSurfaceSubsetInstructions.AdditionalSurfaces** selects individual surfaces to be included in the shrinkwrap model.

The property **IpfcShrinkwrapSurfaceSubsetInstructions.OutputModel** returns the template model where the shrinkwrap geometry is to be created.

## Faceted Solid Options

The **IpfcShrinkwrapFacetedFormatInstructions** interface consists of the following types:

- SWFACETED\_PART--Pro/ENGINEER part with normal geometry. This is the default format type.
- SWFACETED\_STL--An STL file.
- SWFACETED\_VRML--A VRML file.

Use the **Create** method to create the object of the specified type. Upcast the object to use the general methods available in this interface.

Properties Introduced:

- **IpfcShrinkwrapFacetedFormatInstructions.Format**
- **IpfcShrinkwrapFacetedFormatInstructions.FramesFile**

The property **IpfcShrinkwrapFacetedFormatInstructions.Format** returns the the output file format of the shrinkwrap model.

The property **IpfcShrinkwrapFacetedFormatInstructions.FramesFile** 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.

## Faceted Part Options

Methods and Properties Introduced:

- **CCpfcShrinkwrapFacetedPartInstructions.Create()**
- **IpfcShrinkwrapFacetedPartInstructions.Lightweight**

The static method **CCpfcShrinkwrapFacetedPartInstructions.Create()** returns an object used to create a shrinkwrap model of shrinkwrap faceted type. The input parameters of this method are:

- OutputModel--Specify the output model where the shrinkwrap must be created.
- Lightweight--Specify this value as True if the shrinkwrap model is a Lightweight Pro/ENGINEER part.

The property **IpfcShrinkwrapFacetedPartInstructions.Lightweight** specifies if the Pro/ENGINEER part is exported as a light weight faceted geometry.

## VRML Export Options

Methods and Properties Introduced:

- **CCpfcShrinkwrapVRMLInstructions.Create()**
- **IpfcShrinkwrapVRMLInstructions.OutputFile**

The static method **CCpfcShrinkwrapVRMLInstructions.Create()** returns an object used to create a shrinkwrap model of shrinkwrap VRML format. Specify the name of the output model as an input to this method.

The property **IpfcShrinkwrapVRMLInstructions.OutputFile** specifies the name of the output file to be created.

## STL Export Options

Methods and Properties Introduced:

- **CCpfcShrinkwrapVRMLInstructions.Create()**
- **IpfcShrinkwrapVRMLInstructions.OutputFile**

The static method **CCpfcShrinkwrapVRMLInstructions.Create()** returns an object used to create a shrinkwrap model of shrinkwrap STL format. Specify the name of the output model as an input to this method.

The property **IpfcShrinkwrapSTLInstructions.OutputFile** specifies the name of the output file to be created.

## Merged Solid Options

Methods and Properties Introduced:

- **CCpfcShrinkwrapMergedSolidInstructions.Create()**
- **IpfcShrinkwrapMergedSolidInstructions.AdditionalComponents**

The static method **CCpfcShrinkwrapMergedSolidInstructions.Create()** returns an object used to create a shrinkwrap model of merged solids format. Specify the name of the output model as an input to this method.

The property **IpfcShrinkwrapMergedSolidInstructions.AdditionalComponents** specifies individual components of the assembly to be merged into the shrinkwrap model.

## VRML Representation

### Example Code

The following example code leverages the fact that when a model with a model program attached is erased or deleted the stop method of the model program is called. This example code uses the stop method to produce a VRML representation of the model in a standard directory for Web publishing.

```
Imports pfcls
Public Class pfcInterfaceExamples1
    Implements IpfcAsyncActionListener
    Implements ICIPClientObject
    Implements IpfcActionListener

    Dim WithEvents eventTimer As Timers.Timer
    Dim exitFlag As Boolean = False
    Dim aC As pfcls.IpfcAsyncConnection

    Public Sub New(ByRef asyncConnection As pfcls.IpfcAsyncConnection)
        aC = asyncConnection
    End Sub

    Public Function GetClientInterfaceName() As String Implements pfcls.
    ICIPClientObject.GetClientInterfaceName
        GetClientInterfaceName = "IpfcAsyncActionListener"
    End Function

    Public Sub OnTerminate(ByVal _Status As Integer) Implements pfcls.
    IpfcAsyncActionListener.OnTerminate
```

```

        aC.InterruptEventProcessing()
        exitFlag = True
    End Sub

'VRML on erase
'=====
'Function      :   createVRMLOnErase
'Purpose       :   This function uses the listener OnBeforeModelErase
'               :   to create VRML file in given directory.
'               :   Note that this operates in Full Asynchronous Mode.
'=====
    Public Sub createVRMLOnErase(ByVal dirPath As String)
        Dim listenerObj As New VRMLEventListener(dirPath)
        Try
'=====
'Start the timer to call EventProcess at regular intervals
'=====
            eventTimer = New Timers.Timer(500)
            eventTimer.Enabled = True
            AddHandler eventTimer.Elapsed, AddressOf Me.timeElapsed

            ac.Session.AddActionListener(listenerObj)
            aC.AddActionListener(Me)

            Catch ex As Exception
                MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
            End Try
        End Sub

'=====
'Function      :   timeElapsed
'Purpose       :   This function handels the time elapsed event of timer
'               :   which is fired at regular intervals
'=====
    Private Sub timeElapsed(ByVal sender As Object, ByVal e As
                            System.Timers.ElapsedEventArgs)
        If exitFlag = False Then
            aC.EventProcess()
        Else
            eventTimer.Enabled = False
        End If
    End Sub

'=====
'Class         :   VRMLEventListener
'Purpose       :   This class must implement the listner interface along
'               :   with the correct client interface name. The implemented
'               :   methods are called after corresponding actions on the
'               :   model.
'=====
    Private Class VRMLEventListener
        Implements IpfcSessionActionListener
        Implements ICIPClientObject
        Implements IpfcActionListener

        Dim outDir As String

        Public Sub New(ByVal dirPath As String)
            outDir = dirPath
        End Sub

```

```

Public Function GetClientInterfaceName() As String _Implements
ICIPClientObject.GetClientInterfaceName
    GetClientInterfaceName = "IpfcSessionActionListener"
End Function

Public Sub OnBeforeModelErase() Implements pfcls.IpfcSessionActionListener.
OnBeforeModelErase
    Dim model As IpfcModel
    Dim cAC As New pfcls.CCpfcAsyncConnection
    Dim aC As pfcls.IpfcAsyncConnection
    Dim session As IpfcBaseSession
    Dim vrmlInstructions As IpfcVRMLModelExportInstructions
    Try

        aC = cAC.GetActiveConnection
        session = aC.Session
'=====
'Get the current solid
'=====
        model = session.CurrentModel
'=====
'Do nothing if model is not a part
'=====
        If model Is Nothing Then
            Return
        End If
        If (Not model.Type = EpfcModelType.EpfcMDL_PART) Then
            Return
        End If

        vrmlInstructions = (New CCpfcVRMLModelExportInstructions).Create
(outDir)
        model.Export(Nothing, vrmlInstructions)
    Catch ex As Exception
        MsgBox(ex.Message.ToString + Chr(13) +
            ex.StackTrace.ToString)
    End Try

End Sub

Public Sub OnAfterDirectoryChange(ByVal _Path As String) Implements pfcls.
IpfcSessionActionListener.OnAfterDirectoryChange

End Sub

Public Sub OnAfterModelDisplay() Implements pfcls.IpfcSessionActionListener.
OnAfterModelDisplay

End Sub

Public Sub OnAfterModelPurge(ByVal _Desrc As pfcls.IpfcModelDescriptor)
Implements pfcls.IpfcSessionActionListener.OnAfterModelPurge

End Sub

Public Sub OnAfterWindowChange(ByVal _NewWindow As Object) Implements pfcls.
IpfcSessionActionListener.OnAfterWindowChange

End Sub

```

```

    Public Sub OnBeforeModelCopy(ByVal _Container As pfcls.
IpfcDescriptorContainer2) Implements pfcls.IpfcSessionActionListener.
OnBeforeModelCopy

        End Sub

    Public Sub OnBeforeModelDelete() Implements pfcls.IpfcSessionActionListener.
OnBeforeModelDelete

        End Sub

    Public Sub OnBeforeModelPurge(ByVal _Container As pfcls.
IpfcDescriptorContainer) Implements pfcls.IpfcSessionActionListener.
OnBeforeModelPurge

        End Sub

    Public Sub OnBeforeModelRename(ByVal _Container As pfcls.
IpfcDescriptorContainer2) Implements pfcls.IpfcSessionActionListener.
OnBeforeModelRename

        End Sub

    Public Sub OnBeforeModelSave(ByVal _Container As pfcls.
IpfcDescriptorContainer) Implements pfcls.IpfcSessionActionListener.OnBeforeModelSave

        End Sub
End Class
End Class

```

## Importing Files

Method Introduced:

- **IpfcModel.Import()**

The method **IpfcModel.Import()** reads a file into Pro/ENGINEER. The format must be the same as it would be if these files were created by Pro/ENGINEER. The parameters are:

- FilePath--Absolute path of the file to be imported along with its extension.
- ImportData--The ImportInstructions object that controls the import operation.

## Import Instructions

Methods Introduced:

- **CCpfcrRelationImportInstructions.Create()**
- **CCpfcrIGESSectionImportInstructions.Create()**
- **CCpfcrProgramImportInstructions.Create()**
- **CCpfcrConfigImportInstructions.Create()**
- **CCpfcrDWGSetupImportInstructions.Create()**

- **CCpfcSpoolImportInstructions.Create()**
- **CCpfcConnectorParamsImportInstructions.Create()**
- **CCpfcASSEMBTreeCFGImportInstructions.Create()**
- **CCpfcWireListImportInstructions.Create()**
- **CCpfcCableParamsImportInstructions.Create()**
- **CCpfcSTEPImport2DInstructions.Create()**
- **CCpfcIGESImport2DInstructions.Create()**
- **CCpfcDXFImport2DInstructions.Create()**
- **CCpfcDWGImport2DInstructions.Create()**
- **CCpfcSETImport2DInstructions.Create()**

The methods described in this section create an instructions data object to import a file of a specified type into Pro/ENGINEER. The details are as shown in the table below:

–

<b>Interface</b>	<b>Used to Import</b>
IpfcRelationImportInstructions	A list of relations and parameters in a part or assembly.
IpfcIGESSectionImportInstructions	A section model in IGES format.
IpfcProgramImportInstructions	A program file for a part or assembly that can be edited to change the model.
IpfcConfigImportInstructions	Configuration instructions.
IpfcDWGSetupImportInstructions	A drawing s/u file.
IpfcSpoolImportInstructions	Spool instructions.
IpfcConnectorParamsImportInstructions	Connector parameter instructions.
IpfcASSEMBTreeCFGImportInstructions	Assembly tree CFG instructions.
IpfcWireListImportInstructions	Wirelist instructions.

IpfcCableParamsImportInstructions	Cable parameters from an assembly.
IpfcSTEPImport2DInstructions	A part or assembly in STEP format.
IpfcIGESImport2DInstructions	A part or assembly in IGES format.
IpfcDXFImport2DInstructions	A drawing in DXF format.
IpfcDWGImport2DInstructions	A drawing in DWG format.
IpfcSETImport2DInstructions	A class that defines a ruled surface.

**Note:**

- The method IpfcModel.Import() does not support importing of CADAM type of files.
- If a model or the file type STEP, IGES, DWX, or SET already exists, the imported model is appended to the current model. For more information on methods that return models of the types STEP, IGES, DWX, and SET, refer to Getting a Model Object.

## Importing 2D Models

Method Introduced:

- **IpfcBaseSession.Import2DModel()**

The method **IpfcBaseSession.Import2DModel()** imports a two dimensional model based on the following parameters:

- NewModelName--Specifies the name of the new model.
- Type--Specifies the type of the model. The type can be one of the following:
  - STEP
  - IGES
  - DXF
  - DWG
  - SET
- FilePath--Specifies the location of the file to be imported along with the file extension
- Instructions--Specifies the Import2DInstructions object that controls the import operation.

The interface `IpfcImport2DInstructions` contains the following attributes:

- Import2DViews--Defines whether to import 2D drawing views.
- ScaleToFit--If the current model has a different sheet size than that specified by the imported file, set the parameter to true to retain the current sheet size. Set the parameter to false to retain the sheet size of the imported file.
- FitToLeftCorner--If this parameter is set to true, the bottom left corner of the imported file is adjusted to the bottom left corner of the current model. If it is set to false, the size of imported file is retained.

**Note:**

The method `IpfcBaseSession.Import2DModel()` does not support importing of CADAM type of files.

## Importing 3D Geometry

Methods Introduced:

- **IpfcBaseSession.GetImportSourceType()**
- **IpfcBaseSession.ImportNewModel()**

For some input formats, the method **IpfcBaseSession.GetImportSourceType()** returns the type of model that can be imported using a designated file. The input parameters of this method are:

- FileToImport--Specifies the path of the file along with its name and extension
- NewModelImportType--Specifies the type of model to be imported.

The method **IpfcBaseSession.ImportNewModel()** is used to import an external file and creates a new model or set of models of type **IpfcModel:Model**. The input parameters of this method are:

- FileToImport--Specifies the path of the file along with its name and extension
- NewModelImportType--Specifies the type of model to be imported.
- ModelType--Specifies the type of the model. It can be a part, assembly or drawing.
- NewModelName--Specifies a name for the imported model. The import types are as follows:
  - IMPORT\_NEW\_IGES
  - IMPORT\_NEW\_SET
  - IMPORT\_NEW\_VDA
  - IMPORT\_NEW\_NEUTRAL
  - IMPORT\_NEW\_CADDS
  - IMPORT\_NEW\_STEP
  - IMPORT\_NEW\_STL
  - IMPORT\_NEW\_VRML
  - IMPORT\_NEW\_POLTXT
  - IMPORT\_NEW\_CATIA
  - IMPORT\_NEW\_CATIA\_SESSION
  - IMPORT\_NEW\_CATIA\_MODEL
  - IMPORT\_NEW\_DXF
  - IMPORT\_NEW\_ACIS
  - IMPORT\_NEW\_PARASOLID
  - IMPORT\_NEW\_ICEM
  - IMPORT\_NEW\_DESKTOP

## Plotting Files

Methods and Properties Introduced:

- **IpfcModel.Export()**
- **CCpfcPlotInstructions.Create()**
- **IpfcPlotInstructions.PlotterName**
- **IpfcPlotInstructions.OutputQuality**
- **IpfcPlotInstructions.UserScale**
- **IpfcPlotInstructions.PenSlew**
- **IpfcPlotInstructions.PenVelocityX**

- **IpfcPlotInstructions.PenVelocityY**
- **IpfcPlotInstructions.SegmentedOutput**
- **IpfcPlotInstructions.LabelPlot**
- **IpfcPlotInstructions.SeparatePlotFiles**
- **IpfcPlotInstructions.PaperSize**
- **IpfcPlotInstructions.PageRangeChoice**
- **IpfcPlotInstructions.PaperSizeX**
- **IpfcPlotInstructions.FirstPage**
- **IpfcPlotInstructions.LastPage**

## Instructions for objects used to plot drawings

The following is a list of instructions that pertain to the Plot Instruction objects.

- **PlotterName**--A printer name that is offered by the File > Print command.
- **OutputQuality**--A value of 0, 1, 2, or 3. Default is 1. Defines the amount of checking for overlapping lines in a plot or 2-D export file, such as IGES, before making a file. The values are interpreted as follows:
  - 0--Does not check for overlapping lines or collect lines of the same pen color.
  - 1--Does not check for overlapping lines, but collects lines of the same pen color for plotting.
  - 2--Partially checks edges with two vertices, and collects lines of the same pen color for plotting.
  - 3--Does a complete check of all edges against each other, regardless of the number of vertices, font, or color. Collects lines of the same pen color for plotting.
- **Use Scale**--Specifies a scale factor between 0.01 and 100 for scaling a model or drawing for plotting. Default is 0.01.
- **PenSlew**--Set to true if you want to adjust pen velocity. Default is false.
- **PenVelocity X**--When PenSlew is true, this value is a multiple of the default pen speed in the X dimension. Permitted range is 0.1 to 100. Ignored when PenSlew is false.
- **PenVelocity Y**--When PenSlew is true, this value is a multiple of the default pen speed in the y dimension. Permitted range is 0.1 to 100. Ignored when PenSlew is false.
- **SegmentedOutput**--Set to true to generate a segmented plot. Default is false.
- **LabelPlot**--If set to true, generates the plot with a label. Default is false; no label is created.
- **SeparatePlotFiles**--Defines the default in the Print to File dialog box.
  - true--Sets the default to Create Separate Files.
  - false--A single file is created by default.
- **PaperSize**--One of the PlotPaperSize enumeration objects. Default is PlotPaperSize.ASIZEPLOT.
- **PageRangeChoice**--One of the PlotPageRange enumeration objects. Default is PlotPageRange.PLOT\_RANGE\_ALL.
- **PaperSizeX**--When PaperSize is PlotPaperSize.VARIABLEPLOTSIZE, this specifies the size of the plotter paper in the X dimension. Otherwise, the value is null.
- **PaperSizeY**--When PaperSize is PlotPaperSize.VARIABLEPLOTSIZE, this specifies the size of the plotter paper in the Y dimension. Otherwise, the value is null.

## Plotting

To plot a file using the VB API, create a set of plot instructions containing the plotter name as a string. This name should be identical to the name found using the printer Toolbar icon and it should be capitalized.

### Note:

While plotting a drawing, the drawing must be displayed in a window to be successfully plotted.

The following table lists the default plotter settings assigned when a PlotInstructions object is created.

Plotter Setting	Default Value
Output quality	1
User scale	1.0
Pen slew	false
X-direction pen velocity	1.0
Y-direction pen velocity	1.0
Segmented output	false
Label plot	false
Separate plot files	false
Plot paper size	PlotPaperSize.ASIZEPLOT
Plot page range	PlotPageRange. PLOT_RANGE_ALL
Paper size (X)	null
Paper size (Y)	null
Page upper limit	null
Page lower limit	null

## Solid Operations

Methods Introduced:

- **IpfcSolid.CreateImportFeat()**

The method **IpfcSolid.CreateImportFeat()** creates a new import feature in the solid and takes the following input arguments:

- o IntfDataSource--The source of data from which to create the import feature.
- o CoordSys--The pointer to a reference coordinate system. If this is NULL, the function uses the default coordinate system.
- o FeatAttr--The attributes for creation of the new import feature. If this pointer is NULL, the function uses the default attributes.

**Example Code: Returning a Feature Object**

This method will return a feature object when provided with a solid coordinate system name and an import feature's file name. The method will find the coordinate system in the model, set the Import Feature Attributes, and create an import feature. Then the feature is returned.

```
Public Function createImportFeatureFromDataFile(ByVal solid As IpfcSolid,
    _ByVal csys As String, _ByVal fileName As String,
    _ByVal type As EpfcIntfType) _As IpfcFeature

    Dim dataSource As IpfcIntfDataSource
    Dim cSystems As IpfcModelItems
    Dim cSystem As IpfcCoordSystem = Nothing
    Dim importFeature As IpfcFeature
    Dim featAttr As IpfcImportFeatAttr
    Dim i As Integer

    Try

        Select Case type
            Case EpfcIntfType.EpfcINTF_NEUTRAL
                dataSource = (New CCpfcIntfNeutralFile).
                    Create(fileName)
            Case EpfcIntfType.EpfcINTF_CATIA
                dataSource = (New CCpfcIntfCATIA).Create(fileName)
            Case EpfcIntfType.EpfcINTF_IGES
                dataSource = (New CCpfcIntfIges).Create(fileName)
            Case EpfcIntfType.EpfcINTF_PDGS
                dataSource = (New CCpfcIntfPDGS).Create(fileName)
            Case EpfcIntfType.EpfcINTF_SET
                dataSource = (New CCpfcIntfSet).Create(fileName)
            Case EpfcIntfType.EpfcINTF_STEP
                dataSource = (New CCpfcIntfStep).Create(fileName)
            Case EpfcIntfType.EpfcINTF_VDA
                dataSource = (New CCpfcIntfVDA).Create(fileName)
            Case Else
                Throw New Exception("Unknown File Type")
        End Select

        cSystems = solid.ListItems
            (EpfcModelItemType.EpfcITEM_COORD_SYS)

        For i = 0 To cSystems.Count - 1
            If (cSystems.Item(i).GetName.ToString = csys) Then
                cSystem = cSystems.Item(i)
                Exit For
            End If
        Next

        If cSystem Is Nothing Then
```

```

        Throw New Exception("Coordinate System not found in current
                               Solid")
    End If
'=====
'Create the import ImportFeatAttr structure join surfaces, make solids
'from every closed quilt using the add operation
'=====
        featAttr = (New CCpfcImportFeatAttr).Create()
        featAttr.JoinSurfs = True
        featAttr.MakeSolid = True
        featAttr.Operation = EpfcOperationType.EpfcADD_OPERATION

        importFeature = solid.CreateImportFeat(dataSource, cSystem,
                                                featAttr)

        Return importFeature

    Catch ex As Exception
        MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
        Return Nothing
    End Try
End Function

```

## Window Operations

Methods Introduced:

- **IpfcWindow.ExportRasterImage()**

The method **IpfcWindow.ExportRasterImage()** outputs a standard Pro/ENGINEER raster output file.

### Example Code: Generating Raster Files

The following code is used to generate raster image files using a specified window and file type.

```

'Generating Raster Files
'=====
'Function      :   outputImageWindow
'Purpose      :   This function takes a Window and outputs a raster image
'                :   file depicting the window. This method takes as an
'                :   argument the type of the raster file, but the size and
'                :   image quality of the raster file are hardcoded.
'=====
Public Sub outputImageWindow(ByRef window As IpfcWindow,
                            _ByVal type As Integer, _ByVal imageName As String)

    Dim instructions As IpfcRasterImageExportInstructions
    Dim imageExtension As String
    Dim rasterHeight As Double = 7.5
    Dim rasterWidth As Double = 10.0
    Dim dotsPerInch As Integer
    Dim imageDepth As Integer

    Try
        dotsPerInch = EpfcDotsPerInch.EpfcRASTERDPI_100
        imageDepth = EpfcRasterDepth.EpfcRASTERDEPTH_24
    
```

```

        instructions = getRasterInstructions(type, rasterWidth,
                                           _rasterHeight, dotsPerInch, _imageDepth)

        imageExtension = getRasterExtension(type)

        window.ExportRasterImage(imageName + imageExtension, instructions)

    Catch ex As Exception
        MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
    End Try
End Sub

'=====  

'Function    :    outputImageScreen  

'Purpose    :    This function takes a ProE Session and outputs a raster  

'              :    image file depicting the window. This method takes as an  

'              :    argument the type of the raster file, but the size and  

'              :    image quality of the raster file are hardcoded.  

'=====  

Public Sub outputImageScreen(ByRef session As IpfcBaseSession,
                             _ByVal type As Integer,
                             _ByVal imageName As String)

    Dim instructions As IpfcRasterImageExportInstructions
    Dim imageExtension As String
    Dim rasterHeight As Double = 7.5
    Dim rasterWidth As Double = 10.0
    Dim dotsPerInch As Integer
    Dim imageDepth As Integer

    Try
        dotsPerInch = EpfcDotsPerInch.EpfcRASTERDPI_100
        imageDepth = EpfcRasterDepth.EpfcRASTERDEPTH_24

        instructions = getRasterInstructions(type, rasterWidth,
                                           _rasterHeight, dotsPerInch, _imageDepth)

        imageExtension = getRasterExtension(type)

        session.ExportCurrentRasterImage(imageName + imageExtension,
instructions)

    Catch ex As Exception
        MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
    End Try
End Sub

'=====  

'Function    :    getRasterInstructions  

'Purpose    :    A helper method which creates a  

'              :    RasterImageExportInstructions object based on the type.  

'=====  

Private Function getRasterInstructions(ByVal type As Integer,
                                       _ByVal rasterWidth As Double,
                                       _ByVal rasterHeight As Double,
                                       _ByVal dotsPerInch As Integer,
                                       _ByVal imageDepth As Integer) As
_IpfcRasterImageExportInstructions

    Dim instructions As IpfcRasterImageExportInstructions

```

```

Select Case type

    Case EpfcRasterType.EpfcRASTER_BMP
        Dim bmpInstrs As IpfcBitmapImageExportInstructions
        bmpInstrs = (New CCpfcBitmapImageExportInstructions).Create
(rasterWidth, rasterHeight)
        instructions = bmpInstrs

    Case EpfcRasterType.EpfcRASTER_TIFF
        Dim tiffInstrs As IpfcTIFFImageExportInstructions
        tiffInstrs = (New CCpfcTIFFImageExportInstructions).Create
(rasterWidth, rasterHeight)
        instructions = tiffInstrs

    Case EpfcRasterType.EpfcRASTER_JPEG
        Dim jpegInstrs As IpfcJPEGImageExportInstructions
        jpegInstrs = (New CCpfcJPEGImageExportInstructions).Create
(rasterWidth, rasterHeight)
        instructions = jpegInstrs

    Case EpfcRasterType.EpfcRASTER_EPS
        Dim epsInstrs As IpfcEPSImageExportInstructions
        epsInstrs = (New CCpfcEPSImageExportInstructions).Create
(rasterWidth, rasterHeight)
        instructions = epsInstrs

    Case Else
        Throw New Exception("Unsupported Raster Type")
End Select

instructions.DotsPerInch = dotsPerInch
instructions.ImageDepth = imageDepth

Return instructions
End Function
'=====
'Function      :   getRasterExtension
'Purpose      :   A helper method to create file extension based on the
'                Raster type.
'=====
Private Function getRasterExtension(ByVal type As Integer) As String

    Select Case type

        Case EpfcRasterType.EpfcRASTER_BMP
            Return ".bmp"

        Case EpfcRasterType.EpfcRASTER_TIFF
            Return ".tiff"

        Case EpfcRasterType.EpfcRASTER_JPEG
            Return ".jpg"

        Case EpfcRasterType.EpfcRASTER_EPS
            Return ".eps"

    Case Else
        Throw New Exception("Unsupported Raster Type")

```

End Select

End Function

---

# Simplified Representations

---

The VB API gives programmatic access to all the simplified representation functionality of Pro/ENGINEER. Create simplified representations either permanently or on the fly and save, retrieve, or modify them by adding or deleting items.

## Topic

[Overview](#)

[Retrieving Simplified Representations](#)

[Creating and Deleting Simplified Representations](#)

[Extracting Information About Simplified Representations](#)

[Modifying Simplified Representations](#)

[Simplified Representation Utilities](#)

## Overview

Using the VB API, you can create and manipulate assembly simplified representations just as you can using Pro/ENGINEER interactively.

### Note:

The VB API supports simplified representation of assemblies only, not parts.

Simplified representations are identified by the `IpfcSimRep` class. This class is a child of `IpfcModelItem`, so you can use the methods dealing with `IpfcModelItems` to collect, inspect, and modify simplified representations.

The information required to create and modify a simplified representation is stored in a class called `IpfcSimRepInstructions` which contains several data objects and fields, including:

- String--The name of the simplified representation
- `IpfcSimpRepAction`--The rule that controls the default treatment of items in the simplified representation.
- `IpfcSimpRepItem`--An array of assembly components and the actions applied to them in the simplified representation.

A `IpfcSimpRepItem` is identified by the assembly component path to that item. Each `IpfcSimpRepItem` has its own `IpfcSimpRepAction` assigned to it. `IpfcSimpRepAction` is a visible data object that includes a field of type `IpfcSimpRepActionType`.

`IpfcSimpActionType` is an enumerated type that specifies the possible treatment of items in a simplified representation. The possible values are as follows

Values	Action
EpfcSIMPREP_NONE	No action is specified.
EpfcSIMPREP_REVERSE	Reverse the default rule for this component (for example, include it if the default rule is exclude).
EpfcSIMPREP_INCLUDE	Include this component in the simplified representation.
EpfcSIMPREP_EXCLUDE	Exclude this component from the simplified representation.
EpfcSIMPREP_SUBSTITUTE	Substitute the component in the simplified representation.
EpfcSIMPREP_GEOM	Use only the geometrical representation of the component.
EpfcSIMPREP_GRAPHICS	Use only the graphics representation of the component.

## Retrieving Simplified Representations

Methods Introduced:

- **IpfcBaseSession.RetrieveAssemSimpRep()**
- **IpfcBaseSession.RetrieveGeomSimpRep()**
- **IpfcBaseSession.RetrieveGraphicsSimpRep()**
- **IpfcBaseSession.RetrieveSymbolicSimpRep()**
- **CCpfcRetrieveExistingSimpRepInstructions.Create()**

You can retrieve a named simplified representation from a model using the method **IpfcBaseSession.RetrieveAssemSimpRep()**, which is analogous to the Assembly mode option **Retrieve Rep** in the **SIMPLFD REP** menu. This method retrieves the object of an existing simplified representation from an assembly without fetching the generic representation into memory. The method takes two arguments, the name of the assembly and the simplified representation data.

To retrieve an existing simplified representation, pass an instance of **CCpfcRetrieveExistingSimpRepInstructions.Create()** and specify its name as the second argument to this method. Pro/ENGINEER retrieves that representation and any active submodels and returns the object to the simplified representation as a *IpfcAssembly.Assembly* object.

You can retrieve geometry, graphics, and symbolic simplified representations into session using the methods **IpfcBaseSession.RetrieveGeomSimpRep()**, **IpfcBaseSession.RetrieveGraphicsSimpRep()**, and **IpfcBaseSession.RetrieveSymbolicSimpRep()** respectively. Like **IpfcBaseSession.RetrieveAssemSimpRep()**, these methods retrieve the simplified representation without bringing the master representation into memory. Supply the name of the assembly whose simplified representation is to be retrieved as the input parameter for these methods. The methods output the assembly. They do not display the simplified representation.

## Creating and Deleting Simplified Representations

Methods Introduced:

- **CCpfcCreateNewSimpRepInstructions.Create()**
- **IpfcSolid.CreateSimpRep()**
- **IpfcSolid.DeleteSimpRep()**

To create a simplified representation, you must allocate and fill a `IpfcSimpRepInstructions` object by calling the method **CCpfcCreateNewSimpRepInstructions.Create()**. Specify the name of the new simplified representation as an input to this method. You should also set the default action type and add `SimpRepItems` to the object.

To generate the new simplified representation, call **IpfcSolid.CreateSimpRep()**. This method returns the `IpfcSimpRep` object for the new representation.

The method **IpfcSolid.DeleteSimpRep()** deletes a simplified representation from its model owner. The method requires only the `IpfcSimpRep` object as input.

## Extracting Information About Simplified Representations

Methods and Properties Introduced:

- **IpfcSimpRep.GetInstructions()**
- **IpfcSimpRepInstructions.DefaultAction**
- **IpfcCreateNewSimpRepInstructions.NewSimpName**
- **IpfcSimpRepInstructions.IsTemporary**
- **IpfcSimpRepInstructions.Items**

Given the object to a simplified representation, **IpfcSimpRep.GetInstructions()** fills out the `IpfcSimpRepInstructions` object.

The **IpfcSimpRepInstructions.DefaultAction**, **IpfcCreateNewSimpRepInstructions.NewSimpName**, and **IpfcSimpRepInstructions.IsTemporary** methods/properties return the associated

values contained in the `IpfcSimpRepInstructions` object.

The methodproperty **`IpfcSimpRepInstructions.Items`** returns all the items that make up the simplified representation.

## Example 1: Working with Simplified Representation

This code demonstrates the functionality used when working with existing simplified representations in Pro/ENGINEER. This function `matchSimpRepItem` returns an array of simplified representation matching a `ComponentPath` for a certain feature as well as the `SimpRepActionType` for that item's action in the representation. If none are found the method prints the <NOT FOUND> message and returns null.

```
Public Class pfcSimplifiedRepresentationExamples
    'Working with Simplified Representation

    '=====
    'Function      :   matchSimpRepItem
    'Purpose      :   This method will return an array of Simplified
    '              :   Representation matching a ComponentPath for a certain
    '              :   feature as well as the SimpRepActionType for that
    '              :   item's action in the Representation.
    '=====

    Public Function matchSimpRepItem(ByVal path As IpfcComponentPath, _
                                     ByVal type As EpfcSimpRepActionType) _
                                     As CpfcSimpReps

        Dim rootAssembly As IpfcAssembly
        Dim modelItems As IpfcModelItems
        Dim numSimpReps As Integer = 0
        Dim i, j As Integer
        Dim simRep As IpfcSimpRep
        Dim simRepInstrs As IpfcCreateNewSimpRepInstructions
        Dim simRepItems As CpfcSimpRepItems
        Dim numComponents As Integer
        Dim simRepItem As IpfcSimpRepItem
        Dim action As EpfcSimpRepActionType
        Dim found As Boolean = False
        Dim equalIntSeq As Boolean = False
        Dim itemPath As Cintseq
        Dim simpReps As CpfcSimpReps

        Try

            '=====
            'Get the root assembly and all the simplified representations
            '=====

            rootAssembly = path.Root
            modelItems =
                rootAssembly.ListItems(EpfcModelItemType.EpfcITEM_SIMPREP)
```

```

numSimpReps = modelItems.Count
If numSimpReps = 0 Then
    Throw New Exception("No Simplified Representations exist")
End If

simpReps = New CpfcsimpReps

'=====  

'Loop through all the simp reps  

'=====  

For i = 0 To numSimpReps - 1
    simRep = modelItems.Item(i)
    simRepInstrs = simRep.GetInstructions()
    simRepItems = simRepInstrs.Items
    numComponents = simRepItems.Count
'=====  

'Loop through all the items in each simp rep and check if  

any matches the inputs to the function.  

'=====  

For j = 0 To numComponents - 1
    simRepItem = simRepItems.Item(j)

    If TypeOf simRepItem.ItemPath Is  

        IpfcSimpRepCompItemPath Then
        itemPath = CType(simRepItem.ItemPath,  

            IpfcSimpRepCompItemPath).ItemPath
        If (compareSeq(itemPath, path.ComponentIds)) Then
            action = simRepItem.Action.GetType()
            If action = type Then
                simpReps.Insert(simpReps.Count, simRep)
            End If
        End If
    End If
Next
Next

If simpReps.Count = 0 Then
    Return Nothing
Else
    Return simpReps
End If

Catch ex As Exception
    MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
    Return Nothing
End Try
End Function

```

```

'=====  

'Function      :   compareSeq  

'Purpose      :   This method compares two Cintseq objects.  

'=====  


```

```
Public Function compareSeq(ByVal seq1 As Cintseq, ByVal seq2 As  
                          Cintseq) _As Boolean
```

```
    Dim len1, len2 As Integer  
    Dim i As Integer
```

```
    len1 = seq1.Count
```

```
    len2 = seq2.Count
```

```
    If Not len1 = len2 Then
```

```
        Return False
```

```
    Else
```

```
        For i = 0 To len1 - 1
```

```
            If Not seq1.Item(i) = seq2.Item(i) Then
```

```
                Return False
```

```
            End If
```

```
        Next
```

```
    End If
```

```
    Return True
```

```
End Function
```

```
End Class
```

## Modifying Simplified Representations

Methods and Properties Introduced:

- **IpfcSimpRep.GetInstructions()**
- **IpfcSimpRep.SetInstructions()**
- **IpfcSimpRepInstructions.DefaultAction**
- **IpfcCreateNewSimpRepInstructions.NewSimpName**
- **IpfcSimpRepInstructions.IsTemporary**

Using the VB API, you can modify the attributes of existing simplified representations. After you create or retrieve a simplified representation, you can make calls to the set methods listed in this section to designate new values for the fields in the `IpfcSimpRepInstructions` object.

To modify an existing simplified representation retrieve it and then get the `IpfcSimpRepInstructions` object by calling **IpfcSimpRep.GetInstructions()**. If you created the representation programmatically within the same application, the `IpfcSimpRepInstructions` object is already available. Once you have modified the data object, reassign it to the corresponding simplified representation by calling the method **IpfcSimpRep.SetInstructions()**.

## Adding Items to and Deleting Items from a Simplified Representation

Methods and Properties Introduced:

- **IpfcSimpRepInstructions.Items**
- **CCpfcSimpRepItem.Create()**
- **IpfcSimpRep.SetInstructions()**
- **CCpfcSimpRepReverse.Create()**
- **CCpfcSimpRepInclude.Create()**
- **CCpfcSimpRepExclude.Create()**
- **CCpfcSimpRepSubstitute.Create()**
- **CCpfcSimpRepGeom.Create()**
- **CCpfcSimpRepGraphics.Create()**

You can add and delete items from the list of components in a simplified representation using the VB API. 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 `EpfcSimpRepActionType` to `EpfcSIMPREP_EXCLUDE`.

## How to Add Items

1. Get the `IpfcSimpRepInstructions` object, as described in the previous section.
2. Specify the action to be applied to the item with a call to one of following methods.
3. Initialize a `IpfcSimpRepItem` object for the item by calling the method **CCpfcSimpRepItem.Create()** .
4. Add the item to the `IpfcSimpRepItem` sequence. Put the new `IpfcSimpRepInstructions` using **IpfcSimpRepInstructions.Items**.
5. Reassign the `IpfcSimpRepInstructions` object to the corresponding `IpfcSimpRep` object by calling **IpfcSimpRep.SetInstructions()** .

## How to Remove Items

Follow the procedure above, except remove the unwanted `IpfcSimpRepItem` from the sequence.

# Simplified Representation Utilities

Methods Introduced:

- **IpfcModelItemOwner.ListItems()**
- **IpfcModelItemOwner.GetItemById()**
- **IpfcSolid.GetSimpRep()**
- **IpfcSolid.SelectSimpRep()**
- **IpfcSolid.ActivateSimpRep()**
- **IpfcSolid.GetActiveSimpRep()**

This section describes the utility methods that relate to simplified representations.

The method **IpfcModelItemOwner.ListItems()** can list all of the simplified representations in a Solid.

The method **IpfcModelItemOwner.GetItemById()** initializes a `IpfcSimpRep.SimpRep` object. It takes an integer id.

**Note:**

The VB API supports simplified representation of Assemblies only, not Parts.

The method **IpfcSolid.GetSimpRep()** initializes a `IpfcSimpRep` object. The method takes the following arguments:

- **SimpRepname**-- The name of the simplified representation in the solid. If you specify this argument, the method ignores the `rep_id`.

The method **IpfcSolid.SelectSimpRep()** creates a Pro/ENGINEER menu to enable interactive selection. The method takes the owning solid as input, and outputs the object to the selected simplified representation. If you choose the **Quit** menu button, the method throws an exception `XToolkitUserAbort`.

The methods **IpfcSolid.GetActiveSimpRep()** and **IpfcSolid.ActivateSimpRep()** enable you to find and get the currently active simplified representation, respectively. Given an assembly object, **IpfcSolid.Solid.GetActiveSimpRep()** returns the object to the currently active simplified representation. If the current representation is the master representation, the return is **null**.

The method **IpfcSolid.ActivateSimpRep()** activates the requested simplified representation.

To set a simplified representation to be the currently displayed model, you must also call **IpfcModelDisplay()**.

---

# Task Based Application Libraries

---

Applications created using different Pro/ENGINEER API products are interoperable. These products use Pro/ENGINEER as the medium of interaction, eliminating the task of writing native-platform specific interactions between different programming languages.

Application interoperability allows the VB API applications to call into Pro/TOOLKIT from areas not covered in the native interface. It allows you to put a VBA or VB.NET front end on legacy Pro/TOOLKIT applications, and also allows you to use J-Link applications and listeners in conjunction with a Pro/Web.Link or asynchronous J-Link application.

## Topic

[Managing Application Arguments](#)

[Launching a Pro/TOOLKIT DLL](#)

[Launching Tasks from J-Link Task Libraries](#)

## Managing Application Arguments

The VB API passes application data to and from tasks in other applications as members of a sequence of `IpfcArgument` objects. Application arguments consist of a label and a value. The value may be of any one of the following types:

- Integer
- Double
- Boolean
- ASCII string (a non-encoded string, provided for compatibility with arguments provided from C applications)
- String (a fully encoded string)
- `IpfcSelection` (a selection of an item in a Pro/ENGINEER session)
- `IpfcTransform3D` (a coordinate system transformation matrix)

Methods and Properties Introduced:

- **`CMpfcArgument.CreateIntArgValue()`**

- **CMpfcArgument.CreateDoubleArgValue()**
- **CMpfcArgument.CreateBoolArgValue()**
- **CMpfcArgument.CreateASCIIStringArgValue()**
- **CMpfcArgument.CreateStringArgValue()**
- **CMpfcArgument.CreateSelectionArgValue()**
- **CMpfcArgument.CreateTransformArgValue()**
- **IpfcArgValue.dscr**
- **IpfcArgValue.IntValue**
- **IpfcArgValue.DoubleValue**
- **IpfcArgValue.BoolValue**
- **IpfcArgValue.ASCIIStringValue**
- **IpfcArgValue.StringValue**
- **IpfcArgValue.SelectionValue**
- **IpfcArgValue.TransformValue**

The class `pfc.ArgValue` contains one of the seven types of values. The VB API provides different methods to create each of the seven types of argument values.

The **IpfcArgValue.dscr** returns the type of value contained in the argument value object.

Use the methods listed above to access and modify the argument values.

## Modifying Arguments

Methods and Properties Introduced:

- **CCpfcArgument.Create()**
- **IpfcArgument.Label**
- **IpfcArgument.Value**

The method **CCpfcArgument.Create()** creates a new argument. Provide a name and value as the input arguments of this method.

The property **IpfcArgument.Label** returns the label of the argument.

The property **IpfcArgument.Value** returns the value of the argument.

## Launching a Pro/TOOLKIT DLL

The methods described in this section enable the VB API user to register and launch a Pro/TOOLKIT DLL from an application. The ability to launch and control a Pro/TOOLKIT application enables the following:

- Reuse of existing Pro/TOOLKIT code with the VB API applications.
- ATB operations.

Methods and Properties Introduced:

- **IpfcBaseSession.LoadProToolkitDll()**
- **IpfcBaseSession.GetProToolkitDll()**
- **IpfcDll.ExecuteFunction()**
- **IpfcDll.Id**
- **IpfcDll.IsActive()**
- **IpfcDll.Unload()**

Use the method **IpfcBaseSession.LoadProToolkitDll()** 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:

- **ApplicationName**--The name of the application to initialize.
- **DllPath**--The DLL file to load, including the path.
- **TextPath**--The path to the application's message and user interface text files.
- **UserDisplay**--Set this parameter to True, to see the application registered in the Pro/ENGINEER user interface and to see error messages if the application fails.

The application's **user\_initialize()** function is called when the application is started. The method returns a handle to the loaded DLL.

Use the method **IpfcBaseSession.GetProToolkitDll()** to obtain a Pro/TOOLKIT DLL handle. Specify the *Application\_Id*, that is, the DLL's identifier string as the input parameter of this method. The method returns the DLL object or null if the DLL was not in session. The *Application\_Id* can be determined as follows:

- Use the function **ProToolkitDllIdGet()** within the DLL application to get a string representation of the DLL application. Pass NULL to the first argument of **ProToolkitDllIdGet()** to get the string identifier for the calling application.
- Use the **Get** method for the **Id** attribute in the DLL interface. The method **IpfcDll.Id** returns the DLL identifier string.

Use the method **IpfcDll.ExecuteFunction()** to call a properly designated function in the Pro/TOOLKIT DLL library. The input parameters of this method are:

- **FunctionName**--Name of the function in the Pro/TOOLKIT DLL application.
- **InputArguments**--Input arguments to be passed to the library function.

The method returns an object of **IpfcFunctionReturn**. This interface contains data returned by a Pro/TOOLKIT function call. The object contains the return value, as integer, of the executed function and the output arguments passed back from the function call.

The method **IpfcDll.IsActive()** determines whether a Pro/TOOLKIT DLL previously loaded by the method **IpfcBaseSession.LoadProToolkitDll()** is still active.

The method **IpfcDll.Unload()** is used to shutdown a Pro/TOOLKIT DLL previously loaded by the method **IpfcBaseSession.LoadProToolkitDll()** and the application's **user\_terminate()** function is called.

# Launching Tasks from J-Link Task Libraries

The methods described in this section allow you to launch tasks from a predefined J-Link task library.

Methods Introduced:

- **IpfcBaseSession.StartJLinkApplication()**
- **IpfcJLinkApplication.ExecuteTask()**
- **IpfcJLinkApplication.IsActive()**
- **IpfcJLinkApplication.Stop()**

Use the method **IpfcBaseSession.StartJLinkApplication()** to start a J-Link application. The input parameters of this method are similar to the fields of a registry file and are as follows:

- **ApplicationName**--Assigns a unique name to this J-Link application.
- **ClassName**--Specifies the name of the Java class that contains the J-Link application's start and stop method. This should be a fully qualified Java package and class name.
- **StartMethod**--Specifies the start method of the J-Link application.
- **StopMethod**--Specifies the stop method of the J-Link application.
- **AdditionalClassPath**--Specifies the locations of packages and classes that must be loaded when starting this J-Link application. If this parameter is specified as null, the default classpath locations are used.
- **TextPath**--Specifies the application text path for menus and messages. If this parameter is specified as null, the default text locations are used.
- **UserDisplay**--Specifies whether to display the application in the Auxiliary Applications dialog box in Pro/ENGINEER.

Upon starting the application, the static **start()** method is invoked. The method returns a **IpfcJLinkApplication** referring to the J-Link application.

The method **IpfcJLinkApplication.ExecuteTask()** calls a registered task method in a J-Link application. The input parameters of this method are:

- Name of the task to be executed.
- A sequence of name value pair arguments contained by the interface

IpfcArguments.

The method outputs an array of output arguments.

The method **IpfcJLinkApplication.IsActive()** returns a *True* value if the application specified by the `IpfcJLinkApplication` object is active.

The method **IpfcJLinkApplication.Stop()** stops the application specified by the `IpfcJLinkApplication` object. This method activates the application's static **Stop()** method.

---

# Graphics

---

This section covers the VB API Graphics including displaying lists, displaying text and using the mouse.

## Topic

[Overview](#)

[Getting Mouse Input](#)

[Displaying Graphics](#)

[Display Lists and Graphics](#)

## Overview

The methods described in this section allow you to draw temporary graphics in a display window. Methods that are identified as 2D are used to draw entities (arcs, polygons, and text) in screen coordinates. Other entities may be drawn using the current model's coordinate system or the screen coordinate system's lines, circles, and polylines. Methods are also included for manipulating text properties and accessing mouse inputs.

## Getting Mouse Input

The following methods are used to read the mouse position in screen coordinates with the mouse button depressed. Each method outputs the position and an enumerated type description of which mouse button was pressed when the mouse was at that position. These values are contained in the interface `IpfcMouseStatus`.

The enumerated values are defined in **EpcfMouseButton** and are as follows:

- `EpcfMOUSE_BTN_LEFT`
- `EpcfMOUSE_BTN_RIGHT`
- `EpcfMOUSE_BTN_MIDDLE`
- `EpcfMOUSE_BTN_LEFT_DOUBLECLICK`

Methods Introduced:

- **`IpfcSession.UIGetNextMousePick()`**
- **`IpfcSession.UIGetCurrentMouseStatus()`**

The method **`IpfcSession.UIGetNextMousePick()`** returns the mouse position when you press a mouse button. The input argument is the mouse button that you expect the user to select.

The method **`IpfcSession.UIGetCurrentMouseStatus()`** returns a value whenever the mouse is moved or a button is pressed. With this method a button does not have to be pressed for a value to be returned. You can use an input argument to flag whether or not the returned positions are snapped to the window grid.

## Drawing a Mouse Box

This method allows you to draw a mouse box.

Method Introduced:

- **IpfcSession.UIPickMouseBox()**

The method **IpfcSession.UIPickMouseBox()** draws a dynamic rectangle from a specified point in screen coordinates to the current mouse position until the user presses the left mouse button. The return value for this method is of the type `IpfcOutline3D`.

You can supply the first corner location programmatically or you can allow the user to select both corners of the box.

## Displaying Graphics

All the methods in this section draw graphics in the Pro/ENGINEER current window and use the color and linestyle set by calls to **IpfcBaseSession.SetStdColorFromRGB()** and **IpfcBaseSession.SetLineStyle()**. The methods draw the graphics in the Pro/ENGINEER graphics color. The default graphics color is white.

The methods in this section are called using the interface `IpfcDisplay`. The `Display` interface is extended by the `IpfcBaseSession` interface. This architecture allows you to call all these methods on any `IpfcSession` object.

Methods Introduced:

- **IpfcDisplay.SetPenPosition()**
- **IpfcDisplay.DrawLine()**
- **IpfcDisplay.DrawPolyline()**
- **IpfcDisplay.DrawCircle()**
- **IpfcDisplay.DrawArc2D()**
- **IpfcDisplay.DrawPolygon2D()**

The method **IpfcDisplay.SetPenPosition()** sets the point at which you want to start drawing a line. The method **IpfcDisplay.DrawLine()** draws a line to the given point from the position given in the last call to either of the two methods. Call **IpfcDisplay.SetPenPosition()** for the start of the polyline, and **IpfcDisplay.DrawLine()** for each vertex. If you use these methods in two-dimensional modes, use screen coordinates instead of solid coordinates.

The function **IpfcDisplay.DrawCircle()** uses solid coordinates for the center of the circle and the radius value. The circle will be placed to the **XY** plane of the model.

The method **IpfcDisplay.DrawPolyline()** also draws polylines, using an array to define the polyline.

In two-dimensional models the Display Graphics methods draw graphics at the specified screen coordinates.

The method **IpfcDisplay.DrawPolygon2D()** draws a polygon in screen coordinates. The method **IpfcDisplay.DrawArc2D()** draws an arc in screen coordinates.

## Controlling Graphics Display

Properties Introduced:

- **IpfcDisplay.CurrentGraphicsColor**
- **IpfcDisplay.CurrentGraphicsMode**

The property **IpfcDisplay.CurrentGraphicsColor** returns the Pro/ENGINEER standard color used to display graphics. The Pro/ENGINEER default is `EpfcCOLOR_DRAWING` (white).

The property **IpfcDisplay.CurrentGraphicsMode** returns the mode used to draw graphics:

- `EpfcDRAW_GRAPHICS_NORMAL`--Pro/ENGINEER draws graphics in the required color in each invocation.
- `EpfcDRAW_GRAPHICS_COMPLEMENT`--Pro/ENGINEER draws graphics normally, but will erase graphics drawn a second time in the same location. This allows you to create rubber band lines.

#### **Example Code: Creating Graphics On Screen**

This example demonstrates the use of mouse-tracking methods to draw graphics on the screen. The static method **DrawRubberbandLine** prompts the user to pick a screen point. The example uses the 'complement mode' to cause the line to display and erase as the user moves the mouse around the window.

#### **Note:**

This example uses the method `transformPosition` to convert the coordinates into the 3D coordinate system of a solid model, if one is displayed.

```
Imports pfcls

Public Class pfcGraphicsExamples

    Public Sub drawRubberbandLine(ByRef session As pfcls.IpfcSession)

        Dim mouse As IpfcMouseStatus
        Dim firstPosition As CpfcPoint3D
        Dim secondPosition As CpfcPoint3D
        Dim currentMode As EpfcGraphicsMode

        Try
            session.UIDisplayMessage("pfcGraphicsExamples.txt", _
                "Pick first location for rubberband line", Nothing)
'=====
'Expect the user to pick with left button
'=====
            mouse = session.UIGetNextMousePick(EpfcMouseButton.EpfcMOUSE_BTN_LEFT)

            session.UIDisplayMessage("pfcGraphicsExamples.txt", _
                "Click left mouse button to exit", Nothing)

'=====
'Transform screen point to model location, if necessary
'=====
            firstPosition = transformPosition(session, mouse.Position)
'=====
'Set graphics mode to complement, so that graphics erase after use
'=====
            currentMode = session.CurrentGraphicsMode
            session.CurrentGraphicsMode = EpfcGraphicsMode.
EpfcDRAW_GRAPHICS_COMPLEMENT
'=====
'Get mouse position and loop till left mouse button is not pressed
'=====
            mouse = session.UIGetCurrentMouseStatus(False)
```

```

        While Not mouse.SelectedButton = EpfcMouseButton.EpfcMOUSE_BTN_LEFT
            session.SetPenPosition(firstPosition)
            secondPosition = transformPosition(session, mouse.Position)
' =====
'Draw a rubberband line
' =====
            session.DrawLine(secondPosition)
            mouse = session.UIGetCurrentMouseStatus(False)
' =====
'Erase the previous line
' =====
            session.SetPenPosition(firstPosition)
            session.DrawLine(secondPosition)

        End While

        session.CurrentGraphicsMode = currentMode

    Catch ex As Exception
        MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)

    End Try
End Sub
' =====
'Function      :   transformPosition
'Purpose      :   This function transforms the 2D screen coordinates into
'                :   3D model coordinates, if necessary.
' =====
Private Function transformPosition(ByRef session As pfcls.IpfcSession, _
                                ByVal inPosition As CpfcPoint3D) _
    As IpfcPoint3D

    Dim model As IpfcModel
    Dim currView As IpfcView
    Dim invOrient As IpfcTransform3D
    Dim outPosition As CpfcPoint3D

    model = session.CurrentModel
' =====
'Skip transform if model does not exist or is not a 3D model
' =====
    If model Is Nothing Then
        Return inPosition
    End If

    If (Not model.Type = EpfcModelType.EpfcMDL_PART) And _
        (Not model.Type = EpfcModelType.EpfcMDL_ASSEMBLY) And _
        (Not model.Type = EpfcModelType.EpfcMDL_MFG) Then
        Return inPosition
    End If
' =====
'Get current view's orientation and invert it
' =====
        currView = model.GetCurrentView()
        invOrient = currView.Transform
        invOrient.Invert()
' =====
'Get the model point
' =====

```

```
outPosition = invOrient.TransformPoint(inPosition)
Return outPosition
```

```
End Function
End Class
```

**Display example text**

```
#
#
Pick first location for rubberband line
Pick first location for rubberband line
#
#
Click left mouse button to exit
Click left mouse button to exit
#
#
```

## Displaying Text in the Graphics Window

Method Introduced:

- **IpfcDisplay.DrawText2D()**

The method **IpfcDisplay.DrawText2D()** places text at a position specified in screen coordinates. 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.

Text items drawn are not known to Pro/ENGINEER and therefore are not redrawn when you select View, Repaint. To notify the Pro/ENGINEER of these objects, create them inside the **OnDisplay()** method of the Display Listener.

## Controlling Text Attributes

Properties Introduced:

- **IpfcDisplay.TextHeight**
- **IpfcDisplay.WidthFactor**
- **IpfcDisplay.RotationAngle**
- **IpfcDisplay.SlantAngle**

These properties control the attributes of text added by calls to **IpfcDisplay.DrawText2D()**.

You can access the following information:

- Text height (in screen coordinates)
- Width ratio of each character, including the gap, as a proportion of the height
- Rotation angle of the whole text, in counterclockwise degrees
- Slant angle of the text, in clockwise degrees

## Controlling Text Fonts

Methods and Properties Introduced:

- **IpfcDisplay.DefaultFont**
- **IpfcDisplay.CurrentFont**
- **IpfcDisplay.GetFontById()**
- **IpfcDisplay.GetFontByName()**

The property **IpfcDisplay.DefaultFont** returns the default Pro/ENGINEER text font. The text fonts are identified in Pro/ENGINEER by names and by integer identifiers. To find a specific font, use the methods **IpfcDisplay.GetFontById()** or **IpfcDisplay.GetFontByName()**.

## Display Lists and Graphics

When generating a display of a solid in a window, Pro/ENGINEER maintains two display lists. A display list contains a set of vectors that are used to represent the shape of the solid in the view. A 3D display list contains a set of three-dimensional vectors that represent an approximation to the geometry of the edges of the solid. This list gets rebuilt every time the solid is regenerated.

A 2D display list contains the two-dimensional projections of the edges of the solid 3D display list onto the current window. It is rebuilt from the 3D display list when the orientation of the solid changes. The methods in this section enable you to add your own vectors to the display lists, so that the graphics will be redisplayed automatically by Pro/ENGINEER until the display lists are rebuilt.

When you add graphics items to the 2D display list, they will be regenerated after each repaint (when zooming and panning) and will be included in plots created by Pro/ENGINEER. When you add graphics to the 3D 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.

Methods Introduced:

- **IpfcDisplayListener.OnDisplay()**
- **IpfcDisplay.CreateDisplayList2D()**
- **IpfcDisplay.CreateDisplayList3D()**
- **IpfcDisplayList2D.Display()**
- **IpfcDisplayList3D.Display()**
- **IpfcDisplayList2D.Delete()**
- **IpfcDisplayList3D.Delete()**

A display listener is a class that acts similarly to an action listener. You must implement the method inherited from the `IpfcDisplay.DisplayListener` interface. The implementation should provide calls to methods on the provided `IpfcDisplay.Display` object to produce 2D or 3D graphics.

In order to create a display list in Pro/ENGINEER, you call **IpfcDisplay.CreateDisplayList2D()** or **IpfcDisplay.CreateDisplayList3D()** to tell Pro/ENGINEER to use your listener to create the display list vectors.

**IpfcDisplayList2D.Display()** or **IpfcDisplayList3D.Display()** will display or redisplay the elements in your display list. The application should delete the display list data when it is no longer needed.

The methods **IpfcDisplayList2D.Delete()** and the method **IpfcDisplayList3D.Delete()** will remove both the specified display list from a session.

**Note:**

The method **IpfcWindow.Refresh()** does not cause either of the display lists to be regenerated, but simply repaints the window using the 2-D display list.

## Exceptions

Possible exceptions that might be thrown by displaying graphics methods are shown in the following table:

Exception	Reason
XToolkitNotExist	The display list is empty.
XToolkitNotFound	The function could not find the display list or the font specified in a previous call to <b>IpfcDisplay.CurrentFont</b> was not found.
XToolkitCantOpen	The use of display lists is disabled.
XToolkitAbort	The display was aborted.
XToolkitNotValid	The specified display list is invalid.
XToolkitInvalidItem	There is an invalid item in the display list.
XToolkitGeneralError	The specified display list is already in the process of being displayed.

## Example Code

This example demonstrates the use of **pfDisplay** methods with 3D display lists. The static method **AddCircleDisplay()** creates a new 3D display list whose graphics are generated by the code in the **OnDisplay()** method of the **DisplayCircles** class. This display list places circles at all of the vertices of a part model on the screen.

```
Dim list3D As IpfcDisplayList3D

Public Sub addCircleDisplay(ByRef session As pfcls.IpfcSession)

Dim drawCircles As New DisplayCircles

Try
'-----
'Id is an arbitrary number but should be unique to the
'application
```

```

' =====
      list3D = session.CreateDisplayList3D(1, drawCircles)

      session.CurrentWindow.Repaint()

      Catch ex As Exception
          MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)
      End Try
  End Sub
' =====
'Function      :   deleteCircleDisplay
'Purpose       :   The method deletes the display list.
' =====
      Public Sub deleteCircleDisplay(ByRef session As pfcls.IpfcSession)
          If Not list3D Is Nothing Then
              list3D.Delete()
              session.CurrentWindow.Repaint()
          End If
      End Sub
' =====
'Class         :   DisplayCircles
'Purpose       :   Display list listener class - determines how the
'                display list shows the graphics.
' =====
      Private Class DisplayCircles
          Implements IpfcDisplayListener
          Implements ICIPClientObject
          Implements IpfcActionListener

          Public Function GetClientInterfaceName() As String Implements pfcls.
ICIPClientObject.GetClientInterfaceName
              GetClientInterfaceName = "IpfcDisplayListener"
          End Function

          Public Sub OnDisplay(ByVal _Display As pfcls.IpfcDisplay) Implements pfcls.
IpfcDisplayListener.OnDisplay
              Dim currColour As EpfcStdColor
              Dim session As IpfcSession
              Dim model As IpfcModel
              Dim edges As IpfcModelItems
              Dim i As Integer
              Dim edge As IpfcModelItem
              Dim vertex1, vertex2 As IpfcPoint3D
              Dim radius As Double = 0.5

              currColour = _Display.CurrentGraphicsColor
              _Display.CurrentGraphicsColor = EpfcStdColor.EpfcCOLOR_ERROR
' =====
'Get the current model and check that it is a part
' =====
              session = CType(_Display, IpfcSession)
              model = session.CurrentModel

              If model Is Nothing OrElse (Not model.Type = EpfcModelType.EpfcMDL_PART)
Then
                  Return
              End If
' =====
'Circles on all vertices

```

```
' =====  
edges = model.ListItems(EpfcModelItemType.EpfcITEM_EDGE)  
For i = 0 To edges.Count - 1  
    edge = edges.Item(i)  
    vertex1 = edge.Eval3DData(0.0).Point  
    vertex2 = edge.Eval3DData(1.0).Point  
  
    _Display.DrawCircle(vertex1, radius)  
    _Display.DrawCircle(vertex2, radius)  
  
Next  
  
_Display.CurrentGraphicsColor = currColour  
  
End Sub  
  
End Class
```

---

# External Data

---

This chapter explains using External Data in the VB API.

## Topic

[External Data](#)

[Exceptions](#)

## External Data

This chapter describes how to store and retrieve external data. External data enables a The VB API application to store its own data in a Pro/ENGINEER database in such a way that it is invisible to the Pro/ENGINEER user. This method is different from other means of storage accessible through the Pro/ENGINEER user interface.

## Introduction to External Data

External data provides a way for the Pro/ENGINEER application to store its own private information about a Pro/ENGINEER model within the model file. The data is built and interrogated by the 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; the application has complete control over form and content.

The external data for a specific Pro/ENGINEER model is broken down into classes and slots. A class is a named ``bin" for your data, and identifies it as yours so no other Pro/ENGINEER API application (or other classes in your own application) will use it by mistake. An application usually needs only one class. The class name should be unique for each application and describe the role of the data in your application.

Each class contains a set of data slots. Each slot is identified by an identifier and optionally, a name. A slot contains a single data item of one of the following types:

The VB API Type	Data
EpfcEXTDATA_INTEGER	integer
EpfcEXTDATA_DOUBLE	double
EpfcEXTDATA_STRING	string

The The VB API interfaces used to access external data in Pro/ENGINEER are:

The VB API Type	Data Type
IpfcExternalDataAccess	This is the top level object and is created when attempting to access external data.

<code>IpfcExternalDataClass</code>	This is a class of external data and is identified by a unique name.
<code>IpfcExternalDataSlot</code>	This is a container for one item of data. Each slot is stored in a class.
<code>IpfcExternalData</code>	This is a compact data structure that contains either an integer, double or string value.

## Compatibility with Pro/TOOLKIT

The VB API and Pro/TOOLKIT share external data in the same manner. The VB API external data is accessible by Pro/TOOLKIT and the reverse is also true. However, an error will result if The VB API attempts to access external data previously stored by Pro/TOOLKIT as a stream.

## Accessing External Data

Methods Introduced:

- **`IpfcModel.AccessExternalData()`**
- **`IpfcModel.TerminateExternalData()`**
- **`IpfcExternalDataAccess.IsValid()`**

The method **`IpfcModel.AccessExternalData()`** prepares Pro/ENGINEER to read external data from the model file. It returns the `IpfcExternalDataAccess` object that is used to read and write data. This method should be called only once for any given model in session.

The method **`IpfcModel.TerminateExternalData()`** stops Pro/ENGINEER from accessing external data in a model. When you use this method all external data in the model will be removed. Permanent removal will occur when the model is saved.

### Note:

If you need to preserve the external data created in session, you must save the model before calling this function. Otherwise, your data will be lost.

The method **`IpfcExternalDataAccess.IsValid()`** determines if the `IpfcExternalDataAccess` object can be used to read and write data.

## Storing External Data

Methods and Properties Introduced:

- **`IpfcExternalDataAccess.CreateClass()`**
- **`IpfcExternalDataClass.CreateSlot()`**
- **`IpfcExternalDataSlot.Value`**

The first step in storing external data in a new class and slot is to set up a class using the method **`IpfcExternalDataAccess.CreateClass()`**, which provides the class name. The method outputs `pfExternalDataClass`, used by the application to reference the class.

The next step is to use **IpfcExternalDataClass.CreateSlot()** to create an empty data slot and input a slot name. The method outputs a `pfcExternalDataSlot` object to identify the new slot.

**Note:**

Slot names cannot begin with a number.

The property **IpfcExternalDataSlot.Value** specifies the data type of a slot and writes an item of that type to the slot. The input is a `pfcExternalData` object that you can create by calling any one of the methods in the next section.

**Example code:**

This function demonstrates the usage of external data. It provides utility methods to convert a VB hashtable to a model's external data.

Imports pfcls

```
Public Class pfcExternalDataExamples
    Public Sub storeExternalData(ByRef session As IpfcBaseSession, _
                                ByVal table As Hashtable, _
                                ByVal className As String)

        Dim model As IpfcModel
        Dim dataAccess As IpfcExternalDataAccess
        Dim dataClass As IpfcExternalDataClass
        Dim row As DictionaryEntry
        Dim value As Object
        Dim data As IpfcExternalData
        Dim slot As IpfcExternalDataSlot

        Try
            '=====
            'Get the current solid
            '=====
            model = session.CurrentModel
            If model Is Nothing Then
                Throw New Exception("Model not present")
            End If
            '=====
            'Get or create the external class
            '=====
            dataAccess = model.AccessExternalData()
            dataClass = getClassByName(dataAccess, className)
            If dataClass Is Nothing Then
                dataClass = dataAccess.CreateClass(className)
            End If
            '=====
            'Loop on all the keys in the hash table
            '=====
            For Each row In table
                '=====
                'Class name must be string
                '=====
                If Not row.Key.GetType.ToString = "System.String" Then
                    Continue For
                End If
                value = row.Value
            End For
        End Try
    End Sub
End Class
```

```

' =====
' Create proper data type
' =====
    If value.GetType.ToString = "System.Int16" Or _
       value.GetType.ToString = "System.Int32" Or _
       value.GetType.ToString = "System.Byte" Then

        data = (New CMpfcExternal). _
                CreateIntExternalData(CType(value, System.Int32))

    ElseIf value.GetType Is System.Type.GetType("System.Double") Then

        data = (New CMpfcExternal). _
                CreateDoubleExternalData(CType(value, System.Double))

    Else
        data = (New CMpfcExternal). _
                CreateStringExternalData(value.ToString)

    End If

' =====
' Get or create the slot and assign the value
' =====
    slot = getSlotByName(dataClass, row.Key.ToString)
    If slot Is Nothing Then
        slot = dataClass.CreateSlot(row.Key.ToString)
    End If
    slot.Value = data
Next

    'model.Save()

    Catch ex As Exception
        MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.
ToString)
    End Try
End Sub
End Class

```

## Initializing Data Objects

Methods Introduced:

- **CMpfcExternal.CreateIntExternalData()**
- **CMpfcExternal.CreateDoubleExternalData()**
- **CMpfcExternal.CreateStringExternalData()**

These methods initialize a `pfcExternalData` object with the appropriate data inputs.

## Retrieving External Data

Methods and Properties Introduced:

- **IpfcExternalDataAccess.LoadAll()**

- **IpfcExternalDataAccess.ListClasses()**
- **IpfcExternalDataClass.ListSlots()**
- **IpfcExternalData.discr**
- **IpfcExternalData.IntegerValue**
- **IpfcExternalData.DoubleValue**
- **IpfcExternalData.StringValue**

For improved performance, external data is not loaded automatically into memory with the model. When the model is in session, call the method **IpfcExternalDataAccess.LoadAll()** to retrieve all the external data for the specified model from the Pro/ENGINEER model file and put it in the workspace. The method needs to be called only once to retrieve all the data.

The method **IpfcExternalDataAccess.ListClasses()** returns a sequence of **IpfcExternalDataClasses** registered in the model. The method **IpfcExternalDataClass.ListSlots()** provide a sequence of **IpfcExternalDataSlots** existing for each class.

To find out a data type of a `IpfcExternalData`, call **IpfcExternalData.discr** and then call one of these properties to get the data, depending on the data type:

- `IpfcExternalData.IntegerValue`
- `IpfcExternalData.DoubleValue`
- `IpfcExternalData.StringValue`

### Example code:

This function demonstrates the usage of external data. It provides utility methods to get a VB hashtable from a model's external data.

```
Public Function retrieveExternalData(ByRef session As IpfcBaseSession,
                                   _ByVal className As String) As
                                   Hashtable

    Dim model As IpfcModel
    Dim dataAccess As IpfcExternalDataAccess
    Dim dataClass As IpfcExternalDataClass
    Dim slots As IpfcExternalDataSlots
    Dim i As Integer
    Dim table As Hashtable
    Dim value As Object
    Dim data As IpfcExternalData
    Dim slot As IpfcExternalDataSlot

    Try
'=====
'Get the current solid
'=====
        model = session.CurrentModel
        If model Is Nothing Then
            Throw New Exception("Model not present")
        End If
```

```

        table = New Hashtable
'=====  

'Get the external data class  

'=====  

        dataAccess = model.AccessExternalData()  

        dataClass = getClassByName(dataAccess, className)  
  

        If Not dataClass Is Nothing Then  

            slots = dataClass.ListSlots()  

'=====  

'Loop through all the slots  

'=====  

            For i = 0 To slots.Count - 1  

                value = Nothing  

                slot = slots.Item(i)  

'=====  

'Assign value to the object  

'=====  

                data = slot.Value  

                Select Case data.discr  

                    Case EpfcExternalDataType.EpfcEXTDATA_STRING  

                        value = CType(data.StringValue, Object)  

                    Case EpfcExternalDataType.EpfcEXTDATA_INTEGER  

                        value = CType(data.IntegerValue, Object)  

                    Case EpfcExternalDataType.EpfcEXTDATA_DOUBLE  

                        value = CType(data.DoubleValue, Object)  

                End Select  
  

                table.Add(slot.Name, value)  

            Next  

        End If  
  

        Return table  
  

    Catch ex As Exception  

        MsgBox(ex.Message.ToString + Chr(13) + ex.StackTrace.ToString)  

        Return nothing  

    End Try  

End Function  

'=====  

'Function      :   getClassByName  

'Purpose      :   This utility method returns a class, given its name.  

'=====  

    Private Function getClassByName(ByVal dataAccess As IpfcExternalDataAccess,  

    _ByVal className As String)  

    _As IpfcExternalDataClass  
  

        Dim classes As IpfcExternalDataClasses  

        Dim i As Integer  
  

        classes = dataAccess.ListClasses()  

        For i = 0 To classes.Count - 1  

            If classes.Item(i).Name = className Then  

                Return (classes.Item(i))  

            End If  

        Next  

        Return nothing  

    End Function

```

```

' =====
'Function      :   getSlotByName
'Purpose      :   This utility method returns a slot, given its name.
' =====
Private Function getSlotByName(ByVal extClass As IpfcExternalDataClass,
_ByVal slotName As String) _As IpfcExternalDataSlot
    Dim extSlots As IpfcExternalDataSlots
    Dim i As Integer

    extSlots = extClass.ListSlots()

    For i = 0 To extSlots.Count - 1
        If extSlots.Item(i).Name = slotName Then
            Return (extSlots.Item(i))
        End If
    Next
    Return nothing
End Function

```

## Exceptions

Most exceptions thrown by external data methods in The VB API extend `IpfcXExternalDataError`, which is a subclass of `IpfcXToolkitError`.

An additional exception thrown by external data methods is `IpfcXBadExternalData`. This exception signals an error accessing data. For example, external data access might have been terminated or the model might contain stream data from Pro/TOOLKIT.

The following table lists these exceptions.

Exception	Cause
<b>IpfcXExternalDataInvalidObject</b>	Generated when a model or class is invalid.
<b>IpfcXExternalDataClassOrSlotExists</b>	Generated when creating a class or slot and the proposed class or slot already exists.
<b>IpfcXExternalDataNamesTooLong</b>	Generated when a class or slot name is too long.
<b>IpfcXExternalDataSlotNotFound</b>	Generated when a specified class or slot does not exist.
<b>IpfcXExternalDataEmptySlot</b>	Generated when the slot you are attempting to read is empty.
<b>IpfcXExternalDataInvalidSlotName</b>	Generated when a specified slot name is invalid.
<b>IpfcXBadGetExternalData</b>	Generated when you try to access an incorrect data type in a <code>pfcExternalData</code> object.



# Windchill Connectivity APIs

---

Pro/ENGINEER has the capability to be directly connected to Windchill solutions, including Windchill Foundation, ProjectLink, PDMLink, and Windchill ProductPoint servers. This access allows users to manage and control the product data seamlessly from within Pro/ENGINEER.

This section lists the VB APIs that support Windchill servers and server operations in a connected Pro/ENGINEER session.

## Topic

[Introduction](#)

[Accessing a Windchill Server from a Pro/ENGINEER Session](#)

[Accessing Workspaces](#)

[Workflow to Register a Server](#)

[Aliased URL](#)

[Server Operations](#)

[Utility APIs](#)

[Sample Batch Workflow](#)

## Introduction

The methods introduced in this section provide support for the basic Windchill server operations from within Pro/ENGINEER. With these methods, operations such as registering a Windchill server, managing workspaces, and check in or check out of objects will be possible via the VB API. The capabilities of these APIs are similar to the operations available from within the Pro/ENGINEER Wildfire client, with some restrictions.

Windchill ProductPoint does not have the concept of a workspace. New objects are directly stored to a user-specified folder in the Windchill ProductPoint server. New iteration of the objects are stored in the same folder as the previous iteration. Hence some of the APIs related to Workspace operations may not be supported for customizations using Windchill ProductPoint.

## Non-Interactive Mode Operations

Some of the APIs specified in this section operate only in batch mode and cannot be used in the normal Pro/ENGINEER interactive mode. This restriction is mainly centered around the VB API registered servers, that is, servers registered by the VB API are not available in the Pro/ENGINEER Server Registry or in other locations in the Pro/ENGINEER user interface such as the Folder Navigator and embedded browser. If a VB API customization requires the user to have interactive access to the server, the server must be registered via the normal Pro/ENGINEER techniques, that is, either by entry in the Server Registry or by automatic registration of a previously registered server.

All of these APIs are supported from a non-interactive, that is, batch mode application or asynchronous application. For more information about batch mode and asynchronous mode, refer to the section "[VB API Fundamentals:Controlling Pro/ENGINEER](#)".

## Accessing a Windchill Server from a Pro/ENGINEER Session

Pro/ENGINEER allows you to register Windchill servers as a connection between the Windchill database and Pro/ENGINEER. Although the represented Windchill database can be from Windchill Foundation, Windchill ProjectLink, Windchill PDMLink, or Windchill ProductPoint, all types of databases are represented in the same way.

You can use the following identifiers when referring to Windchill servers in the VB API:

- Codebase URL--This is the root portion of the URL that is used to connect to a Windchill server. For example `http://wcserver.company.com/Windchill`.
- Server Alias--A server alias is used to refer to the server after it has been registered. The alias is also used to construct paths to files in the server workspaces and commonspaces. The server alias is chosen by the user or application and it need not have any direct relationship to the codebase URL. An alias can be any normal name, such as `my_alias`.

## Accessing Information Before Registering a Server

To start working with a Windchill server, you must establish a connection by registering the server in Pro/ENGINEER. The methods described in this section allow you to connect to a Windchill server and access information related to the server.

Methods and Properties Introduced:

- **IpfcBaseSession.AuthenticateBrowser()**
- **IpfcBaseSession.GetServerLocation()**
- **IpfcServerLocation.Class**
- **IpfcServerLocation.Location**
- **IpfcServerLocation.Version**
- **IpfcServerLocation.ListContexts()**
- **IpfcServerLocation.CollectWorkspaces()**

Use the method **IpfcBaseSession.AuthenticateBrowser()** to set the authentication context using a valid username and password. A successful call to this method allows the Pro/ENGINEER session to register with any server that accepts the username and password combination. A successful call to this method also ensures that an authentication dialog box does not appear during the registration process. You can call this method any number of times to set the authentication context for any number of Windchill servers, provided that you register the appropriate servers or servers immediately after setting the context.

The property **IpfcServerLocation.Location** specifies a `pfcServer`. `ServerLocation` object representing the codebase URL for a possible server. The server may not have been registered yet, but you can use this object and the methods it contains to gather information about the server prior to registration.

The property **IpfcServerLocation.Class** specifies the class of the server or server location. The values are:

- Windchill--Denotes either a Windchill Classic PDM server or a Windchill PDMLink server.
- ProjectLink--Denotes Windchill ProjectLink type of servers.
- productpoint--Denotes a Windchill ProductPoint server.

The property **IpfcServerLocation.Version** specifies the version of Windchill that is configured on the server or server location, for example, "7.0" or "8.0." This method accepts the server codebase URL as the input.

The method **IpfcServerLocation.ListContexts()** gives a list of all the available contexts for a specified server. A context is used to associate a workspace with a

product, project, or library.

The method **IpfcServerLocation.CollectWorkspaces()** returns the list of available workspaces for the specified server. The workspace objects returned contain the name of each workspace and its context.

**Note:**

This method is not supported for Windchill ProductPoint.

## Registering and Activating a Server

The methods described in this section are restricted to the non-interactive mode only. Refer to the section, [Non-Interactive Mode Operations](#), for more information.

Methods Introduced:

- **IpfcBaseSession.RegisterServer()**
- **IpfcServer.Activate()**
- **IpfcServer.Unregister()**

The method **IpfcBaseSession.RegisterServer()** registers the specified server with the codebase URL. A successful call to **IpfcBaseSession.AuthenticateBrowser()** with a valid username and password is essential for **IpfcBaseSession.RegisterServer()** to register the server without launching the authentication dialog box. Registration of the server establishes the server alias. You must designate an existing workspace to use when registering the server. After the server has been registered, you may create a new workspace.

**Note:**

While working with the Windchill ProductPoint server, specify the value of the input argument `WorkspaceName` as `NULL` for this method.

The method **IpfcServer.Activate()** sets the specified server as the active server in the Pro/ENGINEER session.

The method **IpfcServer.Unregister()** unregisters the specified server. This is similar to **Server Registry>Delete** through the user interface.

## Accessing Information From a Registered Server

Properties Introduced:

- **IpfcServer.IsActive**
- **IpfcServer.Alias**
- **IpfcServer.Context**
- **IpfcWPPServer.GetServerTargetfolder()**
- **IpfcWPPServer.SetServerTargetfolder()**

The property **IpfcServer.IsActive** specifies if the server is active.

The property **IpfcServer.Alias** returns the alias of a server if you specify the codebase URL.

The property **IpfcServer.Context** returns the active context of the active server.

**Note:**

This function is not supported while working with a Windchill ProductPoint server.

The method **IpfcWPPServer.GetServerTargetfolder()** returns a location on the Windchill ProductPoint server where you can save new product items. Specify the location of the target folder on the Windchill ProductPoint server using the method **IpfcWPPServer.SetServerTargetfolder()**. These methods are applicable only when working with a Windchill ProductPoint server.

## Information on Servers in Session

Methods Introduced:

- **IpfcBaseSession.GetActiveServer()**
- **IpfcBaseSession.GetServerByAlias()**
- **IpfcBaseSession.GetServerByUrl()**
- **IpfcBaseSession.ListServers()**

The method **IpfcBaseSession.GetActiveServer()** returns returns the active server handle.

The method **IpfcBaseSession.GetServerByAlias()** returns the handle to the server matching the given server alias, if it exists in session.

The method **IpfcBaseSession.GetServerByUrl()** returns the handle to the server matching the given server URL and workspace name, if it exists in session.

The method **IpfcBaseSession.ListServers()** returns a list of servers registered in this session.

## Accessing Workspaces

For every workspace, a new distinct storage location is maintained in the user's personal folder on the server (server-side workspace) and on the client (client-side workspace cache). Together, the server-side workspace and the client-side workspace cache make up the workspace.

### Note:

Windchill ProductPoint does not have the concept of a workspace or active workspace. Therefore, many methods in this section are not applicable for this product.

Methods and Properties Introduced:

- **CCpfcWorkspaceDefinition.Create()**
- **IpfcWorkspaceDefinition.WorkspaceName**
- **IpfcWorkspaceDefinition.WorkspaceContext**

The class **IpfcWorkspaceDefinition** contains the name and context of the workspace. The method **IpfcServerLocation.CollectWorkspaces()** returns an array of workspace data. Workspace data is also required for the method **IpfcServer.CreateWorkspace()** to create a workspace with a given name and a specific context.

The method **CCpfcWorkspaceDefinition.Create()** creates a new workspace definition object suitable for use when creating a new workspace on the server.

The property **IpfcWorkspaceDefinition.WorkspaceName** retrieves the name of the workspace.

The property **IpfcWorkspaceDefinition.WorkspaceContext** retrieves the context of the workspace.

## Creating and Modifying the Workspace

Methods and Properties Introduced:

- **IpfcServer.CreateWorkspace()**
- **IpfcServer.ActiveWorkspace**
- **IpfcServerLocation.DeleteWorkspace()**

All methods and properties described in this section, except **IpfcServer.ActiveWorkspace**, are permitted only in the non-interactive mode. Refer to the section, [Non-Interactive Mode Operations](#), for more information.

The method **IpfcServer.CreateWorkspace()** creates and activates a new workspace.

The property **IpfcServer.ActiveWorkspace** retrieves the name of the active workspace.

The method **IpfcServerLocation.DeleteWorkspace()** deletes the specified workspace. The method deletes the workspace only if the following conditions are met:

- The workspace is not the active workspace.
- The workspace does not contain any checked out objects.

Use one of the following techniques to delete an active workspace:

- Make the required workspace inactive using **IpfcServer.ActiveWorkspace** with the name of some other workspace and then call **IpfcServerLocation.DeleteWorkspace**.
- Unregister the server using **IpfcServer.Unregister()** and delete the workspace.

## Workflow to Register a Server

### To Register a Server with an Existing Workspace

Perform the following steps to register a Windchill server with an existing workspace:

1. Set the appropriate authentication context using the method `IpfcBaseSession.AuthenticateBrowser()` with a valid username and password.
2. Look up the list of workspaces using the method `IpfcServerLocation.CollectWorkspaces()`. If you already know the name of the workspace on the server, then ignore this step.
3. Register the workspace using the method `IpfcBaseSession.RegisterServer()` with an existing workspace name on the server.
4. Activate the server using the method `IpfcServer.Activate()`.

## To Register a Server with a New Workspace

Perform the following steps to register a Windchill server with a new workspace:

1. Perform steps 1 to 4 in the preceding section to register the Windchill server with an existing workspace.
2. Use the method `IpfcServerLocation.ListContexts()` to choose the required context for the server.
3. Create a new workspace with the required context using the method `IpfcServer.CreateWorkspace()`. This method automatically makes the created workspace active.

### Note:

You can create a workspace only after the server is registered.

## Aliased URL

An aliased URL serves as a handle to the server objects. You can access the server objects in the commonspace (shared folders) and the workspace using an aliased URL. An aliased URL is a unique identifier for the server object and its format is as follows:

- Object in workspace has a prefix `wtws`

```
wtws://<server_alias>/<workspace_name>/<object_server_name>
```

where `<object_server_name>` includes `<object_name>.<object_extension>`

For example, `wtws://my_server/my_workspace/abcd.prt`, `wtws://`

my\_server/my\_workspace/intf\_file.igs

where

<server\_alias> is my\_server

<workspace\_name> is my\_workspace

- Object in commonspace has a prefix wtpub

wtpub://<server\_alias>/<folder\_location>/<object\_server\_name>

For example, wtpub://my\_server/path/to/cs\_folder/abcd.prt

where

<server\_alias> is my\_server

<folder\_location> is path/to/cs\_folder

**Note:**

- object\_server\_name must be in lowercase.
  - The APIs are case-sensitive to the aliased URL.
  - <object\_extension> should not contain Pro/ENGINEER versions, for example, .1 or .2, and so on.
- For Windchill ProductPoint servers, you can specify a large number of URL variations as long as the base server URL is included. For example,
    - wpp://<Server\_Alias>/ProdA/ProENGINEER/Document/jan.prt
    - <Server\_Alias>/ProdA/ProENGINEER/Documents/jan.prt

You can also specify only the part name and the object will be accessed from the server, if the server is the default location being explored.

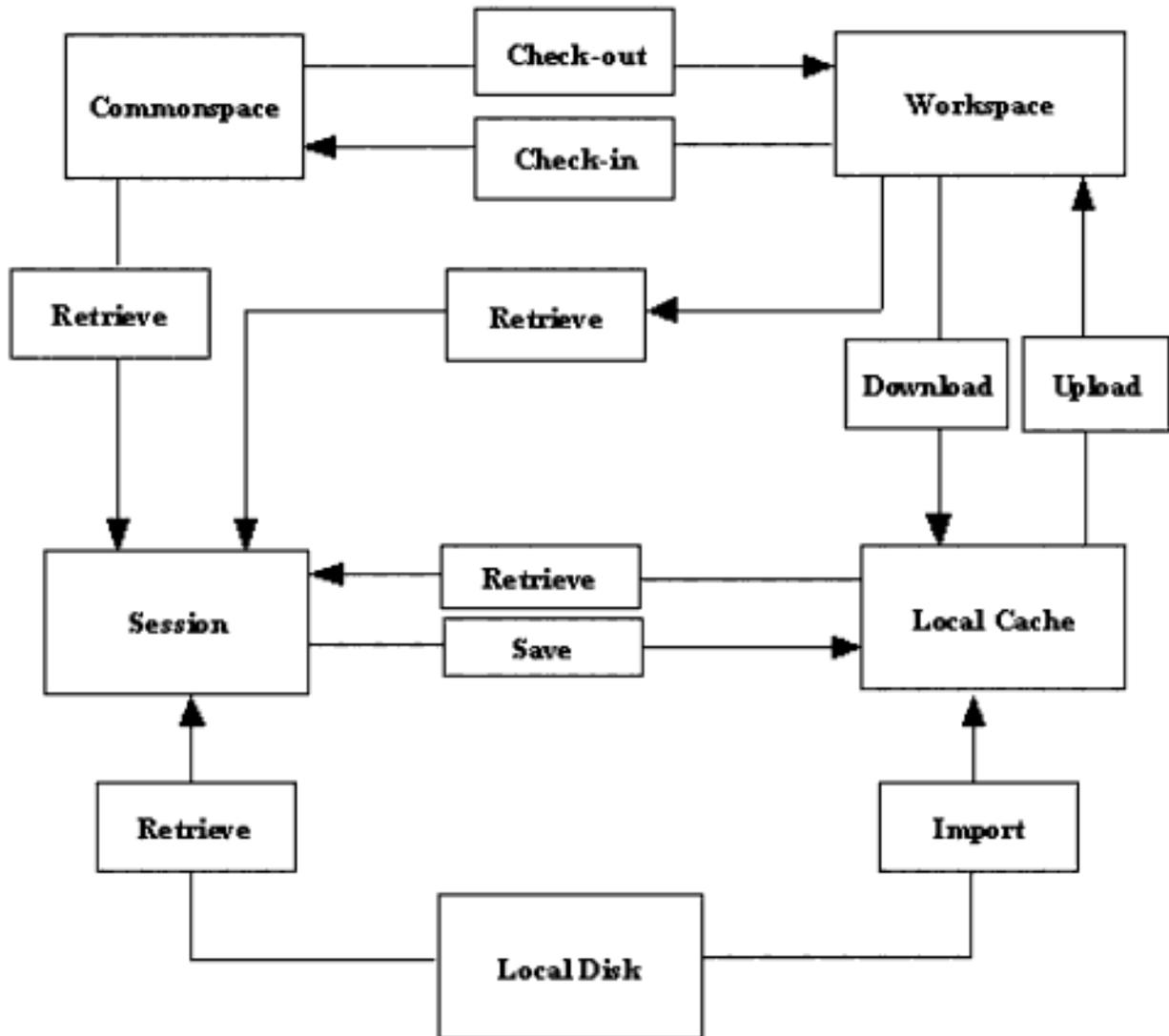
## Server Operations

After registering the Windchill server with Pro/ENGINEER, you can start accessing the data on the Windchill servers. The Pro/ENGINEER interaction with Windchill servers leverages the following locations:

- Commonsense (Shared folders)

- Workspace (Server-side workspace)
- Workspace local cache (Client-side workspace)
- Pro/ENGINEER session
- Local disk

The methods described in this section enable you to perform the basic server operations. The following illustration shows how data is transferred among these locations.



## Save

Methods and Property Introduced:

- **IpfcModel.Save()**
- **CCpfcServerSynchronizeConflict.Create()**

- **IpfcWPPServer.SynchronizeServer()**
- **IpfcServerSynchronizeConflict.Description**
- **IpfcWPPServer.GetServerSynchronizationState()**

The method **IpfcModel.Save()** stores the object from the session in the local workspace cache, when a server is active. For Windchill ProductPoint servers, this method saves an existing model to the location from where it was retrieved. To save a new object to a specified location on the ProductPoint server, first use the method **IpfcWPPServer.SetServerTargetfolder()** to set the target folder location on the server and then call the method **IpfcModel.Model.Save**. If you do not set the target folder location, the method **IpfcModel.Model.Save** saves the new objects to the top-level folder of the active product or context.

The method **CCIpfcServerSynchronizeConflict.Create()** creates the **ServerSynchronizeConflicts** object containing the description of the conflicts encountered during server synchronization.

The method **IpfcWPPServer.SynchronizeServer()** synchronizes the objects in the local cache with the contents in the Windchill ProductPoint server. Specify `NULL` as the value of the input synchronization options. The method returns the synchronization conflict object created by the method **CCIpfcServerSynchronizeConflict.Create()**. Use the property **IpfcServerSynchronizeConflict.Description** to access the description of the synchronization conflict.

The method **IpfcWPPServer.GetServerSynchronizationState()** specifies if the contents of the Windchill ProductPoint server are synchronized with the local cache. This method returns true if the server is synchronized, and false, if otherwise.

## Upload

An upload transfers Pro/ENGINEER files and any other dependencies from the local workspace cache to the server-side workspace.

Methods Introduced:

- **IpfcServer.UploadObjects()**
- **IpfcServer.UploadObjectsWithOptions()**
- **CCIpfcUploadOptions.Create()**

The method **IpfcServer.UploadObjects()** uploads the object to the workspace. The object to be uploaded must be present in the current Pro/ENGINEER session. You must save the object to the workspace using **pfcModel.Model.Save, or import it into the workspace using IpfcBaseSession.ImportToCurrentWS()** before attempting to upload it.

The method **IpfcServer.UploadObjectsWithOptions()** uploads objects to the workspace using the options specified in the `IpfcUploadOptions` interface. These options allow you to upload the entire workspace, auto-resolve missing references, and indicate the target folder location for the new content during the upload. You must save the object to the workspace using **pfcModel.Model.Save, or import it to the workspace using IpfcBaseSession.ImportToCurrentWS()** before attempting to upload it.

Create the `IpfcUploadOptions` object using the method **CCpfcUploadOptions.Create()**.

The methods available for setting the upload options are described in the following section.

## CheckIn

After you have finished working on objects in your workspace, you can share the design changes with other users. The checkin operation copies the information and files associated with all changed objects from the workspace to the Windchill database.

### Note:

The methods described in this section are not applicable to Windchill ProductPoint server operations.

Methods and Properties Introduced:

- **IpfcServer.CheckinObjects()**
- **CCpfcCheckinOptions.Create()**
- **IpfcUploadBaseOptions.DefaultFolder**
- **IpfcUploadBaseOptions.NonDefaultFolderAssignments**
- **IpfcUploadBaseOptions.AutoresolveOption**

- **IpfcCheckinOptions.BaselineName**
- **IpfcCheckinOptions.BaselineNumber**
- **IpfcCheckinOptions.BaselineLocation**
- **IpfcCheckinOptions.BaselineLifecycle**
- **IpfcCheckinOptions.KeepCheckedout**

The method **IpfcServer.CheckinObjects()** checks in an object into the database. The object to be checked in must be present in the current Pro/ENGINEER session. Changes made to the object are not included unless you save the object to the workspace using the method **IpfcModel.Save()** before you check it in.

If you pass `NULL` as the value of the *options* parameter, the checkin operation is similar to the **Auto Check-In** option in Pro/ENGINEER. For more details on **Auto Check-In**, refer to the online help for Pro/ENGINEER.

Use the method **CCpfcCheckinOptions.Create()** to create a new `IpfcCheckinOptions` object.

By using an appropriately constructed *options* argument, you can control the checkin operation. Use the APIs listed above to access and modify the checkin options. The checkin options are as follows:

- **DefaultFolder**--Specifies the default folder location on the server for the automatic checkin operation.
- **NonDefaultFolderAssignment**--Specifies the folder location on the server to which the objects will be checked in.
- **AutoresolveOption**--Specifies the option used for auto-resolving missing references. These options are defined in the `EpfcServerAutoresolveOption` enumerated type, and are as follows:
  - `EpfcSERVER_DONT_AUTORESOLVE`--Model references missing from the workspace are not automatically resolved. This may result in a conflict upon checkin. This option is used by default.
  - `EpfcSERVER_AUTORESOLVE_IGNORE`--Missing references are automatically resolved by ignoring them.
  - `EpfcSERVER_AUTORESOLVE_UPDATE_IGNORE`--Missing references are automatically resolved by updating them in the database and ignoring them if not found.
- **Baseline**--Specifies the baseline information for the objects upon checkin. The baseline information for a checkin operation is as follows:

- **BaselineName**--Specifies the name of the baseline.
- **BaselineNumber**--Specifies the number of the baseline.

The default format for the baseline name and baseline number is "Username + time (GMT) in milliseconds"

- **BaselineLocation**--Specifies the location of the baseline.
- **BaselineLifecycle**--Specifies the name of the lifecycle.
- o **KeepCheckedout**--If the value specified is true, then the contents of the selected object are checked into the Windchill server and automatically checked out again for further modification.

## Retrieval

Standard VB API provides several methods that are capable of retrieving models. When using these methods with Windchill servers, remember that these methods do not check out the object to allow modifications.

Methods Introduced:

- **IpfcBaseSession.RetrieveModel()**
- **IpfcBaseSession.RetrieveModelWithOpts()**
- **IpfcBaseSession.OpenFile()**

The methods **IpfcBaseSession.RetrieveModel()**, **IpfcBaseSession.RetrieveModelWithOpts()**, and **IpfcBaseSession.OpenFile()** load an object into a session given its name and type. The methods search for the object in the active workspace, the local directory, and any other paths specified by the `search_path` configuration option. For Windchill ProductPoint servers, the method **pfcSession.BaseSession.RetrieveModelWithOpts** supports the `instance<generic>` notation for the name of the object.

## Checkout and Download

To modify an object from the commonspace, you must check out the object. The process of checking out communicates your intention to modify a design to the Windchill server. The object in the database is locked, so that other users can obtain read-only copies of the object, and are prevented from modifying the object while you have checked it out.

Checkout is often accompanied by a download action, where the objects are brought

from the server-side workspace to the local workspace cache. In The VB API, both operations are covered by the same set of methods.

**Note:**

The methods described in this section are not applicable to Windchill ProductPoint server operations.

Methods and Properties Introduced:

- **IpfcServer.CheckoutObjects()**
- **IpfcServer.CheckoutMultipleObjects()**
- **CCpfcCheckoutOptions.Create()**
- **IpfcCheckoutOptions.Dependency**
- **IpfcCheckoutOptions.SelectedIncludes**
- **IpfcCheckoutOptions.IncludeInstances**
- **IpfcCheckoutOptions.Version**
- **IpfcCheckoutOptions.Download**
- **IpfcCheckoutOptions.ReadOnly**

The method **IpfcServer.CheckoutObjects()** checks out and optionally downloads the object to the workspace based on the configuration specifications of the workspace.

The input arguments of this method are as follows:

- Mdl--Specifies the object to be checked out. This is applicable if the model has already been retrieved without checking it out.
- File--Specifies the top-level object to be checked out.
- Checkout--The checkout flag. If you specify the value of this argument as true, the selected object is checked out. Otherwise, the object is downloaded without being checked out. The download action enables you to bring read-only copies of objects into your workspace. This allows you to examine the object without locking it.
- Options--Specifies the checkout options object. If you pass NULL as the value of this argument, then the default Pro/ENGINEER checkout rules apply. Use the method **CCpfcCheckoutOptions.Create()** to create a new **IpfcCheckoutOptions** object.

Use the method **IpfcServer.CheckoutMultipleObjects()** to check out and download multiple objects to the workspace based on the configuration specifications of the workspace. This method takes the same input arguments as listed above, except for *Mdl and File*. Instead it takes the argument *Files* that specifies the sequence of the objects to check out or download.

By using an appropriately constructed *options* argument in the above functions, you can control the checkout operation. Use the APIs listed above to modify the checkout options. The checkout options are as follows:

- Dependency--Specifies the dependency rule used while checking out dependents of the object selected for checkout. The types of dependencies given by the `IpfcServerDependency` enumerated type are as follows:
  - `EpfcSERVER_DEPENDENCY_ALL`--All objects that are dependent on the selected object are checked out.
  - `EpfcSERVER_DEPENDENCY_REQUIRED`--All models required to successfully retrieve the originally selected model from the CAD application are selected for checkout.
  - `EpfcSERVER_DEPENDENCY_NONE`--None of the dependent objects are checked out.
- `IncludeInstances`--Specifies the rule for including instances from the family table during checkout. The type of instances given by the `IpfcServerIncludeInstances` enumerated type are as follows:
  - `EpfcSERVER_INCLUDE_ALL`--All the instances of the selected object are checked out.
  - `EpfcSERVER_INCLUDE_SELECTED`--The application can select the family table instance members to be included during checkout.
  - `EpfcSERVER_INCLUDE_NONE`--No additional instances from the family table are added to the object list.
- `SelectedIncludes`--Specifies the sequence of URLs to the selected instances, if `IncludeInstances` is of type `EpfcSERVER_INCLUDE_SELECTED`.
- `Version`--Specifies the version of the checked out object. If this value is set to `NULL`, the object is checked out according to the current workspace configuration.
- `Download`--Specifies the checkout type as download or link. The value `download` specifies that the object content is downloaded and checked out, while `link` specifies that only the metadata is downloaded and checked out.
- `Readonly`--Specifies the checkout type as a read-only checkout. This option is applicable only if the checkout type is `link`.

The following truth table explains the dependencies of the different control factors in the method **IpfcServer.CheckoutObjects()** and the effect of different combinations on the end result.

<b>Argument <i>checkout</i> in IpfcServer.CheckoutObjects()</b>	<b>pfcServer.CheckoutOptions.SetDownload</b>	<b>pfcServer.CheckoutOptions.SetReadOnly</b>	<b>Result</b>
true	true	NA	Object is checked out and its content is downloaded.
true	true	NA	Object is checked out but content is not downloaded.
false	NA	true	Object is downloaded without checkout.
false	NA	false	Not supported

## Undo Checkout

Method Introduced:

- **IpfcServer.UndoCheckout()**

Use the method **IpfcServer.UndoCheckout()** to undo a checkout of the specified object. When you undo a checkout, the changes that you have made to the content and metadata of the object are discarded and the content, as stored in the server, is downloaded to the workspace. This method is applicable only for the model in the active Pro/ENGINEER session.

## Import and Export

The VB API provides you with the capability of transferring specified objects to and from a workspace. Import and export operations must take place in a session with no

models. An import operation transfers a file from the local disk to the workspace.

Methods and Properties Introduced:

- **IpfcWPPServer.SetWsimpexFolderoption()**
- **IpfcBaseSession.ExportFromCurrentWS()**
- **IpfcBaseSession.ImportToCurrentWS()**
- **IpfcWSImportExportMessage.Description**
- **IpfcWSImportExportMessage.FileName**
- **IpfcWSImportExportMessage.MessageType**
- **IpfcWSImportExportMessage.Resolution**
- **IpfcWSImportExportMessage.Succeeded**
- **IpfcBaseSession.SetWSExportOptions()**
- **CCpfcWSExportOptions.Create()**
- **IpfcWSExportOptions.IncludeSecondaryContent**

The method **IpfcWPPServer.SetWsimpexFolderoption()** sets the target folder to import data or the source folder to the Windchill ProductPoint servers or to export data from these servers. Set the target folder location using this method before calls to **IpfcBaseSession.ExportFromCurrentWS()** and **IpfcBaseSession.ImportToCurrentWS()**. This function is used for Windchill ProductPoint servers only.

The method **IpfcBaseSession.ExportFromCurrentWS()** exports the specified objects from the current workspace to a disk in a linked session of Pro/ENGINEER. For Windchill ProductPoint servers this function exports files from the specified source folder location on the server to a disk.

The method **IpfcBaseSession.ImportToCurrentWS()** imports the specified objects from a disk to the current workspace in a linked session of Pro/ENGINEER. For Windchill ProductPoint servers this method copies files from the local disk to the specified target folder location on the server.

Both **IpfcBaseSession.ExportFromCurrentWS()** and **IpfcBaseSession.ImportToCurrentWS()** allow you to specify a dependency criterion to process the following items:

- All external dependencies
- Only required dependencies
- No external dependencies

Both **IpfcBaseSession.ExportFromCurrentWS()** and **IpfcBaseSession.ImportToCurrentWS()** return the messages generated during the export or import operation in the form of the `IpfcWSImportExportMessages` object. Use the APIs listed above to access the contents of a message. The message specified by the `IpfcWSImportExportMessage` object contains the following items:

- **Description**--Specifies the description of the problem or the message information.
- **FileName**--Specifies the object name or the name of the object path.
- **MessageType**--Specifies the severity of the message in the form of the `EpfcWSImportExportMessageType` enumerated type. The severity is one of the following types:
  - `EpfcWSIMPEX_MSG_INFO`--Specifies an informational type of message.
  - `EpfcWSIMPEX_MSG_WARNING`--Specifies a low severity problem that can be resolved according to the configured rules.
  - `EpfcWSIMPEX_MSG_CONFLICT`--Specifies a conflict that can be overridden.
  - `EpfcWSIMPEX_MSG_ERROR`--Specifies a conflict that cannot be overridden or a serious problem that prevents processing of an object.
- **Resolution**--Specifies the resolution applied to resolve a conflict that can be overridden. This is applicable when the message is of the type `WSIMPEX_MSG_CONFLICT`.
- **Succeeded**--Determines whether the resolution succeeded or not. This is applicable when the message is of the type `EpfcWSIMPEX_MSG_CONFLICT`.

The method **IpfcBaseSession.SetWSExportOptions()** sets the export options used while exporting the objects from a workspace in the form of the `IpfcWSExportOptions` object. Create this object using the method **CCpfcWSExportOptions.Create()**. The export options are as follows:

- **Include Secondary Content**--Indicates whether or not to include secondary content while exporting the primary Pro/ENGINEER model files. Use the property `IpfcWSExportOptions.IncludeSecondaryContent` to set this option.

## File Copy

The VB API provides you with the capability of copying a file from the workspace or target folder to a location on the disk and vice-versa.

Methods Introduced:

- **IpfcBaseSession.CopyFileToWS()**
- **IpfcBaseSession.CopyFileFromWS()**

Use the method **IpfcBaseSession.CopyFileToWS()** to copy a file from the disk to the workspace. The file can optionally be added as secondary content to a given workspace file. For Windchill ProductPoint servers, use this method to copy a viewable file from disk as a new item in the target folder specified by the method **IpfcWPPServer.SetServerTargetfolder()**. If the viewable file is added as secondary content, a dependency is created between the Pro/ENGINEER model and the viewable file.

Use the method **IpfcBaseSession.CopyFileFromWS()** to copy a file from the workspace to a location on disk. For Windchill ProductPoint servers, use this method to copy a single file from the current target folder specified by the method **IpfcWPPServer.SetServerTargetfolder()** to the local disk.

When importing or exporting Pro/ENGINEER models, PTC recommends that you use methods **IpfcBaseSession.ImportToCurrentWS()** and **IpfcBaseSession.ExportFromCurrentWS()**, respectively, to perform the import or export operation. Methods that copy individual files do not traverse Pro/ENGINEER model dependencies, and therefore do not copy a fully retrievable set of models at the same time.

Additionally, only the methods **pfcSession.BaseSession.ImportToCurrentWS** and **pfcSession.BaseSession.ExportFromCurrentWS** provide full metadata exchange and support. That means **pfcSession.BaseSession.ImportToCurrentWS** can communicate all the Pro/ENGINEER designated parameters, dependencies, and family table information to a PDM system while **pfcSession.BaseSession.ExportFromCurrentWS** can update exported Pro/ENGINEER data with PDM system changes to designated and system parameters, dependencies, and family table information. Hence PTC recommends the use of **IpfcBaseSession.CopyFileToWS()** and **IpfcBaseSession.CopyFileFromWS()** to process only non-Pro/ENGINEER files.

## Server Object Status

Methods Introduced:

- **IpfcServer.IsObjectCheckedOut()**

- **IpfcServer.IsObjectModified()**

The methods described in this section verify the current status of the object in the workspace. The method **IpfcServer.IsObjectCheckedOut()** specifies whether the object is checked out for modification.

The method **IpfcServer.IsObjectModified()** specifies whether the object has been modified since checkout. This method returns the value `false` if newly created objects have not been uploaded.

**Note:**

These methods are not applicable for Windchill ProductPoint server operations.

## Object Lock Status

In comparison with other Windchill servers, Windchill ProductPoint does not have the concept of a workspace. This means that as soon as changes are saved to the server, they are visible to all users with access to work-in-progress versions. The functions described in this section enable you to establish an exclusive lock when modifying a server-managed part in Pro/ENGINEER. The functions described in this section are applicable only for Windchill ProductPoint server operations.

Methods and Properties Introduced:

- **IpfcWPPServer.LockServerObjects()**
- **CCpfcServerLockConflict.Create()**
- **IpfcServerLockConflict.ObjectName**
- **IpfcServerLockConflict.ConflictMessage**
- **IpfcWPPServer.GetServerObjectLockStatus()**
- **IpfcWPPServer.GetServerObjectsLockStatus()**
- **CCpfcServerLockStat.Create()**
- **IpfcServerLockStat.ObjectName**
- **IpfcServerLockStat.Status**

- **IpfcServerLockStat.StatusMessage**
- **IpfcWPPServer.UnlockServerObjects()**

The method **IpfcWPPServer.LockServerObjects()** establishes an explicit lock on the specified objects on the server. Specify the full path, name, and extension for the input objects. This method returns the `IpfcServerLockConflict` object that contains the details of conflicts, if any, that occurred during the lock operation. Use the property **IpfcServerLockConflict.ObjectName** to access the name of the object for which the lock conflict occurred. Use the property **IpfcServerLockConflict.ConflictMessage** to get details of the lock conflict.

The method **IpfcWPPServer.GetServerObjectLockStatus()** checks the lock status of the specified object on the Windchill ProductPoint server. Specify the full path, name, and extension for the input object. The method returns the `IpfcServerLockStat` object that contains information regarding the lock status.

The method **IpfcWPPServer.GetServerObjectsLockStatus()** checks the lock status of a set of objects on the Windchill ProductPoint server. Specify the full path, name, and extension for the input objects. The method returns an array of `IpfcServerLockStat` objects that contain information regarding the lock status of the input objects.

Use the property **IpfcServerLockStat.ObjectName** to access the name of the object, including the extension, for which the lock status is described.

Use the property **IpfcServerLockStat.Status** to access the status of the lock on the object on the server.

- **PRO\_OBJ\_LOCK\_STAT\_UNSET**--Specifies that no lock has been applied on the object.
- **PRO\_OBJ\_LOCK\_STAT\_HARDLOCK**--Specifies that objects are locked and that the current user is not the owner of the locks. Therefore, this user cannot modify or release the lock.
- **PRO\_OBJ\_LOCK\_STAT\_SOFTLOCK**--Specifies that the objects are locked and the current user is the owner of the locks. Therefore, this user can release the lock.
- **PRO\_OBJ\_LOCK\_STAT\_UNLOCKED**--Specifies that the explicit or implicit lock has been removed from the object and it is available for editing.

Use the property **IpfcServerLockStat.StatusMessage** to access the status of the object on the server. It provides the name of the user who locked the object and the time of locking.

The method **IpfcWPPServer.UnlockServerObjects()** unlocks a set of objects that have been explicitly locked on the product server. This method returns the `IpfcServerLockConflict` object that contains the details of conflicts, if any, that occurred during the unlock operation.

## Delete Objects

Method Introduced:

- **IpfcServer.RemoveObjects()**

The method **IpfcServer.RemoveObjects()** deletes the array of objects from the workspace. When passed with the *ModelNames* array as NULL, this method removes all the objects in the active workspace.

## Conflicts During Server Operations

An exception is provided to capture the error condition while performing the following server operations using the specified APIs:

Operation	API
Checkin an object or workspace	<code>IpfcServer.CheckinObjects()</code>
Checkout an object	<code>IpfcServer.CheckoutObjects()</code>
Undo checkout of an object	<code>IpfcServer.UndoCheckout()</code>
Upload object	<code>IpfcServer.UploadObjects()</code>
Download object	<code>IpfcServer.CheckoutObjects()</code> (with <code>download</code> as <code>true</code> )
Delete workspace	<code>IpfcServerLocation.DeleteWorkspace()</code>

Remove object	<code>IpfcServer.RemoveObjects()</code>
---------------	-----------------------------------------

These APIs throw a common exception **XToolkitCheckoutConflict** if an error is encountered during server operations. The exception description will include the details of the error condition. This description is similar to the description displayed by the Pro/ENGINEER HTML user interface in the conflict report.

## Utility APIs

The methods specified in this section enable you to obtain the handle to the server objects to access them. The handle may be the aliased URL or the model name of the http URL. These utilities enable the conversion of one type of handle to another.

Methods Introduced:

- **IpfcServer.GetAliasedUrl()**
- **IpfcBaseSession.GetModelNameFromAliasedUrl()**
- **IpfcBaseSession.GetAliasFromAliasedUrl()**
- **IpfcBaseSession.GetUrlFromAliasedUrl()**

The method **IpfcServer.GetAliasedUrl()** enables you to search for a server object by its name. Specify the complete filename of the object as the input, for example, `test_part.prt`. The method returns the aliased URL for a model on the server. For more information regarding the aliased URL, refer to the section [Aliased URL](#). During the search operation, the workspace takes precedence over the shared space.

You can also use this method to search for files that are not in the Pro/ENGINEER format. For example, `my_text.txt`, `prodev.dat`, `intf_file.stp`, and so on.

The method **IpfcBaseSession.GetModelNameFromAliasedUrl()** returns the name of the object from the given aliased URL on the server.

The method **IpfcBaseSession.GetUrlFromAliasedUrl()** converts an aliased URL to a standard URL for the objects on the server.

For example, `wtws://my_alias/Wildfire/abcd.prt` is converted to an appropriate URL on the server as `http://server.mycompany.com/Windchill`.

For Windchill ProductPoint, the aliased URL `wpp://<Server_Alias>/ProdA/ProENGINEER/Document/jan.prt` is converted to an appropriate URL on server, for example, `http://server.mycompany.com/`.

The method **`IpfcBaseSession.GetAliasFromAliasedUrl()`** returns the server alias from aliased URL.

## Sample Batch Workflow

A typical workflow using the Windchill APIs for an asynchronous non-graphical application is as follows:

1. Start a Pro/ENGINEER session using the method `pfcAsyncConnection.pfcAsyncConnection.AsyncConnection_Connect`.
2. Authenticate the browser using the method **`IpfcBaseSession.AuthenticateBrowser()`**.
3. Register the server with the new workspace using the method **`IpfcBaseSession.RegisterServer()`**.
4. Activate the server using the method **`IpfcServer.Activate()`**.
5. Check out and retrieve the model from the vault URL using the method **`IpfcServer.CheckoutObjects()`** followed by **`IpfcBaseSession.RetrieveModel()`**.
6. Modify the model according to the application logic.
7. Save the model to the workspace using the method **`IpfcModel.Save()`**.
8. Check in the modified model back to the server using the method **`IpfcServer.CheckinObjects()`**.
9. After processing all models, unregister from the server using the method **`IpfcServer.Unregister()`**.
10. Delete the workspace using **`IpfcServerLocation.DeleteWorkspace()`**.
11. Stop Pro/ENGINEER using the method **`IpfcAsyncConnection.End()`**.



# Sample Applications

---

This section lists the sample applications provided with the VB API.

## Topic

[Installing the VB API](#)

[Sample Applications](#)

## Installing the VB API

The VB API is available on the same CD as Pro/ENGINEER. When Pro/ENGINEER is installed using PTC.SetUp, one of the optional components is "API Toolkits". This includes Pro/TOOLKIT, J-Link, Pro/Web.Link, and Visual Basic API.

If you select Visual Basic API, a directory called `vbapi` is created under the Pro/ENGINEER loadpoint and the VB API is automatically installed in this directory. This directory contains all the libraries, example applications, and documentation specific to the VB API.

## Sample Applications

The VB API sample applications are available in the directories `vbapi_examples` and `vbapi_appls` under the root directory `vbapi`.

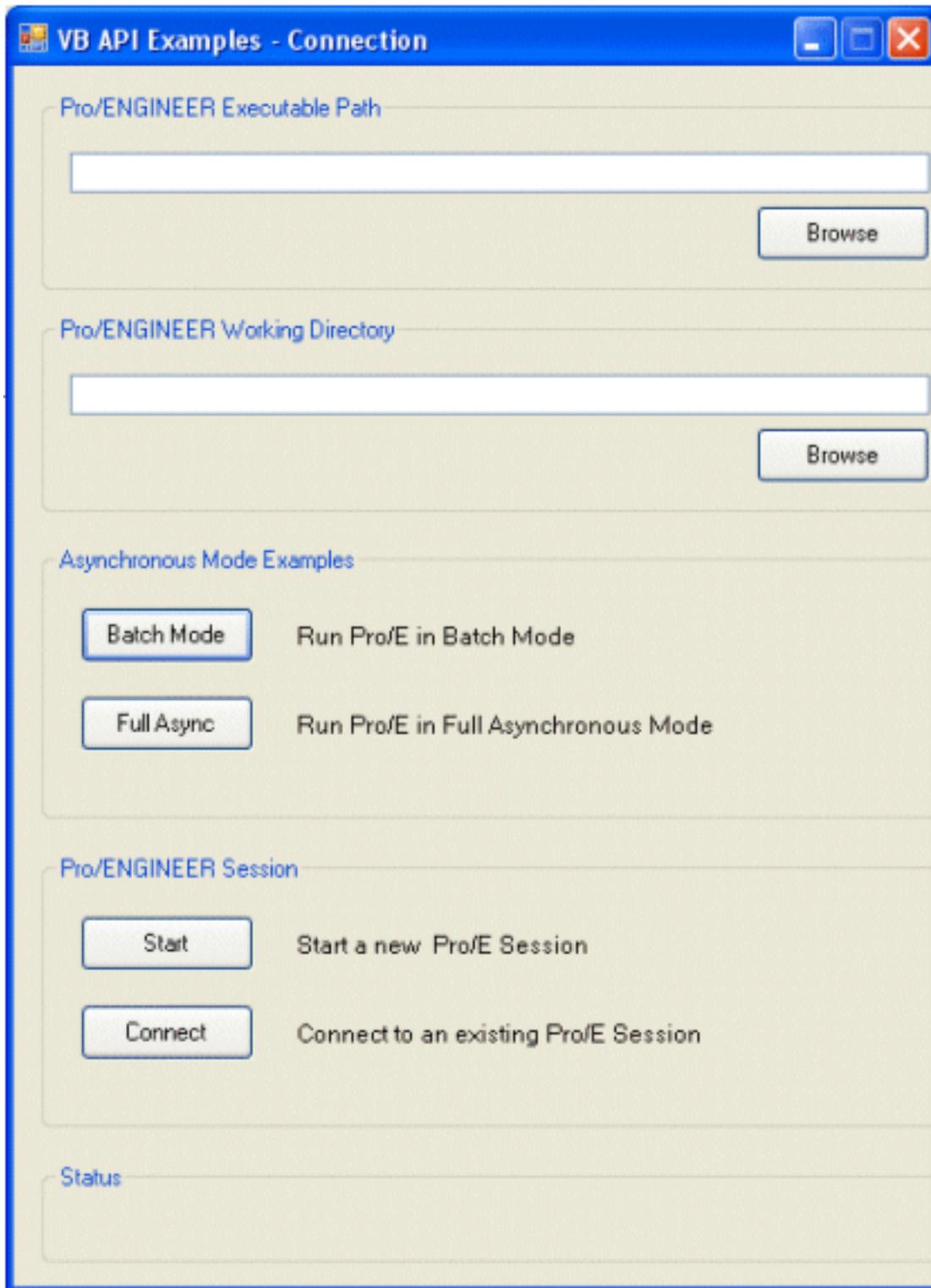
## VBAPIExamples

Location	Main Classes
<code>vbapi/ vbapi_examples</code>	<code>formConnection</code> and <code>formExamples</code>

The `vbapi_examples` directory is a collection of all the VB.NET example source files present in the VB API User's Guide. All the example source files are also available along with a VB.NET solution called `VB API Examples.sln` and a project file called `VB API Examples.vbproj` as a single `VBAPIExamples.zip` file in the same directory.

Set up and run the examples using the following procedure:

1. Set the `PRO_COMM_MSG_EXE` environment variable to the full path of the executable `pro_comm_msg.exe` for your application to communicate with Pro/ENGINEER. Typically, the path to the executable is `[Pro/E loadpoint]/[machine type]/obj/pro_comm_msg.exe`, where `machine_type` is `i486_nt` for 32-bit Windows and `x86_win64` for 64-bit Windows installations.
2. Register the COM server by running the `vb_api_register.bat` file located at `[Pro/E loadpoint]/bin`.
3. Unzip the `VBAPIExamples.zip` file in a local folder on your machine and open the `VB API Examples.sln` solution in Microsoft Visual Studio.
4. Set the COM reference for your project to **Pro/E VB API Type Library for Pro/E Wildfire 4.0**.
5. Build the solution and execute the `VB API Examples.exe` created in your local folder. The **VB API Examples - Connection** form as shown in the following figure is loaded.



6. Click **Start** to start a new Pro/ENGINEER session in the simple asynchronous mode. You must specify the Pro/ENGINEER working directory and executable path before attempting to start a new session. You can also connect to an existing session in the simple asynchronous mode by clicking **Connect**. Click **Batch Mode** to start a new Pro/ENGINEER session in the batch mode, or click **Full Async** to start a new Pro/ENGINEER session in the full asynchronous mode. Refer to the [`VB API](#)

[Fundamentals:Controlling Pro/ENGINEER](#)' chapter for more information on the modes of communication.

7. Once you are connected to a Pro/ENGINEER session, the **VB API Examples** form is loaded. You can execute all the examples available in the `vbapi_examples` directory from this form.

## Parameters and Dimensions

Location	Main Class
<code>vbapi/vbapi_appls/vbparam</code>	FormPD

The parameters and dimensions example is an asynchronous mode VB.NET application that allows you to access and modify the parameters and dimensions of a Pro/ENGINEER model. All the VB source files for this application are available along with a VB.NET solution called `ParameterAndDimension.sln` and a project file called `ParameterAndDimension.vbproj` as a single `VBParam.zip` file in the `vbparam` directory.

Set up and run this application using the following procedure:

1. Set the `PRO_COMM_MSG_EXE` environment variable to the full path of the executable `pro_comm_msg.exe` for your application to communicate with Pro/ENGINEER. Typically, the path to the executable is `[Pro/E loadpoint]/[machine type]/obj/pro_comm_msg.exe`, where `machine_type` is `i486_nt` for 32-bit Windows and `x86_win64` for 64-bit Windows installations.
2. Register the COM server by running the `vb_api_register.bat` file located at `[Pro/E loadpoint]/bin`.
3. Unzip the `VBParam.zip` file in a local folder on your machine and open the `ParameterAndDimension.sln` solution in Microsoft Visual Studio.
4. Set the COM reference for your project to **Pro/E VB API Type Library for Pro/E Wildfire 4.0**.

5. Build the solution and execute the `ParameterAndDimension.exe` created in your local folder. The **Parameters and Dimensions** form is loaded.
6. Start Pro/ENGINEER and open a PART model containing parameters and dimensions.
7. Click the **Connect** button in the form to connect to the active Pro/ENGINEER session in the simple asynchronous mode. Click the **Add** button to connect in the full asynchronous mode, wherein a new **PDMenu** menu gets added to the menubar in the Pro/ENGINEER user interface.

You can perform the same set of operations on parameters and dimensions from the **Parameters and Dimensions** form in the simple asynchronous mode and from the **PDMenu** menu in the full asynchronous mode.

8. Click **Disconnect** to disconnect from the current Pro/ENGINEER session.

You can perform the following operations on parameters from the **Parameters and Dimensions** form:

- Retrieve all the parameters of a PART model in the current Pro/ENGINEER session inside the parameter table in the Parameters and Dimensions form.
- Modify the unit, value, designated status, and description, except name and type for each parameter.
- Delete a parameter and all the values associated with it.
- Save the updated list of parameters back in the model.
- Save the list of parameters retrieved from the model in an XML file, or read the parameters from a previously saved XML file in the form.

You can perform the following operations on dimensions from the **Parameters and Dimensions** form:

- Retrieve all the dimensions of a PART model in the current Pro/ENGINEER session inside the dimensions table in the Parameters and Dimensions form.
- Modify the name, nominal value, tolerance type, tolerance value 1, and tolerance value 2, except ID and type for each dimension.
- Save the updated list of dimensions back in the model.
- Save the list of dimensions retrieved from the model in an XML file or read the dimensions from a previously saved XML file in the form.

The **Parameters and Dimensions** form containing the parameters retrieved from a PART model is shown in the following figure.

Parameters And Dimensions

Parameter Dimensions

Model

Retrieve Retrieve Parameter from current Model

Save Save Parameters to current Model

XML File

Path to XML File  Browse

Read Read Parameter from XML File

Save Save Parameters to XML File

Auto Delete False

	NAME	TYPE	UNIT	VALUE	DESIG STATUS	DESCRIPTION
▶	DESCRIPTION	String	No Units		True	
	MODELED_BY	String	No Units		True	
	PTC_COMMO...	String	No Units	p_mdldchk_tk_1...	False	
	VERSION	Integer	No Units	1	False	
	P_FC2_INFO	Integer	No Units	0	False	
	P_FC2_PARAM	Integer	No Units	0	False	
	P_FC2_LAYER	Integer	No Units	0	False	
	P_FC2_FEAT	Integer	No Units	0	False	
	P_FC2_RELAT	Integer	No Units	0	False	
	P_FC2_DATUM	Real	No Units	0	False	

Validate

Delete Row

Clear

Status

Parameter information obtained from Model P\_MDLCHK\_TK\_1

Add Add Import / Export functionality to Pro/Engineer Menu

Connect

Disconnect

# Digital Rights Management

---

This section describes the implications of DRM on the VB API applications.

## Topic

[Introduction](#)

[Implications of DRM on the VB API](#)

[Additional DRM Implications](#)

## Introduction

Digital Rights Management (DRM) helps to control access to your intellectual property. Intellectual property could be sensitive design and engineering information that you have stored within Pro/ENGINEER parts, assemblies, or drawings. You can control access by applying policies to these Pro/ENGINEER objects. Such objects remain protected by the policies even after they are distributed or downloaded. Pro/ENGINEER objects for which you have applied policies are called DRM-protected objects. For more information on the use of DRM in Pro/ENGINEER Wildfire 4.0, refer to the DRM online help.

The following sections describe how the VB API applications deal with DRM-protected objects.

## Implications of DRM on the VB API

Any VB API application accessing DRM-protected objects can run only in interactive Pro/ENGINEER sessions having COPY permissions. As the VB API applications can extract content from models into an unprotected format, the VB API applications will not run in a Pro/ENGINEER session lacking COPY permission.

If the user tries to open a model lacking the COPY permission into a session with a VB API application running, Pro/ENGINEER prompts the user to spawn a new session. Also, new VB API applications will not be permitted to start when the Pro/ENGINEER session lacks COPY permission.

If a VB API application tries to open a model lacking COPY permission from an

interactive Pro/ENGINEER session, the application throws the **pfExceptions.XToolkitNoPermission** exception.

When a VB API application tries to open a protected model from a non-interactive or batch mode application, the session cannot prompt for DRM authentication, instead the application throws the **pfExceptions.XToolkitNoPermission** exception.

## Exception Types

Some VB API methods require specific permissions in order to operate on a DRM-protected object. If these methods cannot proceed due to DRM restrictions, the following exceptions are thrown:

- **pfExceptions.XToolkitNoPermission**--Thrown if the method cannot proceed due to lack of needed permissions.
- **pfExceptions.XToolkitAuthenticationFailure**--Thrown if the object cannot be opened because the policy server could not be contacted or if the user was unable to interactively login to the server.
- **pfExceptions.XToolkitUserAbort**--Thrown if the object cannot be operated upon because the user cancelled the action at some point.

The following table lists the methods along with the permission required and implications of operating on DRM-protected objects.

Methods	Permission Required	Implications
<p>IpfcBaseSession. RetrieveAssemSimpRep()</p> <p>IpfcBaseSession. CreateDrawingFromTemplate()</p> <p>IpfcBaseSession. RetrieveGraphicsSimpRep()</p> <p>IpfcBaseSession. RetrieveGeomSimpRep()</p> <p>IpfcBaseSession.RetrieveModel ( )</p>	<p>OPEN</p>	<ul style="list-style-type: none"> <li>● If file has OPEN and COPY permissions, model opens after authentication.</li> <li>● Throws the pfExceptions.XToolkitNoPermission</li> </ul>

<p>IpfcBaseSession. RetrieveModelWithOpts()</p> <p>IpfcBaseSession. RetrievePartSimpRep()</p> <p>IpfcBaseSession. RetrieveSymbolicSimpRep()</p>		<p>exception otherwise.</p>
<p>IpfcModel.Rename()</p>	<p>OPEN</p>	<ul style="list-style-type: none"> <li>• File is saved with the current policy to disk if it has COPY permission.</li> </ul>
<p>IpfcModel.Backup()</p> <p>IpfcModel.Copy()</p>	<p>SAVE</p>	<ul style="list-style-type: none"> <li>• File is saved with the current policy to disk if it has SAVE and COPY permissions.</li> <li>• Throws the pfcExceptions. XToolkitNoPermission exception if model has COPY permission, but lacks SAVE permission.</li> </ul>
<p>IpfcModel.Save()</p>	<p>SAVE</p>	<ul style="list-style-type: none"> <li>• File is saved with the current policy to disk if it has SAVE and COPY permissions.</li> <li>• Throws the pfcExceptions. XToolkitNoPermission exception if model has COPY permission, but lacks SAVE permission.</li> <li>• Throws the pfcExceptions. XToolkitNoPermission exception if the assembly file has models with COPY permission, but lacking SAVE permission.</li> </ul>

<p>IpfcModel.Export() for PlotInstructions</p> <p>IpfcModel.Export() for ProductViewExportInstructions (only if the input model is a drawing)</p> <p>IpfcBaseSession.ExportCurrentRasterImage()</p>	<p>PRINT</p>	<ul style="list-style-type: none"> <li>• Drawing file is printed if it has PRINT permission.</li> <li>• Throws the pfcExceptions.XToolkitNoPermission exception if drawing file lacks PRINT permission.</li> </ul>

## Copy Permission to Interactively Open Models

When the user tries to open protected content lacking COPY permission through the Pro/ENGINEER user interface with a VB API application running in the same session:

1. Pro/ENGINEER checks for the authentication credentials through the user interface, if they are not already established.
2. If the user has permission to open the file, Pro/ENGINEER checks if the permission level includes COPY. If the level includes COPY, Pro/ENGINEER opens the file.
3. If COPY permission is not included, the following message is displayed:



4. If the user clicks **Cancel**, the file is not opened in the current Pro/ENGINEER session and no new session is spawned.
5. If the user clicks **OK**, an additional session of Pro/ENGINEER is spawned

which does not permit any VB API application. VB API applications set to automatically start by Pro/ENGINEER will not be started. Asynchronous applications will be unable to connect to this session.

6. The new session of Pro/ENGINEER is automatically authenticated with the same session credentials as were used in the previous session.
7. The model that Pro/ENGINEER was trying to load in the previous session is loaded in this session.
8. Other models already open in the previous session will not be loaded in the new session.
9. Session settings from the previous session will not be carried into the new session.
10. The new session will be granted the licenses currently used by the previous session. This means that the next time the user tries to do something in the previous session, Pro/ENGINEER must obtain a new license from the license server. If the license is not available, the action will be blocked with an appropriate message.

## **Additional DRM Implications**

- The method `IpfcModel.CheckIsSaveAllowed()` returns false if prevented from save by DRM restrictions.
  - The method `IpfcBaseSession.CopyFileToWS()` is designed to fail and throw the `pfcExceptions.XToolkitNoPermission` exception if passed a DRM-protected Pro/ENGINEER model file.
  - The method `IpfcBaseSession.ImportToCurrentWS()` reports a conflict in its output text file and does not copy a DRM-protected Pro/ENGINEER model file to the Workspace.
  - While erasing an active Pro/ENGINEER model with DRM restrictions, the methods `IpfcModel.Erase()` and `IpfcModel.EraseWithDependencies()` do not clear the data in the memory until the control returns to Pro/ENGINEER from the Pro/TOOLKIT application. Thus, the Pro/ENGINEER session permissions may also not be cleared immediately after these methods return.
-

# Geometry Traversal

---

This section illustrates the relationships between faces, contours, and edges. Examples E-1 through E-5 show some sample parts and list the information about their surfaces, faces, contours, and edges.

## Topic

[Example 1](#)

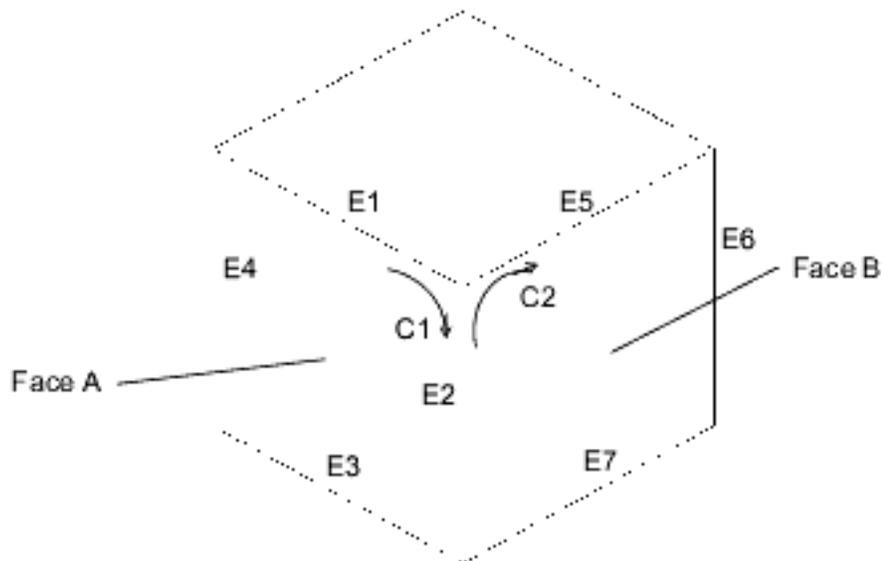
[Example 2](#)

[Example 3](#)

[Example 4](#)

[Example 5](#)

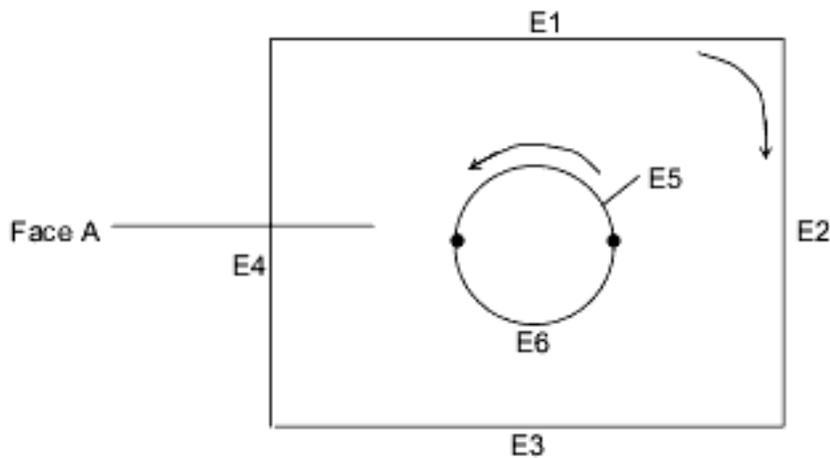
## Example 1



This part has 6 faces.

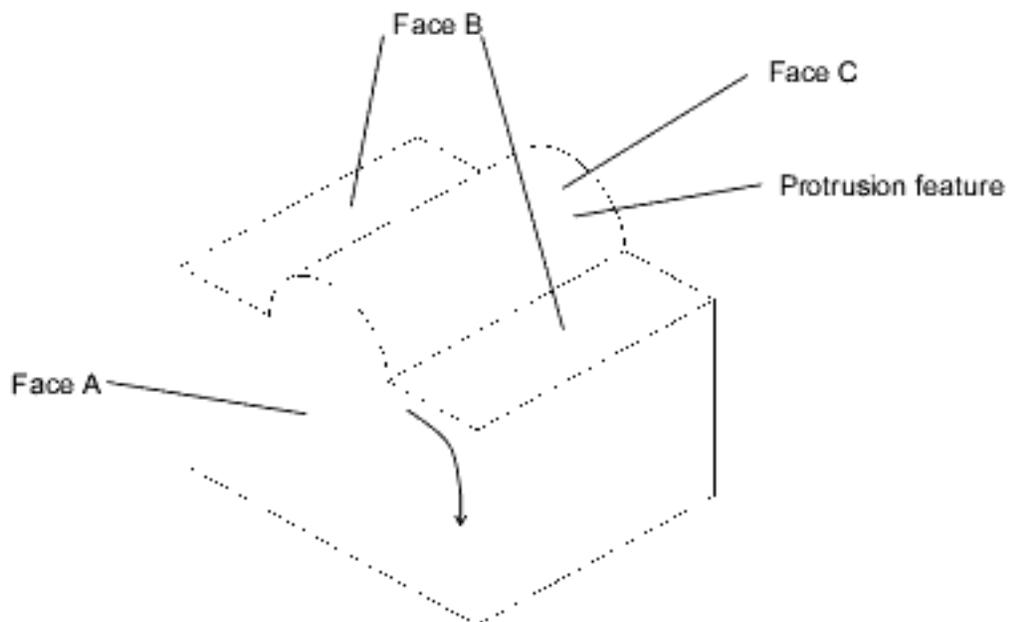
- Face A has 1 contour and 4 edges.
- Edge E2 is the intersection of faces A and B.
- Edge E2 is a component of contours C1 and C2.

## Example 2



Face A has 2 contours and 6 edges.

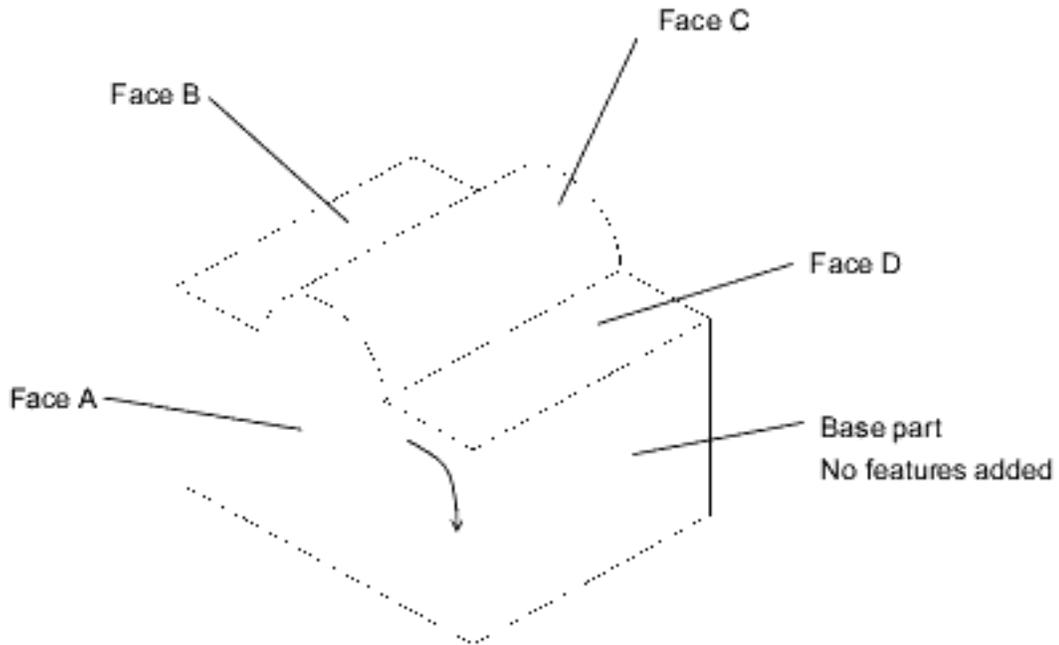
## Example 3



This part was extruded from a rectangular cross section. The feature on the top was added later as an extruded protrusion in the shape of a semicircle.

- Face A has 1 contour and 6 edges.
- Face B has 2 contours and 8 edges.
- Face C has 1 contour and 4 edges.

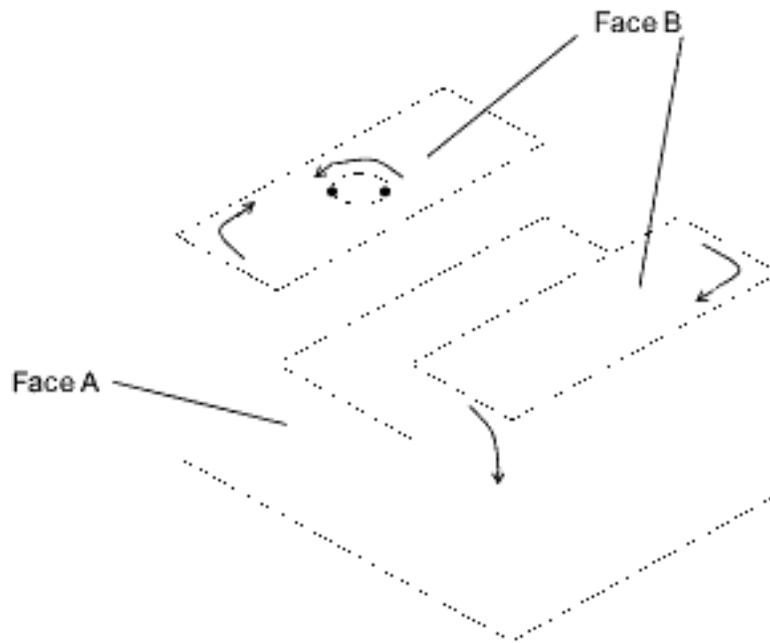
## Example 4



This part was extruded from a cross section identical to Face A. In the Sketcher, the top boundary was sketched with two lines and an arc. The sketch was then extruded to form the base part, as shown.

- Face A has 1 contour and 6 edges.
- Face B has 1 contour and 4 edges.
- Face C has 1 contour and 4 edges.
- Face D has 1 contour and 4 edges.

## Example 5



This part was extruded from a rectangular cross section. The slot and hole features were added later.

- Face A has 1 contour and 8 edges.
  - Face B has 3 contours and 10 edges.
-

# Geometry Representations

---

This section describes the geometry representations of the data used by the VB API.

## Topic

[Surface Parameterization](#)

[Edge and Curve Parameterization](#)

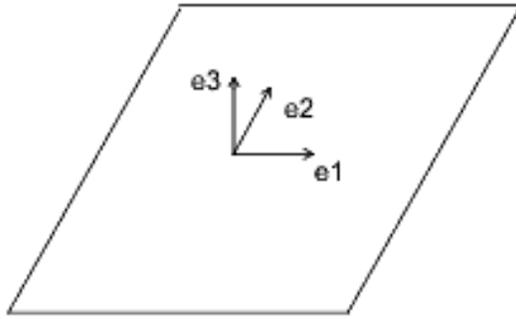
## Surface Parameterization

A surface in Pro/ENGINEER 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.

This section describes the surface parameterization. The surfaces 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

## Plane



The plane entity consists of two perpendicular unit vectors ( $e1$  and  $e2$ ), the normal to the plane ( $e3$ ), and the origin of the plane.

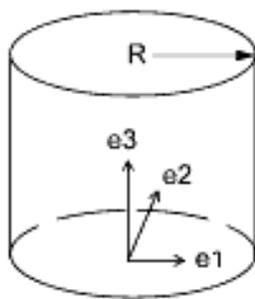
Data Format:

```
e1[3]      Unit vector, in the  $u$  direction
e2[3]      Unit vector, in the  $v$  direction
e3[3]      Normal to the plane
origin[3]  Origin of the plane
```

Parameterization:

$$(x, y, z) = u * e1 + v * e2 + origin$$

## Cylinder



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:

```
e1[3]      Unit vector, in the  $u$  direction
e2[3]      Unit vector, in the  $v$  direction
```

<code>e3[3]</code>	Normal to the plane
<code>origin[3]</code>	Origin of the cylinder
<code>radius</code>	Radius of the cylinder

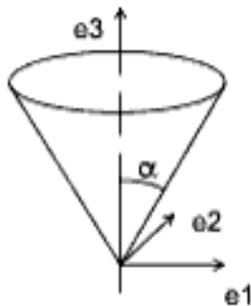
Parameterization:

$$(x, y, z) = \text{radius} * [\cos(u) * e1 + \sin(u) * e2] + v * e3 + \text{origin}$$

Engineering Notes:

For the cylinder, cone, torus, and general surface of revolution, a local coordinate system is used that consists of three orthogonal unit vectors ( $e1$ ,  $e2$ , and  $e3$ ) and an origin. The curve lies in the plane of  $e1$  and  $e3$ , and is rotated in the direction from  $e1$  to  $e2$ . The  $u$  surface parameter determines the angle of rotation, and the  $v$  parameter determines the position of the point on the generating curve.

## Cone



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  $v * \tan(\alpha)$ .

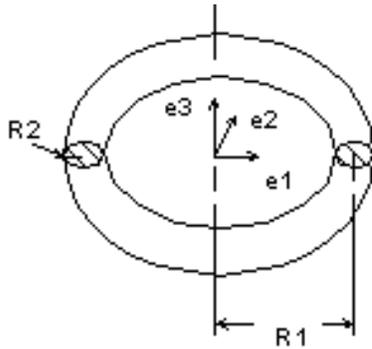
Data Format:

<code>e1[3]</code>	Unit vector, in the $u$ direction
<code>e2[3]</code>	Unit vector, in the $v$ direction
<code>e3[3]</code>	Normal to the plane
<code>origin[3]</code>	Origin of the cone
<code>alpha</code>	Angle between the axis of the cone and the generating line

Parameterization:

$$(x, y, z) = v * \tan(\alpha) * [\cos(u) * e1 + \sin(u) * e2] + v * e3 + \text{origin}$$

## Torus



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  $R_2 * \sin(v)$ .

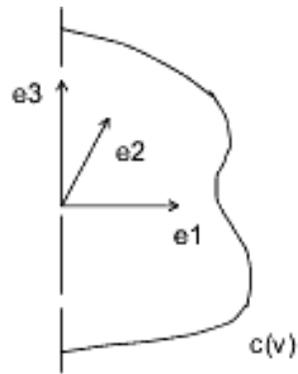
Data Format:

<code>e1[3]</code>	Unit vector, in the $u$ direction
<code>e2[3]</code>	Unit vector, in the $v$ direction
<code>e3[3]</code>	Normal to the plane
<code>origin[3]</code>	Origin of the torus
<code>radius1</code>	Distance from the center of the generating arc to the axis of revolution
<code>radius2</code>	Radius of the generating arc

Parameterization:

$$(x, y, z) = (R_1 + R_2 * \cos(v)) * [\cos(u) * e1 + \sin(u) * e2] + R_2 * \sin(v) * e3 + \text{origin}$$

## General Surface of Revolution



A general surface of revolution is created by rotating a curve entity, usually a spline, around an axis. The curve is evaluated at the normalized parameter  $v$ , and the resulting point is rotated around the axis through an angle  $u$ . The surface of revolution data structure consists of a local coordinate system and a curve structure.

Data Format:

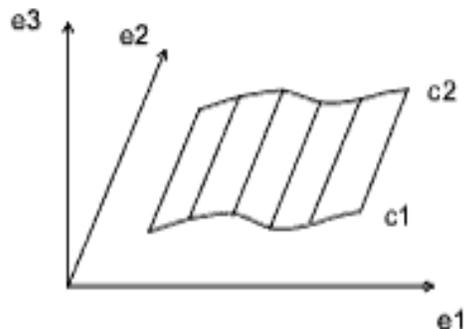
e1[3]	Unit vector, in the $u$ direction
e2[3]	Unit vector, in the $v$ direction
e3[3]	Normal to the plane
origin[3]	Origin of the surface of revolution
curve	Generating curve

Parameterization:

$\text{curve}(v) = (c1, c2, c3)$  is a point on the curve.

$$(x, y, z) = [c1 * \cos(u) - c2 * \sin(u)] * e1 + [c1 * \sin(u) + c2 * \cos(u)] * e2 + c3 * e3 + \text{origin}$$

## Ruled Surface



A ruled surface is the surface generated by interpolating linearly between corresponding points of two curve entities. The  $u$  coordinate is the normalized parameter at which both curves are evaluated, and the  $v$  coordinate is the linear parameter between the two points. The curves are not defined in the local coordinate system of the part, so the resulting point must be transformed by the local coordinate system of the surface.

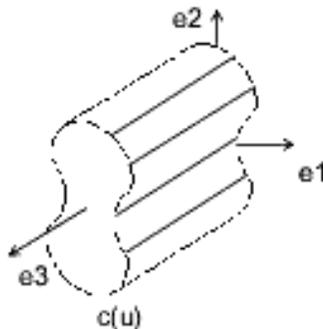
Data Format:

```
e1[3]      Unit vector, in the  $u$  direction
e2[3]      Unit vector, in the  $v$  direction
e3[3]      Normal to the plane
origin[3]  Origin of the ruled surface
curve_1    First generating curve
curve_2    Second generating curve
```

Parameterization:

```
( $x'$ ,  $y'$ ,  $z'$ ) is the point in local coordinates.
( $x'$ ,  $y'$ ,  $z'$ ) = (1 -  $v$ ) * C1( $u$ ) +  $v$  * C2( $u$ )
( $x$ ,  $y$ ,  $z$ ) =  $x'$  * e1 +  $y'$  * e2 +  $z'$  * e3 + origin
```

## 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:

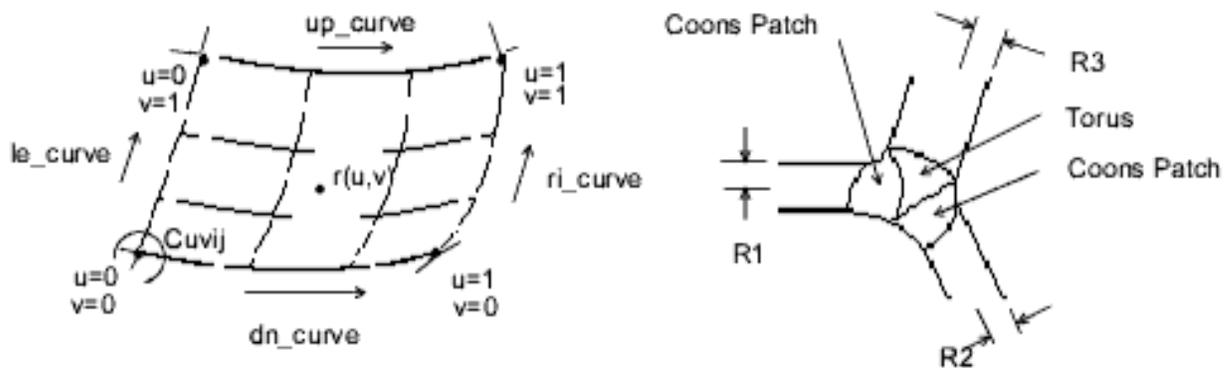
```
e1[3]      Unit vector, in the  $u$  direction
e2[3]      Unit vector, in the  $v$  direction
e3[3]      Normal to the plane
```

origin[3]    Origin of the tabulated cylinder  
curve        Generating curve

Parameterization:

$(x', y', z')$  is the point in local coordinates.  
 $(x', y', z') = C(u) + (0, 0, v)$   
 $(x, y, z) = x' * e1 + y' * e2 + z' * e3 + origin$

## Coons Patch

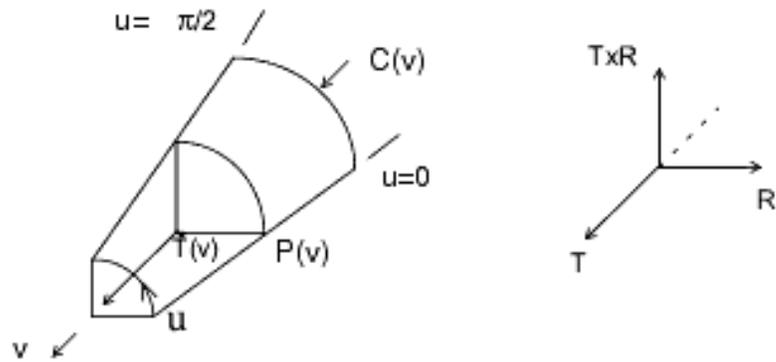


A Coons patch is used to blend surfaces together. For example, you would use a Coons patch at a corner where three fillets (each of a different radius) meet.

Data Format:

le_curve	$u = 0$ boundary
ri_curve	$u = 1$ boundary
dn_curve	$v = 0$ boundary
up_curve	$v = 1$ boundary
point_matrix[2][2]	Corner points
uvder_matrix[2][2]	Corner mixed derivatives

## Fillet Surface



A fillet surface is found where a round or a fillet is placed on a curved edge, or on an edge with non-constant arc radii. On a straight edge, a cylinder would be used to represent the fillet.

Data Format:

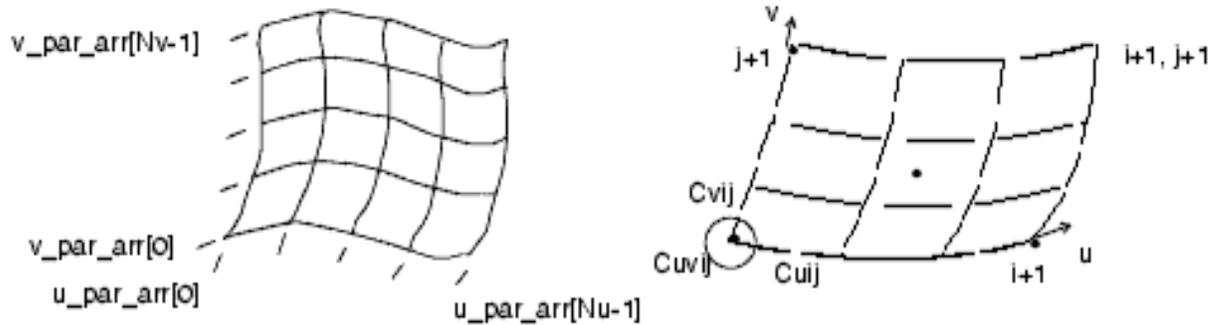
pnt\_spline  $P(v)$  spline running along the  $u = 0$  boundary  
 ctr\_spline  $C(v)$  spline along the centers of the fillet arcs  
 tan\_spline  $T(v)$  spline of unit tangents to the axis of the fillet arcs

Parameterization:

$$R(v) = P(v) - C(v)$$

$$(x, y, z) = C(v) + R(v) * \cos(u) + T(v) \times R(v) * \sin(u)$$

## Spline Surface



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:

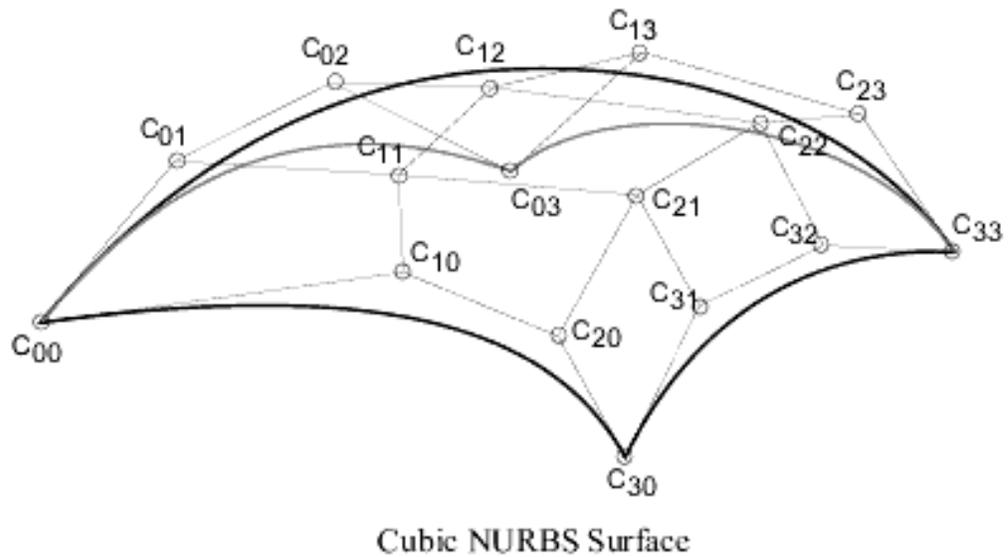
<code>u_par_arr[]</code>	Point parameters, in the $u$ direction, of size $Nu$
<code>v_par_arr[]</code>	Point parameters, in the $v$ direction, of size $Nv$
<code>point_arr[][3]</code>	Array of interpolant points, of size $Nu \times Nv$
<code>u_tan_arr[][3]</code>	Array of $u$ tangent vectors at interpolant points, of size $Nu \times Nv$
<code>v_tan_arr[][3]</code>	Array of $v$ tangent vectors at interpolant points, of size $Nu \times Nv$
<code>uvder_arr[][3]</code>	Array of mixed derivatives at interpolant points, of size $Nu \times Nv$

Engineering Notes:

- Allows for a unique  $3 \times 3$  polynomial around every patch.
- There is second order continuity across patch boundaries.
- The point and tangent vectors represent the ordering of an array of  $[i][j]$ , where  $u$  varies with  $i$ , and  $v$  varies with  $j$ . In walking through the `point_arr[][3]`, you will find that the innermost variable representing  $v(j)$  varies first.

## NURBS Surface

The NURBS surface is defined by basis functions (in  $u$  and  $v$ ), expandable arrays of knots, weights, and control points.



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  $u$

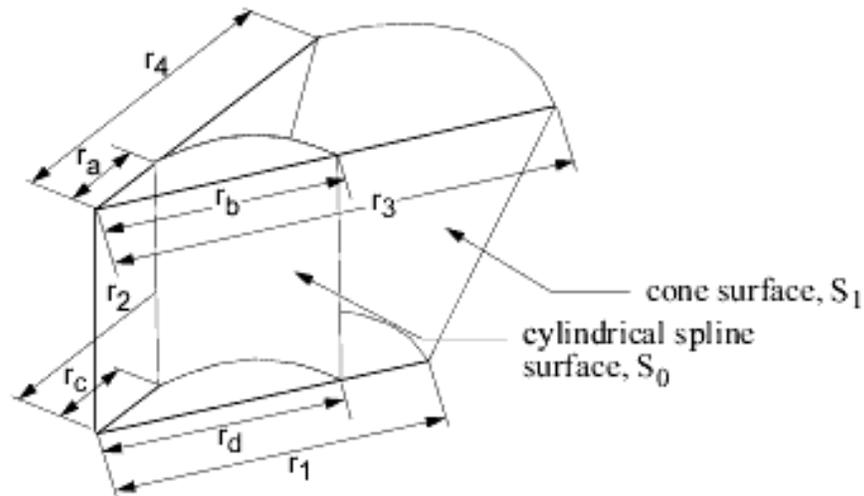
$l$  = degree in  $v$   
 $N1$  = (number of knots in  $u$ ) - (degree in  $u$ ) - 2  
 $N2$  = (number of knots in  $v$ ) - (degree in  $v$ ) - 2  
 $B_{i,k}$  = basis function in  $u$   
 $B_{j,l}$  = basis function in  $v$   
 $w_{ij}$  = weights  
 $C_{i,j}$  = control points  $(x,y,z) * w_{i,j}$

Engineering Notes:

The *weights* and *c\_points\_arr* arrays represent matrices of size  $wghts[N1+1][N2+1]$  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.



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
splsurf	Spline surface data structure

The spline surface data structure contains the following fields:

u_par_arr[]	Point parameters, in the $u$ direction, of size $N_u$
v_par_arr[]	Point parameters, in the $v$ direction, of size $N_v$
point_arr[][3]	Array of points, in cylindrical coordinates, of size $N_u \times N_v$ . 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 $N_u \times N_v$
v_tan_arr[][3]	Array of $v$ tangent vectors, in cylindrical coordinates, of size $N_u \times N_v$
uvder_arr[][3]	Array of mixed derivatives, in cylindrical coordinates, of size $N_u \times N_v$

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:

$$\begin{aligned} x' &= r \cos(\theta) \\ y' &= r \sin(\theta) \\ z' &= z \end{aligned}$$

A cylindrical spline surface can be obtained, for example, by creating a smooth rotational blend (shown in the figure).

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

$(r1 = r2, r3 = r4, \text{ and } r1 \neq r3)$ .

If a replacement cannot be done (such as for the surface  $S0$  in the figure ( $ra \neq rb$  or  $rc \neq rd$ )), leave it as a cylindrical spline surface representation.

## Edge and Curve Parameterization

This parameterization represents edges (line, arc, and spline) as well as the curves (line, arc, spline, and NURBS) within the surfaces.

This section describes edges and curves, arranged in order of complexity. For ease of use, the alphabetical listing is as follows:

- Arc
- Line
- NURBS
- Spline

### Line

Data Format:

```
end1[3]   Starting point of the line
end2[3]   Ending point of the line
```

Parameterization:

$$(x, y, z) = (1 - t) * \text{end1} + t * \text{end2}$$

### Arc

The arc entity is defined by a plane in which the arc lies. The arc is centered at the origin, and is parameterized by the angle of rotation from the first plane unit vector in the direction of the second plane vector. The start and end angle parameters of the arc and the radius are also given. The direction of the arc is counterclockwise if the start angle is less than the end angle, otherwise it is clockwise.

Data Format:

```
vector1[3]   First vector that defines the
              plane of the arc
```

vector2[3]	Second vector that defines the plane of the arc
origin[3]	Origin that defines the plane of the arc
start_angle	Angular parameter of the starting point
end_angle	Angular parameter of the ending point
radius	Radius of the arc.

Parameterization:

$t'$  (the unnormalized parameter) is  
 $(1 - t) * \text{start\_angle} + t * \text{end\_angle}$   
 $(x, y, z) = \text{radius} * [\cos(t') * \text{vector1} + \sin(t') * \text{vector2}] + \text{origin}$

## 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  $p\_max$  be the parameter of the last spline point. Then,  $t'$ , the unnormalized parameter, is  $t * p\_max$ .

Locate the  $i$ th spline segment such that:

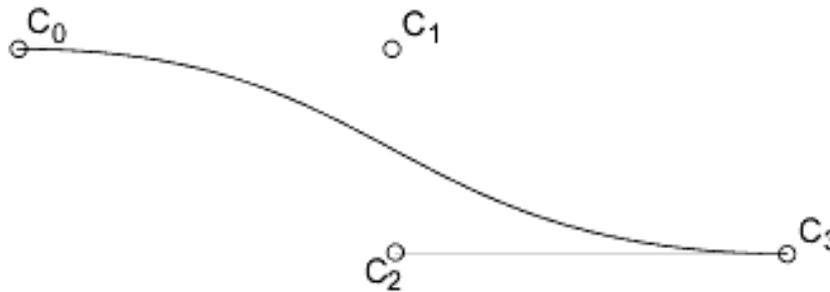
$$\text{par\_arr}[i] < t' < \text{par\_arr}[i+1]$$

(If  $t < 0$  or  $t > +1$ , use the first or last segment.)

$$t_0 = (t' - \text{par\_arr}[i]) / (\text{par\_arr}[i+1] - \text{par\_arr}[i])$$
$$t_1 = (\text{par\_arr}[i+1] - t') / (\text{par\_arr}[i+1] - \text{par\_arr}[i])$$

## NURBS

The NURBS (nonuniform rational B-spline) curve is defined by expandable arrays of knots, weights, and control points.



Cubic NURBS Curve

Data Format:

degree	Degree of the basis function
params[]	Array of knots
weights[]	Array of weights for rational NURBS, otherwise NULL.
c_pnts[][][3]	Array of control points

Definition:

$$R(t) = \frac{\sum_{i=0}^N C_i \times B_{i,k}(t)}{\sum_{i=0}^N w_i \times B_{i,k}(t)}$$

$k$  = degree of basis function

$N = (\text{number of knots}) - (\text{degree}) - 2$

$w_i = \text{weights}$

$C_i = \text{control points } (x, y, z) * w_i$

$B_{i,k} = \text{basis functions}$

By this equation, the number of control points equals  $N+1$ .

References:

Faux, I.D., M.J. Pratt. *Computational Geometry for Design and Manufacture*. Ellis Harwood Publishers, 1983.

Mortenson, M.E. *Geometric Modeling*. John Wiley & Sons, 1985.

---