Parasolid V13.0

PS/Workshop V2.1 Developer Guide

June 2001

Important Note

Parker's House
46 Regent Street
Cambridge CB2 1DP
UK
Tel: +44 (0)1223 371555
Fax: +44 (0)1223 316931
email: ps-support@ugs.com
Web: www.parasolid.com

Trademarks

Parasolid is a trademark of Unigraphics Solutions Inc.

HP and HP-UX are registered trademarks of Hewlett-Packard Co.

SPARCstation and Solaris are trademarks of Sun Microsystems, Inc.

Alpha AXP and VMS are trademarks of Digital Equipment Corp.

IBM, RISC System/6000 and AIX are trademarks of International Business Machines Corp.

OSF is a registered trademark of Open Software Foundation, Inc.

UNIX is a registered trademark of UNIX Systems Laboratories, Inc.

Microsoft Visual C/C++ and Window NT are registered trademarks of Microsoft Corp.

Intel is a registered trademark of Intel Corp.

Silicon Graphics is a registered trademark, and IRIX a trademark, of Silicon Graphics, Inc.

All other trademarks are the property of their respective owners.

Table of Contents

Introduction 1

1.1 Introduction

This manual provides a complete guide for developers who want to write their own modules for use with PS/Workshop. It contains the following chapters:

- Chapter 1 (this chapter) introduces you to the manual, tells you how to install the PS/Workshop SDK, and introduces you to the PS/Workshop AppWizard that is installed with the SDK.
- Chapter 2, “A Short Tutorial”, provides a step-by-step example that you can work through to gain a basic understanding of the processes involved in writing your own modules. It explains how to create a simple module using the PS/Workshop AppWizard.
- Chapter 3, “Using COM in PS/Workshop” provides a brief introduction to COM, and the way that it is used in PS/Workshop.
- Chapter 4, “Event Handling Within A Module” explains how events and messages are passed from PS/Workshop to a module, and how they should subsequently be handled by the module.
- Chapter 5, “Module Structure” describes the structure of the default classes and functions available in a module created using the PS/Workshop AppWizard.
- Chapter 6, “The PS/Workshop Interfaces” is an introduction to the COM interfaces in PS/Workshop that can be used by your module to access PS/Workshop functionality.
- Chapter 7, “Adding a New Menu Item to PS/Workshop” explains how you can add new menus and menu commands to the PS/Workshop menu bar, to give users access to the functionality in your module.
- Chapter 9, “The Draw List” explains how you can control which parts of a document are displayed, and what options are used to display them.
- Chapter 8, “Registering Handlers for Different Filetypes” describes how to modify PS/Workshop to load and save files in formats other than the default file formats supported by the core version of PS/Workshop itself.
- Appendix A, “Interface Functions” is a complete reference for the COM interfaces provided by PS/Workshop that you can use in a PS/Workshop module.
- Appendix B, “Known Issues” describes miscellaneous issues relevant to developing modules for use in PS/Workshop.

This manual assumes that you are familiar with the functionality in the core version of PS/Workshop, as well as the mechanism for loading and unloading PS/Workshop modules. For more details, see the *PS/Workshop V2 User Guide*.

1.2 Installation

In order to develop PS/Workshop modules, you need to install the PS/Workshop SDK (Software Development Kit). This needs to be installed when you install PS/Workshop itself, by choosing the appropriate option in the installation wizard. The SDK is installed by default for the “Typical” configuration, so if you installed PS/Workshop with this configuration, you already have the SDK.

If you do not have the SDK installed, run the installation again and ensure that the option is checked.

1.3 PS/Workshop SDK directory structure

The PS/Workshop SDK consists of the following files and folders:

Folder	Files
<*INSTALL_DIR*>	PSWorkshopAddonWizard.awx: The Visual Studio PS/Workshop project wizard. See Section 1.4 for more details. PSWorkshop.tlb: Type library definitions for PS/Workshop.
<*INSTALL_DIR*>\Help	The help files for PS/Workshop: ■ PSWorkshopUserGuide.pdf The PS/Workshop User Guide in PDF format. ■ PSWorkshopDevelopmentGuide.pdf This document in PDF format.
<*INSTALL_DIR*>\Modules	Source code and object releases for the sample PS/Workshop modules ■ Analyse.dll A module that performs a number of analytical operations on the parts in a document ■ Edge.dll A module that demonstrates the edge blending functionality of Parasolid
<*INSTALL_DIR*>\Modules\Source	Contains the source code for a number of example modules.
<*INSTALL_DIR*>\Modules\Source\Edge Blend	The source code for the edge blending module.
<*INSTALL_DIR*>\Modules\Tutorial	The full source code for the tutorial module described in Chapter 2, “A Short Tutorial”.

The example modules provided were written and compiled using Microsoft Visual Studio C++ 6.0 with Service Pack 4.0 installed.

1.4 The PS/Workshop module wizard

When you install the SDK, a module wizard is copied to the Visual Studio directory. This wizard produces a module that contains all the code for successfully initialising the module as well as adding a new menu and submenu to PS/Workshop. It also sets up and registers a stub function that is called from the menu.

The module wizard is shown as “PSWorkshop AppWizard” in the Projects tab of the New dialog in Visual Studio.

If you do not see this wizard, copy `<INSTALL_DIR>\PSWorkshopAddonWizard.awx` to `\Common\MSDev98\Template` under the Visual Studio directory.

Note: It is strongly advised that you use the PS/Workshop AppWizard to create your own modules. The wizard provides your module with basic functionality that will save you development time. In addition, some parts of this manual assume that your module has the same overall structure and functionality as that provided by the wizard.

A Short Tutorial 2

2.1 Introduction

This chapter contains a tutorial that guides you through the various stages required to implement a simple module in PS/Workshop. If you follow each step in the tutorial, you will create a module that hollows a given body in PS/Workshop.

In this tutorial you will learn how to:

- Create a new base project (Section 2.2)
- Add a new menu and menu item to PS/Workshop (Section 2.3)
- Add a handler for the new menu item (Section 2.4)
- Write code to hollow a body (Section 2.5)

> **Note:** If you have installed the PS/Workshop SDK, the source code for each step of this tutorial is installed in the folder `<INSTALL_DIR>\Modules\Tutorial`

Creating and displaying a dialog box is beyond the scope of this project. If you are interested in doing this, look at the example edge blending module provided. Alternatively, the MSDN documentation contains a number of tutorials that cover producing and displaying dialogs.

2.2 Creating a new project

Create a new template project using Visual Studio as follows:

- Choose **File > New** and click on the Projects tab
- Select the PS/Workshop AppWizard icon
- Type `Tutorial` in the Project name field
- Click **OK**

Compile the project and make sure that the module loads into PS/Workshop correctly:

- Start PS/Workshop and open a document
- Check that a new **Tutorial Debug** menu, and an associated menu command, appear on the PS/Workshop menu.

The Tutorial module is also listed in the Available Add-Ins list of the Add-Ins tab of the PS/Workshop Options dialog. You must close any open documents in PS/Workshop before opening the Options dialog in order to see this tab.

Note: On compilation you may receive the warning "/DELAYLOAD:pskernel.dll ignored; no imports found from pskernel.dll". This warning can be ignored as the module currently doesn't access any Parasolid functionality. See Chapter B, "Known Issues" for more information.

2.3 Adding a Hollow menu item

In this section you change the name of the default menu added by the template, and add a new **Hollow** command to it.

Using the ClassView in Visual Studio, locate the StartModule function within the CAddonMain class. This function is called after a module is successfully loaded and is responsible for initialising the module and adding any required menus or menu items to PS/Workshop.

Within StartModule a call is made to the PS/Workshop interface function AddMenuItem which adds a menu item to PS/Workshop. AddMenuItem contains the following arguments:

```
IPSWAddIn        *pAddIn,
BSTR         CommandName,
long         CommandID,
PSW_Menu_Mode      mode
```

The `CommandName` argument controls which menu to add the item to; as well as what text should appear on any sub-item. In order to create a new **Operations** menu with a **Hollow** command you must modify the following line within StartModule:

```
CComBSTR bstrMenuItem
   (OLESTR("&Tutorial Debug\nMy NewCommand") );
```

should become

```
CComBSTR bstrMenuItem
   (OLESTR("&Operations\n&Hollow") );
```

Note: It is a good idea to use a CComBSTR smart pointer to encapsulate any BSTR, as this automatically frees the resources of the BSTR when it goes out of scope. For further details see Section 3.5.2, “Creating a BSTR”.

The `CommandID` argument specifies the ID that is passed back to the module if the menu item has been chosen. This ID should be unique within a module. You need to add a new ID to associate with the new **Hollow** command.

In Visual Studio, define an ID called `ID_ON_HOLLOW` as follows:

- Choose **View > Resource Symbols**
- Click **New** in the Resource Symbols dialog
- Type `ID_ON_HOLLOW` in the Name field of the New Symbol dialog

The `mode` argument controls the type of menu to add to PS/Workshop. This can have one of the following values:

Value	Description
PSW_Menu_App	Display the menu item at the application level (i.e. when there are no documents open in PS/Workshop)
PSW_Menu_Doc	Display the menu item at the document level (i.e. only when there are one or more documents open in PS/Workshop)

For the purposes of this example, leave the default value of `mode`: PSW_Menu_doc.

Change the call to AddMenuItem from:

```
return m_pAppInterface->AddMenuItem( pApp, bstrMenuItem,
ID_ON_MY_COMMAND, PSW_Menu_Doc);
```

to:

```
return m_pAppInterface->AddMenuItem( pApp, bstrMenuItem,
ID_ON_HOLLOW, PSW_Menu_Doc);
```

Finally, compile and run the code, and open a document in PS/Workshop. You should see a new **Operations** menu on the PS/Workshop menu that contains a **Hollow** command. If you choose **Operations > Hollow**, nothing happens yet, since you have not defined a function to handle this event.

The full source code for CAddonMain::StartModule should be as follows:

```
HRESULT CAddonMain::StartModule( IPSWAddIn* pApp )
{

ATLASSERT( pApp );

HRESULT hr = E_FAIL;

CComBSTR bstrMenuItem(OLESTR("&Operations\n&Hollow") );

return m_pAppInterface->AddMenuItem( pApp, bstrMenuItem,
ID_ON_HOLLOW, PSW_Menu_Doc);

}
```

For more information about adding menu items to PS/Workshop, see Chapter 7, “Adding a New Menu Item to PS/Workshop”.

2.4 Adding a handler for the OnCommand event

So far you have created an **Operations** menu containing a **Hollow** command. If you choose **Operations > Hollow**, this calls the IPSWEvents::OnCommand function on the module with the ID which you passed to the IPSWApp::AddMenuItem function (ID_ON_HOLLOW).

By default, a PS/Workshop module routes this message from the CAddinImpl class to the CAddonMain::OnCmdMsg function, which can either handle the event itself or call CAddonDoc::OnCmdMsg. This function in turn can either handle the event or pass it to CAddonView::OnCmdMsg to handle. The class you choose to handle the event depends on the functionality you want for the command.

In this case, to make the **Hollow** command functional, you must add a handler at the document level – i.e. the CAddonDoc class – since the command performs a hollow operation that acts on the parts in a document.

For a complete description of event handling, see Chapter 4, “Event Handling Within A Module”. You may find it useful to set a number of break points in the code and debug the module to better understand the event handling process.

Add the following private function to the CAddonDoc class using the Visual Studio ClassWizard (**View > ClassWizard**).

```
HRESULT CAddonDoc::OnHollow()
{
  AfxMessageBox("CAddonDoc::OnHollow reached");
  return S_OK;
}
```

For the time being, CAddonDoc::OnHollow simply displays a message when it is called. Section 2.5 describes how to add code that performs the hollow operation itself.

The ID of the new command is passed to CAddonDoc::OnCmdMsg as an argument. The OnCmdMsg function uses a switch statement to check whether it should handle the command and which function to route it to. In order to correctly handle the hollow command you need to modify this switch statement such that when it receives the ID_ON_HOLLOW ID it will call the CAddonDoc::OnHollow handler function.

To do this, add the following case into the switch statement:

```
  switch( nID )
  {
  case ID_ON_MY_COMMAND:
    OnMyFunction();
    hr = S_OK;
    break;
  case ID_ON_HOLLOW:        // Our newly added case statement
    OnHollow();
    hr = S_OK;
    break;
  default:
    break;
  }
```

Compile and run the code again, and open a document in PS/Workshop . Choose **Operations > Hollow** to display the message box and confirm that CAddonDoc::OnHollow has been successfully called.

2.5 Writing code to perform the hollow

You now have the framework of the hollow operation in place, and you can add code to perform the hollow itself. To do this you need to complete the following steps:

- Obtain any selected faces from the body (these are used as pierce faces for the hollow)
- Create a partition mark that you can rollback to if the hollow fails
- Call PK_BODY_hollow_2, using the selected faces as pierce faces
- Check for any errors and roll back to before the hollow if necessary
- Force an update of PS/Workshop if necessary

To perform the hollow operation, add the following code in place of the body of the CAddonDoc::OnHollow function.

```
// set our return argument
HRESULT hr = E_FAIL;

// Changes the cursor to an hourglass for the duration of the
// function to show the user that something is happening
CWaitCursor cursor;

// The member variables m_nParts and m_pkParts contain copies of
// the parts in the associated PS/Workshop document. These
// variables are automatically initialised and should be kept up
// to date with any changes in the number of parts in the
// document

// In this case if we have no parts in the document then there
// is no point continuing

if ( m_nParts == 0 )
{
  AfxMessageBox("CAddonDoc::OnHollow->No parts to hollow!");
  return S_OK;
}
// Clear the last error. This is for our rather simplistic error
// handling routine which is used later on
PK_LOGICAL_t  pk_was_error = PK_LOGICAL_true;
PK_ERROR_sf_t pk_error_sf;
PK_ERROR_clear_last( &pk_was_error );
// Obtain any selected faces in the document - these will be
// pierced during the hollow operation.
int     nFaces = 0;
PK_FACE_t*     faces = NULL;

// Get the number of selected faces
hr = m_pSelectionList->get_Count( PK_CLASS_face, &nFaces );
if ( FAILED( hr ) )
return hr;

// now get the actual selected faces
if ( nFaces )
{
  // allocate our array to store the selected faces in
```

```
  faces = new PK_FACE_t[ nFaces ];
  if ( faces == NULL )      // check if the allocation succeeded
    return E_OUTOFMEMORY;
  // there are two methods to obtain the faces required:
  // Call the IPSWSelectionList->get_Item function for each face
  // this has a performance disadvantage in that it makes n
  // separate calls to PS/Workshop to get all the faces

  for ( int i = 0; i < nFaces; i++ )
  {
     hr =
           m_pSelectionList->get_Item( PK_CLASS_face, i, &faces[
i ] );
    if ( FAILED( hr ) )
    {
      // delete our array
      delete [] faces;
      return hr;
    }
  }

  // Alternatively, get all the faces in one call to PS/Workshop.
  // For this we need to obtain the enumerator for the list
  // (IPSWEnumSelectionList).
  /*
CComPtr<IPSWEnumSelectionList> pEnumSel = NULL;
hr = m_pSelectionList->get__NewEnum( PK_CLASS_face,
(IUnknown**)&pEnumSel );
  if ( FAILED( hr ) )
  {
    // free our array and return
    delete [] faces;
    return hr;
  }
  // we can now get all the faces at one time
  hr = pEnumSel->Next( nFaces, faces, NULL );
  if ( FAILED( hr ) )
  {
    delete [] faces;
    return hr;
  }
  */
}
// create a mark to rollback to in case the hollow fails
PK_PMARK_t pmark = PK_ENTITY_null;
m_pRollback->MakePMark( &pmark );

// set the options for the hollow (here we are simply using the
// defaults).
PK_BODY_hollow_o_t hollowOptions;
PK_BODY_hollow_o_m( hollowOptions );

// fill in any faces to be pierced
if ( nFaces > 0 )
{
  hollowOptions.n_pierce_faces = nFaces;
  hollowOptions.pierce_faces = faces;
}
// and the return arguments
PK_TOPOL_track_r_t tracking;
```

```
PK_TOPOL_local_r_t results;

// finally call the function - here we are assuming that we only
// have one body in the document

PK_BODY_hollow_2( m_pkParts[0], 0.001, 1.0e-6, &hollowOptions,
&tracking, &results );

// Now we check to see if the previous function succeeded or not
PK_ERROR_ask_last( &pk_was_error, &pk_error_sf );
if( PK_LOGICAL_true == pk_was_error )
{
    CString pk_err_str = pk_error_sf.function;
    pk_err_str = pk_err_str + "\n returns \n";
    pk_err_str = pk_err_str + pk_error_sf.code_token;
    AfxMessageBox( pk_err_str, MB_OK | MB_ICONSTOP );
    if ( pk_error_sf.severity != PK_ERROR_mild )
    {
      // Then the model may be corrupted and we need to rollback
      m_pRollback->RollbackTo( &pmark, 1 );
      hr = E_FAIL;
    }
  }
// We also need to check the result structure to see if the
// function succeeded
  else if ( results.status != PK_local_status_ok_c )
  {
// Then again we need to rollback because it has failed
    AfxMessageBox("Hollow Failed");
    m_pRollback->RollbackTo( &pmark, 1 );
    hr = E_FAIL;
  }
  else
  {
// The hollow succeeded so force an update of PS/Workshop
    m_pDocInterface->Update( TRUE );
    // Delete the created pmark
    m_pRollback->DeletePMark( pmark );
    hr = S_OK;
  }

// Now we can free up some memory

  // our faces array
if ( nFaces > 0 )
  {
    nFaces = 0;
    delete [] faces;
    faces = NULL;
  }

  // Our return arguments from the hollow
  PK_TOPOL_track_r_f( &tracking );
  PK_TOPOL_local_r_f( &results );

// And finally return the result of the hollow operation
return hr;
```

Using COM in PS/Workshop

3

3.1 What is a COM object?

COM (Component Object Model) is a binary method for defining objects whose functionality can be used by an application regardless of the source code language in which either the object or the application are written. It allows code to be shared at a binary level rather than a source code level. A COM object is a binary object (usually implemented as a DLL) that exposes a set of methods that an application such as a PS/Workshop module can call.

Applications interact with COM objects in a similar way to C++ objects, although there are some clear differences:

- COM objects enforce strict encapsulation. The public methods in a COM object are grouped into one or more interfaces. To use a method, you must first create the COM object and then obtain the interface that contains the method from that object.
- COM objects must be created using COM-specific techniques, as described in Section 3.2, “Creating a COM object”.
- The lifetime of COM objects must be controlled using COM-specific techniques, as described in Section 3.4, “Managing the lifetime of COM objects”.
- Each COM object has a unique registered identifier that is used to create the object. COM automatically loads the correct DLL. You do not need to explicitly load the DLL or link to a static library in order to use a COM object.

PS/Workshop exposes a number of interfaces to provide access to its core functionality. The use of COM means that you do not have to develop your module code in the same language that PS/Workshop was itself developed in, though examples throughout this manual are given in C++.

A complete reference to the interfaces provided by PS/Workshop can be found in Appendix A, “Interface Functions”.

Further details about some of the concepts and use of COM can be found from the following sources:

- Inside COM by Dale Rogerson (Microsoft Press; 1997; ISBN: 1572313498)
- Inside Distributed COM by H. Eddon and G. Eddon (Microsoft Press; 1998; ISBN: 157231849X)

3.2 Creating a COM object

In C++, objects can either be

- created on the heap using the `new` operator, in which case their lifetime is controlled when the requisite `delete` operator is called, or
- defined on the stack, in which case their lifetime is controlled by the scope of the object.

By contrast, COM objects are created either

- indirectly, using a creation method exposed by a particular interface, or
- directly, using the function CoCreateInstance.

The first of these is the simplest approach, and is the one that you should use for the vast majority of interfaces exposed for PS/Workshop. The exception to this is the IPSWDrawOpts interface, which can be created directly in order to specify a set of drawing options to pass to the IPSWDrawList interface.

3.2.1 Creating a COM object indirectly

Creating a COM object using a public object creation method is straightforward. You pass the method the address of an interface pointer, and the method then creates the object and returns an interface pointer. When you use this approach, the type of interface that is returned is defined by the method, though you can often specify a number of things about how the object should be created.

The following example shows how to create a COM object indirectly:

```
IPSWApp*  pApp;         // address of interface pointer

IPSWDoc*  pDoc = NULL;  // returned interface pointer

HRESULT hr = pApp->OpenNewDocument(&pDoc);
                                // call to object creation method
```

The pointer to the new interface is contained in `pDoc`. You can use that pointer to access any of the interface's methods, as described in Section 3.3.1. The result of the call to the method is contained in `hr`, which can be tested for success or failure, as described in Section 3.3.2.

3.2.2 Creating a COM object directly

To create a COM object directly, you must do the following:

- Initialize COM using the function CoInitialize
- Create the object using the function CoCreateInstance
- Uninitialize COM using the function CoUninitialize

If you create a new module using the PS/Workshop AppWizard, then initializing and uninitializing COM is handled by the supplied framework.

In addition, you need to know the Class ID (CLSID) of the object you want to create. If this CLSID is not publicly available, you cannot create the object directly.

The following example shows how you can create an IPSWDrawOpts object, using ATL smart interface pointers, as described in Section 3.4.3.:

```
CComPtr<IPSWDrawOpts> pDrawOpts = NULL;
if ( FAILED( pDrawOpts.CoCreateInstance( CLSID_PSWDrawOpts ) )
)
{
  // recovery code
}
```

This creates a single uninitialized object of the class associated with the specified CLSID.

3.3 Using COM interfaces

To simplify the use of COM interfaces in PS/Workshop, as much of the COM complexity as possible is hidden behind the scenes. If you use the PS/Workshop AppWizard to create a new module, supporting code that you do not need to use explicitly is placed in areas of the source code that you do not need to alter. If you do not use the PS/Workshop AppWizard, then you can find the definitions of the PS/Workshop interfaces using the type library supplied in the PS/Workshop installation directory.

3.3.1 Calling public methods

Unlike C++, you do not access a COM object's methods directly. Instead, you must obtain a pointer to the interface that exposes the method. To call the method, you use essentially the same syntax that you would to invoke a pointer to a C++ method. For example, to invoke the IPSWDrawOpts::Reset method, you would use the following syntax.

```
IPSWDrawOpts *pDrawOpts;
...
pDrawOpts->Reset(...);
```

3.3.2 Testing for success or failure

All public methods exposed by an interface return a 32-bit integer called an HRESULT. For the interfaces exposed by PS/Workshop, this is used to indicate the return status of the method. Success codes are given names with an S_ prefix (such as S_OK), and failure codes are given names with an E_ prefix (such as E_FAIL). Specific error codes are documented with the reference documentation for each method. General error codes are taken from the standard set defined in `Winerror.h`.

The fact that methods may return a variety of success or failure codes means that you need to be careful when testing whether a call to a given method has been successful or not. If you need detailed information about the outcome of a call to a method, you need to test against individual return values. However, if you want to implement a robust method for detecting the general success or failure of a method call, you should use the following two macros, which are defined in `Winerror.h`:

- The SUCCEEDED macro returns TRUE if the method call was successful, and FALSE otherwise.
- The FAILED macro returns TRUE if the method call failed, and TRUE otherwise.

These macros give you a simple way of testing for general success or failure, as shown in the following example:

```
if(FAILED(hr))
  {
   //Code to handle failure
  }
else
  {
   //Code to handle success
  }
```

3.4 Managing the lifetime of COM objects

When an object is created, the system allocates the necessary memory resources. When that object is no longer needed, you should ensure that the resources it has used are freed up once again. To ensure this happens, each object is responsible for deleting itself. However, COM does not let you destroy objects directly, because a given COM object may be used by several applications; if one application destroyed an object, any other applications using that object would fail. Instead, the lifetime of a COM object is managed using a *reference count* system.

An object's reference count is the number of times one of its interfaces has been requested by an application. Each time an interface is requested, the reference count is incremented by 1. When an application has finished with an interface, it releases it, decrementing the reference count by 1. Once an object's reference count has reached zero, it is removed from memory.

Management of an object's lifetime is done primarily through an interface called IUnknown, or through the object creation methods exposed by other interfaces. All COM interfaces must inherit the IUnknown interface in order to manage the lifetime of objects they are exposed in.

Incrementing an object's reference count is done in one of the following ways:

- Calling a public object creation method increments that object's reference counter automatically.
- Calling IUnknown::QueryInterface to return an interface increments the object's reference counter if the call was successful.
- Calling IUnknown::AddRef increments the object's reference counter. You should call this explicitly whenever you obtain a new interface pointer

Decrementing an object's reference count is done by calling IUnknown::Release. You must release all interface pointers, regardless of how the object reference counter was incremented. Using ATL smart interface pointers, as described in Section 3.4.3, can help to make this task simpler.

Performance issue: Failing to release an interface is one of the most common ways of creating memory leaks in an application that uses COM interfaces. You must ensure that reference counting is handled properly in your PS/Workshop modules, since PS/Workshop will not exit correctly if the reference count for any of its interfaces is not zero.

3.4.1 Managing interfaces in method arguments

If one of the received arguments for a method is an interface then you do not need to call either AddRef or Release on the interface pointer. The only exception to this is if your module wishes to keep a copy of the interface – in this case you should call AddRef on the interface pointer, and you must also release the interface later.

If an interface is a return argument for a function, it is the module's responsibility to ensure that the interface is released at some later date. Using ATL smart interface pointers, as described in Section 3.4.3, can help to make this task simpler.

3.4.2 IUnknown

Every COM object must inherit a standard interface called IUnknown, which contains a number of methods that implement the reference count system. IUnknown exposes the following methods:

QueryInterface

```
HRESULT QueryInterface(
  REFIID riid,   // .
  LPVOID* ppvObj //
);
```

Received arguments	
riid	Reference ID of the interface requested
Returned arguments	
ppvObj	Address of interface pointer if successful

Determines whether an object supports a particular interface. If it does, QueryInterface returns the interface and increments the object's reference count.

Use this method to request additional interfaces to the one returned by an object's creation method.

Specific Errors	
E_NOINTERFACE	No such interface supported.
E_POINTER	Invalid pointer.

AddRef

```
ULONG AddRef();
```

Increments the object's reference count.

This method should be called whenever a new interface pointer is obtained. However, you should rarely need to use this method, since the object's reference count is automatically incremented whenever an interface is obtained by calling an object creation method, or when QueryInterface is called.

Release

```
ULONG Release();
```

Releases an interface pointer, and decrements the object's reference count.

As soon as the reference count reaches 0 the object destroys itself.

3.4.3 ATL smart interface pointers

ATL smart interface pointers are used to encapsulate PS/Workshop interfaces in COM objects. Using smart interface pointers simplifies both using and managing COM interfaces; the encapsulated interface is correctly released once the smart pointer goes out of scope, so you do not need to worry about releasing it explicitly.

The following example illustrates the difference between using smart pointers to encapsulate COM interfaces, and using standard pointers:

With smart pointers

```
IPSWDrawList    *pDrawList    // Previously initialised pointer

// create our smart interface pointer to hold our interface
CComPtr<IPSWDrawOpts> pDrawOpts;
HRESULT hr = pDrawList->get_DrawOptions(&pDrawOpts);
if ( SUCCEEDED( hr ) )
{
  // Go off and do something
)
```

Without smart pointers

```
IPSWDrawList    *pDrawList    // Previously initialised pointer

// create a pointer to hold our interface
IPSWDrawOpts *pDrawOpts;
HRESULT hr = pDrawList->get_DrawOptions(&pDrawOpts);
if ( SUCCEEDED( hr ) )
{
  // Go off and do something

  // remember to release the interface when finished with it
  pDrawOpts->Release();
)
```

3.5 Converting between data-types

This section contains a number of approaches to converting between different datatypes that you might find useful when developing modules.

3.5.1 Converting from BSTR to CString

To convert from a BSTR to a CString:

```
BSTR bsDocTitle;        // Previously initialised BSTR
CString csTitle       = bsDocTitle;
```

3.5.2 Creating a BSTR

To create a BSTR, use one of the following methods:

Method 1:

```
CComBSTR bsStr;         // Unitialised BSTR
CString  csString    = _T("Test String");
BsStr  = csString;
```

Method 2:

```
CComBSTR bstrMenuItem( OLESTR("Test String") );
```

Method 3:

```
CString csString      = _T("Test String");
CComBSTR bsStr     = csString.AllocSysString();
```

The CComBSTR is a special ATL (Active Template Library) object that encapsulates a BSTR. Using this construct simplifies the process of handling a BSTR. In particular, it means that the resources of the encapsulated BSTR are correctly freed once the object has gone out of scope.

Event Handling Within A Module

4

4.1 General message handling

In order to be recognized as a PS/Workshop module and loaded by PS/Workshop, all modules must support the IPSWAddIn interface.

In addition, in order to provide event handling capability, all modules must support IPSWEvents interface, so that the module can receive messages from PS/Workshop. See Appendix A, “Interface Functions” for a complete reference of all PS/Workshop interfaces.

If your module does not support the IPSWEvents interface, it can only access CAddonMain level functionality. In particular, it cannot access any documents which are opened in PS/Workshop.

The base class CAddinImpl that is created by the PS/Workshop AppWizard sets up the necessary support for both these interfaces automatically, and encapsulates a CAddonMain object.

Modules created using the PS/Workshop AppWizard route any messages from the CAddinImpl class to the CAddonMain class. Events can then either be handled by the relevant function inside this class, or passed on to CAddonDoc. In turn, events can either be handled inside CAddonDoc or passed to CAddonView to handle. This mechanism is illustrated in Figure 4–1.

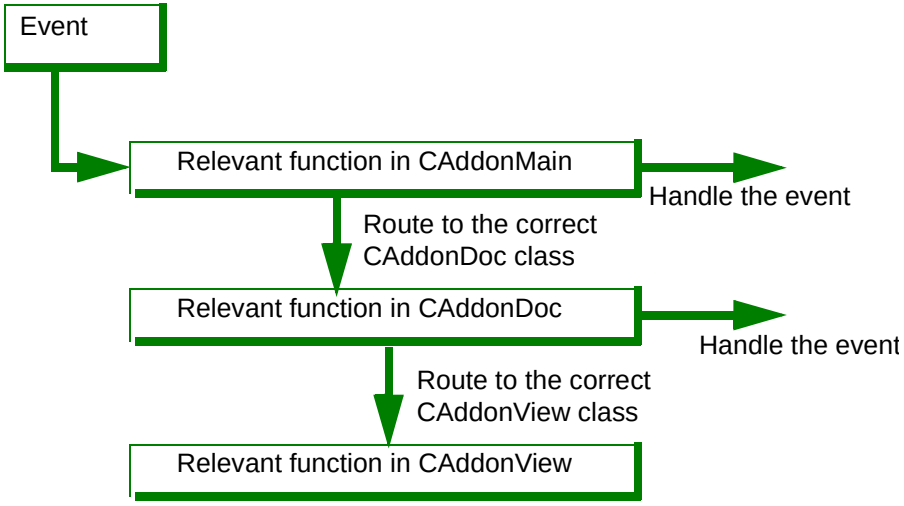

Figure 4–1 The message handling mechanism within a PS/Workshop module

The class you decide to use to handle a given event depends on the functionality you want for the operation concerned, as shown in the table below:

Class handling operation	Functionality available
CAddonMain	Functionality for the operation is available at the application level, i.e. when there are no documents open.
CAddonDoc	Functionality for the operation is available at the document level, i.e. for the current document.
CAddonView	Functionality for the operation is available at the view level, i.e. for the current view of the current document.

The following message handlers are available for each of the classes above. Full information about each message handler is given in Chapter 5, “Module Structure”.

Class	Available message handlers
CAddonMain	`OnCmdMsg OnDocOpen OnDocClose OnViewOpen OnViewClose OnAppDestroy OnSelectTopols OnPartChange`

Class	Available message handlers
CAddonDoc	`OnCmdMsg OnDocClose OnViewOpen OnViewClose OnSelectTopols OnPartChange`
CAddonView	`OnCmdMsg`

Modules created using the PS/Workshop AppWizard use several of the available message handlers to ensure that the list of CAddonDoc and CAddonView classes are correctly created and destoyed.

4.2 Handling menu command events

The OnCmdMsg functions available in CAddonMain, CAddonDoc, and CAddonView can be called from PS/Workshop when the user chooses a menu command that has been added by a module.

As with other events, modules route menu command events from the CAddinImpl class to the relevant function in CAddonMain: in this case, CAddonMain::OnCmdMsg. This function can then either handle the event or call the corresponding CAddonDoc::OnCmdMsg function. In turn, this function can either handle the event or pass it to the CAddonView::OnCmdMsg to handle.

The ID associated with any particular menu command must be unique within the module, otherwise the function called may not be the correct one. Section 2.3, “Adding a Hollow menu item”, provides an example of how you can do this.

Figure 4–2 displays the command handling mechanism in more detail.

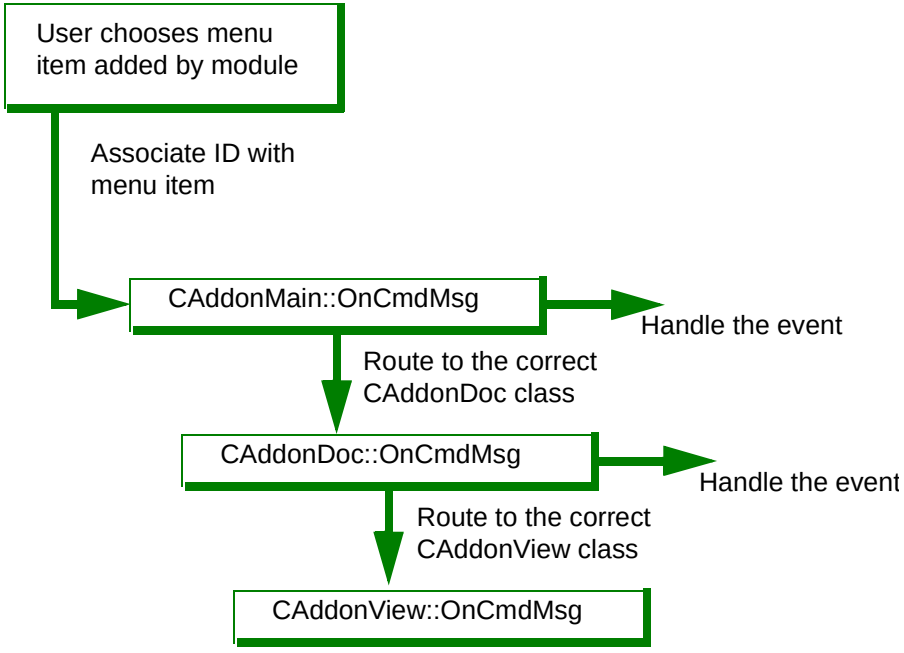

Figure 4–2 The command handling mechanism within a PS/Workshop module

OnCmdMsg can also be called if a user registered File type is opened or saved (as specified using the IPSWApp::RegisterFile*XXX*Functions) in which case the second parameter contains the name of the file to open/save.

When you handle an event using the OnCmdMsg function for a given class, you should set the value of the `hr` variable to S_OK to ensure that no classes further down the event chain are called. An example of how this might be done is shown below:

```
HRESULT CAddonDoc::OnCmdMsg(UINT nID, void* lpparm )
{
  HRESULT hr = S_FALSE;

  // either handle the event here...
  switch( nID )
  {
  case ID_ON_MY_COMMAND:
    OnMyFunction();
    hr = S_OK;    // We need to set this variable here
                  // otherwise the event will
    break;        // be passed onto the view (which means we
                  // could have the
  default:        //event responded to twice
    break;
  }

  // check to see if we have already handled this event
  if ( hr == S_FALSE )
  {
    // try routing this to the active view
    CAddonView* pView = GetActiveView();
    HRESULT hr    = S_FALSE;
    if ( pView )
      hr = pView->OnCmdMsg( nID, lpparm );
  }

  return hr;
}
```

Module Structure 5

This chapter describes the module structure that is created for you automatically when you use the PS/Workshop AppWizard as a basis for a new module. The relationships between the classes produced are detailed in Figure 5–1. A more in-depth description of each class is given in the rest of this chapter. See Appendix A, “Interface Functions” for a description of the interfaces that each class supports.

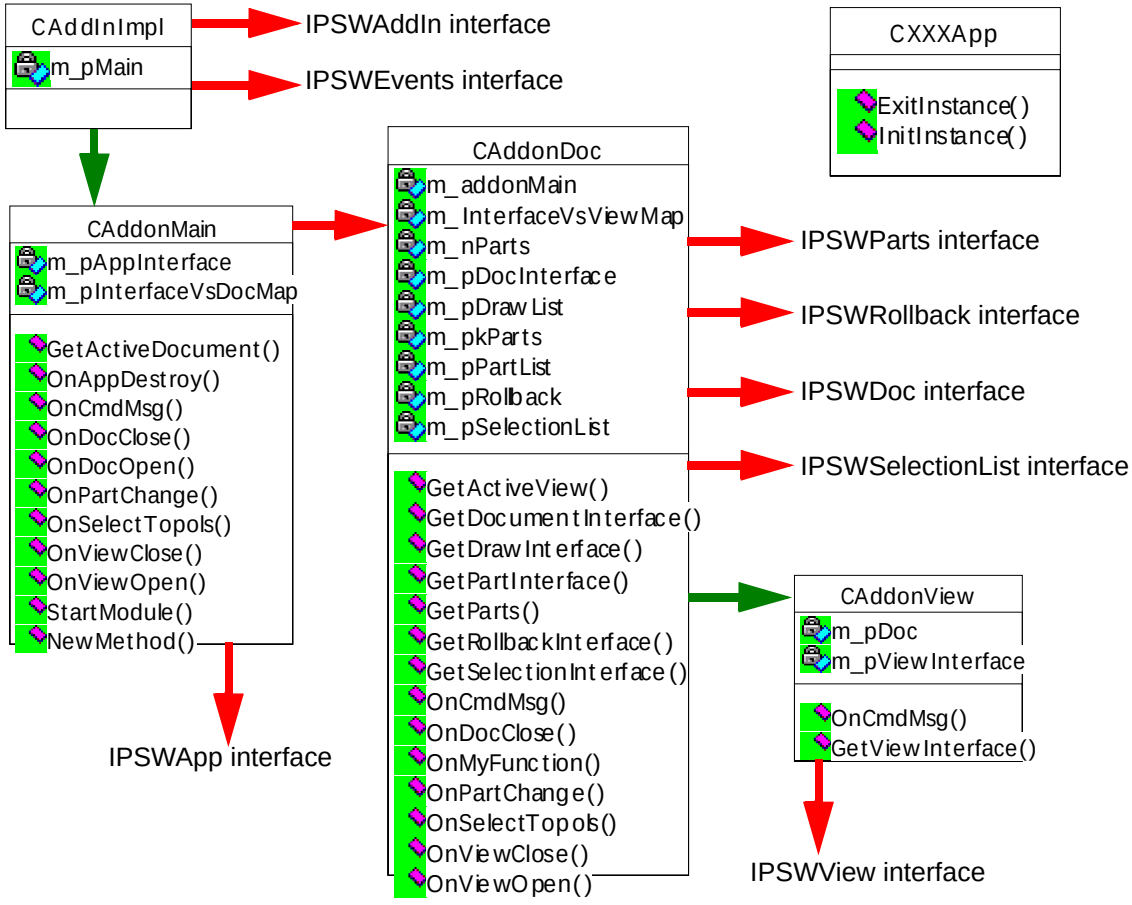

Figure 5–1 Overview of the structure of a module created using the AppWizard

5.1 CAddinImpl

The CAddinImpl class receives all events and messages from PS/Workshop and passes them to the encapsulated CAddonMain class.

It is also responsible for instantiating the class itself (IPSWAddIn::OnConnection) and deleting the class (IPSWAddIn::OnDisconnection).

This class supports both the IPSWAddIn and IPSWEvents interfaces. During the initial IPSWAddIn::OnConnection event, CAddinImpl creates an instance of the CAddonMain class. Subsequent messages from PS/Workshop are passed to this encapsulated CAddonMain object.

CAddinImpl has a single data member:

Data member	Description
`CAddonMain *m_pMain`	An instance of the CAddonMain class

5.2 CXXXApp

The CXXXApp class is an MFC-generated module application object, and is provided as part of the MFC framework of the module. You should place any initialisation code for a module in either the CAddinImpl or CAddonMain classes, rather than this class.

5.2.1 CXXXApp Functions

CXXXApp contains the following functions.

InitInstance

```
BOOL InitInstance()
```

Performs initialisation of the module.

ExitInstance

```
int ExitInstance()
```

Performs clean-up of the application.

5.3 CAddonMain

The CAddonMain class receives all events and messages from the CAddinImpl class. These can either be handled by the appropriate message handler in CAddonMain, or passed on to CAddonDoc for handling.

5.3.1 Summary

The following is a summary of the CAddonMain class.

Data members	Description
`CComQIPtr<IPSWApp> m_pAppInterface`	A pointer to the PS/Workshop IPSWApp interface
`CMap<IPSWDoc*, IPSWDoc*, CAddonDoc*, CAddonDoc* > m_InterfaceVsDocMap`	Maintains a mapping between the list of the IPSWDoc interface pointer and the associated CAddonDoc class

Constructors	Description
`CAddonMain`	Constructs the CAddonMain object

Message handlers	Description
`OnDocClose`	Handles the IPSWEvents::OnDocClose message
`OnDocOpen`	Handles the IPSWEvents::OnDocOpen message
`OnPartChange`	Handles the IPSWEvents::OnPartChange message
`OnSelectTopols`	Handles the IPSWEvents::OnSelectTopols message
`OnViewClose`	Handles the IPSWEvents::OnViewClose message
`OnViewOpen`	Handles the IPSWEvents::OnViewOpen message
`OnCmdMsg`	Handles the IPSWEvents::OnCommand message

Module methods	Type	Description
`GetActiveDocument`	`CAddonDoc*`	Returns the currently active CAddonDoc class

Initialiser/Destructors	Description
`StartModule`	Handles the IPSWAddIn::OnConnection event
`OnAppDestroy`	Handles the IPSWAddIn::OnDisconnection event

5.3.2 CAddonMain functions

CAddonMain contains the following functions.

CAddonMain

Constructor

```
CAddonMain
(
--- received arguments ---
   IDispatch *pIPSWApp
)
```

Constructs a CAddonMain object.

Received arguments	
pIPSWApp	Pointer to the PS/Workshop dispatch interface

OnDocClose

Message handler

```
HRESULT  OnDocClose
(
--- received arguments ---
   IPSWDoc *pDoc
)
```

Handles the IPSWEvents::OnDocClose event.

Received arguments	
pDoc	The IPSWDoc interface associated with this document

This function traverses the interface/document map (m_pInterfaceVsDocMap) and passes the event to the correct document to handle.

OnDocOpen

Message handler

```
HRESULT  OnDocOpen
(
--- received arguments ---
   IPSWDoc *pDoc
)
```

Handles the IPSWEvents::OnDocOpen event.

Received arguments	
pDoc	The IPSWDoc interface associated with this document

This function creates a new CAddonDoc object and adds this to the interface/document map (m_pInterfaceVsDocMap).

OnPartChange

Message handler

```
HRESULT OnPartChange()
```

Handles the IPSWEvents::OnPartChange event.

This function passes the event to each of the documents.

OnSelectTopols

Message handler

```
HRESULT OnSelectTopols()
```

Handles the IPSWEvents::OnSelectTopols event.

This function passes the event to each of the documents.

OnViewClose

Message handler

```
HRESULT OnViewClose
(
--- received arguments ---
   IPSWDoc *pDoc,
   IPSWView *pView
)
```

Handles the IPSWEvents::OnViewClose event.

Received arguments	
pDoc	The IPSWDoc interface associated with this view
pView	The IPSWView interface associated with this view

This function traverses the interface/document map (m_pInterfaceVsDocMap) for the associated document and passes the event to it.

OnViewOpen

Message handler

```
HRESULT  OnViewOpen
(
--- received arguments ---
   IPSWDoc *pDoc,
   IPSWView *pView
)
```

Handles the IPSWEvents::OnViewOpen event.

Received arguments	
pDoc	The IPSWDoc interface associated with this view
pView	The IPSWView interface associated with this view

This function traverses the interface/document map (m_pInterfaceVsDocMap) and passes the event to the correct CAddonDoc.

OnCmdMsg

Message handler

```
HRESULT  OnCmdMsg
(
--- received arguments ---
   UINT  nID,
   void *lpparam = NULL
)
```

Handles the IPSWEvents::OnCommand event.

Received arguments	
nID	The ID associated with this command
lpparam	Extra data related to the command

The OnCmdMsg function looks for the function associated with the ID in the CAddonMain class. If it fails to find one, the event is passed to the CAddonDoc class to handle.

GetActiveDocument

Module method

```
CAddonDoc* GetActiveDocument()
```

This function returns the currently active CAddonDoc class. If there is no currently active document it returns NULL.

StartModule

Initializer

```
HRESULT  StartModule
(
--- received arguments ---
   IPSWAddIn *pApp
)
```

This function handles initialisation of the PS/Workshop module. It is called as a result of IPSWAddIn::OnConnection.

Received arguments	
pApp	The IPSWAddIn interface of the module

You should add any one-time initialisation code for your module here. This includes code for adding menu items, as well as registering any callback functions.

If this function returns a failure code then the CAddinImpl::OnConnection deletes the CAddonMain class and itself returns a failure which forces PS/Workshop to unload the module.

OnAppDestroy

Destructor

```
HRESULT  OnAppDestroy()
```

This function handles destruction of the module. It is called as a result of IPSWAddIn::OnDisconnection.

OnAppDestroy traverses the interface/document map (m_pInterfaceVsDocMap) deleting each document.

5.4 CAddonDoc

The CAddonDoc class represents a module document that shadows the current PS/Workshop document.

5.4.1 Summary

The following is a summary of the CAddonDoc class.

Data members	Description
`CAddonMain *m_addonMain`	The parent CAddonMain class
`int m_nParts`	The number of parts in the document
`PK_PART_t *m_pkParts`	A copy of the parts in the document
`CComPtr<IPSWDoc> m_pDocInterface`	The associated IPSWDoc smart interface pointer
`CComPtr<IPSWParts> m_pPartList`	The associated IPSWParts smart interface pointer
`CComPtr<IPSWSelectionList> m_pSelectionList`	The associated IPSWSelectionList smart interface pointer
`CComPtr<IPSWDrawList> m_pDrawList`	The associated IPSWDrawList smart interface pointer
`CcomPtr<IPSWRollback> m_pRollback`	The associated IPSWRollback smart interface pointer
`CMap< IPSWView*, IPSWView*, CAddonView*, CAddonView* > m_InterfaceVsViewMap`	Maintains a list of the IPSWView interfaces and the associated CAddonView class

Constructors	Description
`CAddonDoc`	Constructs the CAddonDoc object

Message handlers	Description
`OnDocClose`	Handles the CAddonMain::OnDocClose message
`OnPartChange`	Handles the CAddonMain::OnPartChange message
`OnSelectTopols`	Handles the CAddonMain::OnSelectTopols message
`OnViewClose`	Handles the CAddonMain::OnViewClose message
`OnViewOpen`	Handles the CAddonMain::OnViewOpen message
`OnCmdMsg`	Handles the CAddonMain::OnCmdMsg message

Interface access functions	Description
`CComPtr<IPSWDoc>& GetDocumentInterface`	Returns the IPSWDoc interface
`CComPtr<IPSWSelectionList>& GetSelectionInterface`	Returns the IPSWSelectionList interface
`CComPtr<IPSWDrawList>& GetDrawInterface`	Returns the IPSWDrawList interface

Interface access functions	Description
`CComPtr<IPSWParts>& GetPartInterface`	Returns the IPSWParts interface
`CComPtr<IPSWRollback>& GetRollbackInterface`	Returns the IPSWRollback interface

Module methods	Type	Description
`GetActiveView`	`CAddonView*`	Returns the currently active view
`GetParts`	`HRESULT`	Returns the number of parts in the document

Custom functions	Description
`OnMyFunction`	Demonstration User extensible function.

5.4.2 CAddonDoc functions

CAddonDoc contains the following functions.

CAddonDoc

Constructor

```
CAddonDoc
(
--- received arguments ---
   IPSWDoc *pDoc,
   CAddonMain *pMain
)
```

This function constructs a CAddonDoc object and sets the values of the following interfaces:

- IPSWSelectionList
- IPSWDrawList
- IPSWRollback
- IPSWParts

Received arguments	
`pDoc`	The associated IPSWDoc interface
`pMain`	The parent CAddonMain

The CAddonDoc constructor also queries and sets the correct values for m_nParts and m_pkParts.

OnDocClose

Message handler

```
HRESULT  OnDocClose
(
--- received arguments ---
   IPSWDoc *pDoc
)
```

This function handles any CAddonMain::OnDocClose events passed to it from CAddonMain.

Received arguments	
pDoc	The IPSWDoc interface associated with this document

It traverses the view/interface map (m_InterfaceVsViewMap) and deletes the associated view.

OnPartChange

Message handler

```
HRESULT OnPartChange()
```

This function handles the CAddonMain::OnPartChange events passed to it from CAddonMain.

It does the following:

- Deletes the `m_pkParts` array, which stores a copy of the parts in the document.
- Sets `m_nparts` (the length of `m_pkParts`) to zero.
- Gets the parts in the document from PS/Workshop again, and places them in `m_pkParts`.
- Sets `m_nparts` accordingly.

OnSelectTopols

Message handler

```
HRESULT OnSelectTopols()
```

This function handles the CAddonMain::OnSelectTopols events passed to it from CAddonMain.

Events can either be handled here or passed to the active view using the CAddonView class.

OnViewClose

Message handler

```
HRESULT OnViewClose
(
--- received arguments ---
   IPSWDoc *pDoc,
   IPSWView *pView
)
```

This function handles the CAddonMain::OnViewClose events passed to it from CAddonMain.

Received arguments	
pDoc	The IPSWDoc interface associated with this view
pView	The IPSWView interface associated with this view

It traverses the view/interface map (m_InterfaceVsViewMap) and deletes the associated view.

OnViewOpen

Message handler

```
HRESULT OnViewOpen
(
--- received arguments ---
   IPSWDoc *pDoc,
   IPSWView *pView
)
```

This function handles the CAddonMain::OnViewOpen events passed to it from CAddonMain.

Received arguments	
pDoc	The IPSWDoc interface associated with this view
pView	The IPSWView interface associated with this view

It traverses the view/interface map (m_InterfaceVsViewMap) and opens the associated view.

OnCmdMsg

Message handler

```
HRESULT  OnCmdMsg
(
--- received arguments ---
   UINT nID,
   void *lpparam = NULL
)
```

This function handles CAddonMain::OnCmdMsg events passed to it from CAddonMain.

Received arguments	
nID	The ID associated with this command
lpparam	Extra data related to the command

It looks for a handler for the nID message in the CAddonDoc class. If it is unable to find a handler then the event is passed to the CAddonView class.

GetDocumentInterface

Interface access function

```
CComPtr<IPSWDoc>& GetDocumentInterface()
```

This function returns the IPSWDoc interface.

GetSelectionInterface

Interface access function

```
CComPtr<IPSWSelectionList>& GetSelectionInterface()
```

This function returns the IPSWSelectionList interface.

GetDrawInterface

Interface access function

```
CComPtr<IPSWDrawList>& GetDrawInterface()
```

This function returns the IPSWDrawList interface.

GetPartInterface

Interface access function

```
CComPtr<IPSWParts>& GetPartInterface()
```

This function returns the IPSWParts interface.

GetRollbackInterface

Interface access function

```
CComPtr<IPSWRollback>& GetRollbackInterface()
```

This function returns the IPSWRollback interface.

GetActiveView

Module method

```
CAddonView* GetActiveView()
```

This function returns the currently active view. If there is no active view then this will return NULL.

GetParts

Module method

```
HRESULT  GetParts
(
--- returned arguments ---
   int &nParts,
   PK_PART_t *&pkParts
)
```

This function returns the number of parts in the document. It interrogates and returns a copy of the parts in the PS/Workshop document.

Returned arguments	
&nParts	The number of parts
&pkParts	The parts

This function allocates an array of the parts in PS/Workshop. It is up to the module to correctly free the returned array, using `delete[]`.

Note: This function does not return m_nPart and m_pkParts variables defined in the CAddonDoc class

OnMyFunction

Custom function

```
HRESULT OnMyFunction()
```

This function is supplied to demonstrate the message handling functionality.

When the module as created by the AppWizard is compiled and built, a new menu item is added to PS/Workshop. OnMyFunction is called when the user clicks on this new menu item. OnMyFunction simply displays a message indicating that this function has been reached.

5.5 CAddonView

The CAddonView class represents a module view that shadows the current PS/Workshop view.

5.5.1 Summary

The following is a summary of the CAddonView class.

Data members	Description
`CAddonDoc* m_doc`	A pointer to the parent CAddonDoc
`CComQIPtr<IPSWView2> m_pViewInterface`	The associated IPSWView2 interface

Constructors	Description
`CAddonView`	Constructs the CAddonView object

Message handlers	Description
`OnCmdMsg`	Handles the CAddonDoc::OnCmdMsg message

Interface access functions	Description
`CComQIPtr<IPSWView2>& GetViewInterface`	Returns the encapsulated IPSWView2 interface

5.5.2 CAddonView functions

CAddonView contains the following functions.

CAddonView

Constructor

```
CAddonView
(
--- received arguments ---
   IPSWView *pView,
   CAddonDoc *pDoc
)
```

This function constructs the CAddonView object.

Received arguments	
pView	The associated IPSWView interface
pDoc	The CAddonDoc associated with this CAddonView

OnCmdMsg

Message handler

```
HRESULT OnCmdMsg
(
--- received arguments ---
   UINT nID,
   void *lpparam
)
```

This function handles the CAddonDoc::OnCmdMsg message.

Received arguments	
nID	The ID associated with this message
lpparam	Optional extra information for this message

GetViewInterface

Interface access function

```
CComQIPtr<IPSWView2>& GetViewInterface()
```

This function returns the encapsulated IPSWView2 interface.

The PS/Workshop Interfaces

6

PS/Workshop functionality is made available to modules through a number of COM interfaces. These interfaces have been designed to correspond as closely as possible to the Microsoft Foundation Classes (MFC) Multiple Document Interface (MDI) paradigm common in many Windows-based applications, including PS/Workshop.

There are 3 main interfaces:

- IPSWApp: associated with the PS/Workshop application
- IPSWDoc: associated with an open document in PS/Workshop
- IPSWView2: associated with a given view on an open document in PS/Workshop

Similarly, the structure of a module created using the PS/Workshop AppWizard is designed to emulate the MDI structure, and provides the following classes:

- CAddonMain, which contains a pointer to the IPSWApp interface
- CAddonDoc, which contains a pointer to the IPSWDoc interface related to the document
- CAddonView, which contains a pointer to the IPSWView interface associated with the view.

These classes are described in full in Chapter 5, “Module Structure”.

Interfaces	Description
IPSWApp	The IPSWApp interface represents the overall PS/Workshop application. There is exactly one IPSWApp interface for each instance of PS/Workshop. The IPSWApp interface is supplied to a module when the module is first loaded. The IPSWApp interface contains functions for modifying the PS/Workshop user interface, opening documents and registering callback functions.
IPSWDoc	The IPSWDoc interface represents each of the PS/Workshop parts (documents) currently open. For each open document there exists a corresponding IPSWDoc interface pointer. Those functions called from IPSWDoc only affect the associated PS/Workshop document.

Interfaces	Description
IPSWView	This interface has been superseded by IPSWView2
IPSWView2	Each document open in PS/Workshop has one or more views attached. Each IPSWView is associated with an IPSWDoc. **Note:** In the current version of PS/Workshop there is only one IPSWView associated with each IPSWDoc.
IPSWParts IPSWSelectionList IPSWRollback IPSWDrawList	Each of these interfaces can be obtained from an IPSWDoc interface, and as such they only affect the associated document. They allow control over ■ the number of Parasolid parts in the document ■ the selection of entities ■ Parasolid partitioned rollback ■ custom drawing of entities respectively.
IPSWEnumParts IPSWEnumSelectionList IPSWEnumDrawList	Each of these interfaces can be obtained from the associated IPSW interface (for example IPSWEnumParts can be obtained from IPSWParts). They allow the entities associated with each interface to be enumerated.
IPSWDrawOpts	This interface can be obtained from IPSWDrawList (in which case it contains the current draw options). It provides control over how particular entities are displayed in the draw list. IPSWDrawOpts is somewhat different from other interfaces, in that it can also be created and passed to the IPSWDrawList interface (in which case it contains the new draw options).

Figure 6–1 shows how the various PS/Workshop interfaces interact. For a complete reference for all the functions available in each COM interface, see Appendix A, “Interface Functions”. For an introduction to the PS/Workshop draw list, see Chapter 9, “The Draw List”.

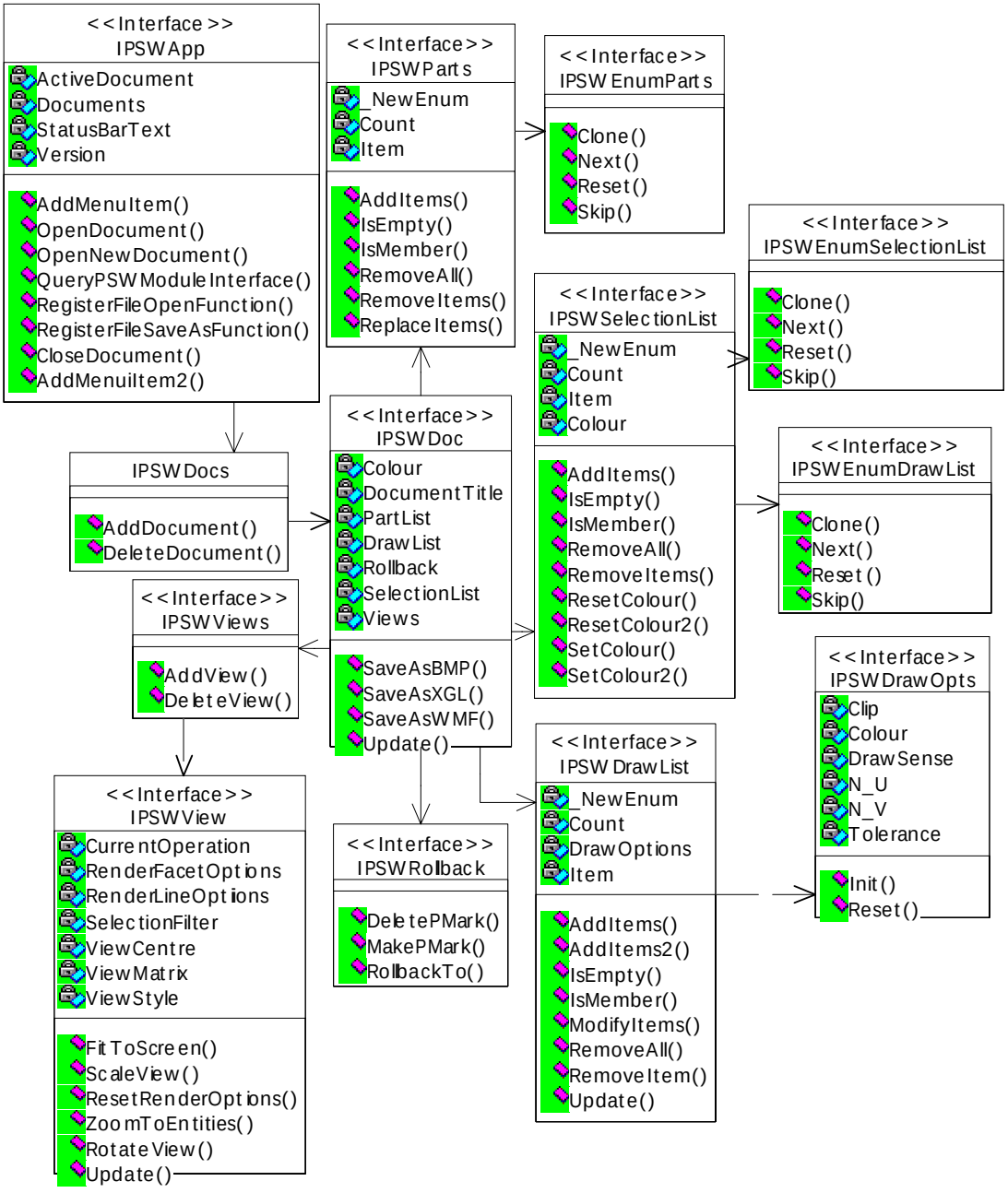

Figure 6–1 The PS/Workshop COM interface

Adding a New Menu Item to PS/Workshop

7

You can add a new menu item to PS/Workshop using IPSWApp::AddMenuItem or IPSWApp::AddMenuItem2. Typically this is done in CAddonMain::StartModule, which is called when the module is first loaded.

In order to add a new menu item you need to do the following:

- Define an ID to be associated with the new menu item. This ID must be unique within your module.

  The easiest way of doing this is to add a new ID via the Resources Symbol dialog in Visual Studio (**View > Resource Symbols**).

- Decide which class you want to handle the new menu item in. The class you choose depends on whether the menu item needs to operate at the application, document, or view level.
- Add a function to the chosen class. This function should return HRESULT.
- Modify OnCmdMsg in the chosen class to ensure that your function is called when it receives the ID.

  For example, suppose that you wish to add a function called OnBlend with the ID ID_COMMAND_ON_BLEND to the CAddonDoc class. In this case, the CAddonDoc::OnCmdMsg would be as follows:

```
HRESULT CAddonDoc::OnCmdMsg(UINT nID, void* lpparm )
{
HRESULT hr = S_FALSE;

  // either handle the event here...
  switch( nID )
  {
  case ID_COMMAND_ON_BLEND:
    OnBlend();
    hr = S_OK;
    break;
  default:
    break;
  }

// check if the message has already been handled
  if ( hr == S_FALSE)
  {
    // try routing this to the active view
    CAddonView* pView = GetActiveView();
    if ( pView )
      hr = pView->OnCmdMsg( nID, lpparm );
  }

  return hr;
}
```

- Add a call to AddMenuItem in CAddonMain::StartModule to add the menu item.

  For example, the following code above adds an **Operations** menu containing a **Blend** command to the document level of PS/Workshop (i.e. the menu is only available when a document has been opened) . The ID ID_COMMAND_ON_BLEND is also associated with the menu item.

```
CComBSTR bstrMenuItem( OLESTR("&Operations\nBlend") );
HRESULT hr = m_pAppInterface->AddMenuItem(pApp, bstrMenuItem,
                   ID_COMMAND_ON_BLEND, PSW_Menu_Doc);
```

IPSWApp::AddMenuItem2 provides additional functionality to AddMenuItem, so that you can specify the precise position of any menu or menu item in the PS/Workshop menu bar.

Registering Handlers for Different Filetypes

8

If required, your module can load files into or save files from PS/Workshop with specific file extensions. In order to do this, you must register functions with PS/Workshop that are called when trying to either open or save files with the file extensions you want to use.

When it is called, each registered function is passed the filename of the file to open or save. It is then the responsibility of the registered function to correctly open or save the file.

The following functions in the IPSWApp interface can be used to register handlers for different filetypes:

Function	Description
RegisterFileOpenFunction	This function takes a description string that contains two parts: ■ The first part is the text that appears as an entry in the “Files of type” box in the File Open dialog. ■ The second part contains information about the file extensions to be associated with the function. See the RegisterFileOpenFunction documentation for details of the exact formatting of the string.
RegisterFileSaveAsFunction	This function is similar to RegisterFileOpenFunction, but the description appears in the File Save As dialog rather than the File Open dialog. See the RegisterFileSaveAsFunction documentation for details of the exact formatting of the string.

The Draw List 9

9.1 Specifying which parts to render

By default, PS/Workshop attempts to render all the parts in a document. Sometimes, though, this might not be the best method: for example, you may only be interested in rendering a specific area of the document, or only one part in the document. In such cases you need to use the draw list.

The draw list provides control over which entities to display, and how they should be displayed. There is one draw list associated with each document. Using the draw list also lets you display geometry in wireframe modes.

The following entity classes are supported by the draw list:

- PK_CLASS_body
- PK_CLASS_face
- PK_CLASS_edge
- PK_CLASS_vertex
- PK_CLASS_point
- PK_CLASS_curve and all subclasses
- PK_CLASS_surf and all subclasses

The draw list can be obtained from a document through the IPSWDoc::DrawList interface property.

Initially, the draw list for a given document is empty. Whenever this is the case, PS/Workshop draws all the parts in the document. If entities are subsequently added to the draw list, only those entities are displayed.

Entities can be added to the draw list using IPSWDrawList::AddItems (which adds items to the draw list using the default draw options) or IPSWDrawList::AddItems2 (which adds items to the draw list using a specified set of draw options). An entity can only exist in a draw list once – attempting to add it again will cause the AddItems or AddItems2 (whichever was called) to fail.

Entities can be removed from the draw list using IPSWDrawList::RemoveItems or IPSWDrawList::RemoveAll. Once the draw list is empty, PS/Workshop once again displays all parts in the document.

9.2 Setting drawing options

As well as adding entities to a draw list you can control how those entities are displayed using the IPSWDrawOpts interface. This interface is used as an argument for a number of the draw functions.

IPSWDrawOpts lets you control the following properties:

Properties	Description	Entities affected
Clip	Clip entity to part box (currently ignored)	curves surfaces
Colour	Colour in which to draw entity	geometry topology
DrawSense	Whether to display sense of entity	edges curves faces surfaces
N_U	Number of U param hatch lines to display	surfaces
N_V	Number of V param hatch lines to display	surfaces
Tolerance	Whether to display tolerance of entity	edges vertices
ToleranceColour	Colour in which to display entity tolerance	edges vertices

Further details of both the IPSWDrawList and IPSWDrawOpts interfaces can be found in Appendix A, “Interface Functions”.

Interface Functions A

A.1 Introduction

This chapter provides a complete reference to all the functions and properties available in the COM interfaces provided by PS/Workshop.

- For a more general introduction to the interfaces available, see Chapter 6, “The PS/Workshop Interfaces”.
- For an explanation of the support your module should provide for the interfaces, and the support provided by default when you use the PS/Workshop AppWizard, see Chapter 5, “Module Structure”.

As for all COM interfaces, the interfaces described in this chapter inherit the IUnknown interface. That is, they all have QueryInterface, AddRef, and Release methods to control the lifetimes of objects that expose the interface. See Section 3.4, “Managing the lifetime of COM objects”, for more information.

A.2 IPSWApp

The IPSWApp interface represents the overall PS/Workshop application. There is exactly one IPSWApp interface for each instance of PS/Workshop. The IPSWApp interface is supplied to a module when the module is first loaded.

The IPSWApp interface contains functions for modifying the PS/Workshop user interface, opening documents and registering callback functions.

A.2.1 Summary

The following is a summary of the IPSWApp interface.

Properties

Property	Type	Description
`ActiveDocument`	`IPSWDoc`	The currently active document
`Documents`	`IPSWDocs`	The document collection object
`StatusBarText`	`BSTR`	The status bar text
`Version`	`double`	The current PS/Workshop version

Functions

Function	Description
AddMenuItem	Adds an item to a PS/Workshop menu
AddMenuItem2	Adds an item to a PS/Workshop menu in a specific position
OpenDocument	Loads an existing Parasolid partfile
OpenNewDocument	Opens a new (empty) document
CloseDocument	Closes a given document
QueryPSWModuleInterface	Queries for the existence of a particular interface
RegisterFileOpenFunction	Registers a custom FileOpen format
RegisterFileSaveAsFunction	Registers a custom FileSaveAs format

A.2.2 IPSWApp Properties

ActiveDocument

```
HRESULT   get_ActiveDocument
(
--- returned arguments ---
   IPSWDoc **ppDoc
)
```

Returns the IPSWDoc interface pointer associated with the currently active document. If there is no active document this returns S_FALSE.

Returned arguments	
ppDoc	The Active document

When get_ActiveDocument returns S_FALSE or E_FAIL, the value of ppDoc is undefined.

```
HRESULT   put_ActiveDocument
(
--- received arguments ---
   IPSWDoc *pDoc
)
```

Attempts to activate the document associated with the pDoc interface.

Received arguments	
pDoc	The document to activate

Specific Errors	
E_INVALIDARG	There is no document associated with the given pDoc.

Documents

```
HRESULT   get_Documents
(
--- returned arguments ---
   IPSWDocs **pDocs
)
```

Returns the IPSWDocs interface (a collection of IPSWDoc interfaces) which can be used to interate over the documents in PS/Workshop.

Returned arguments	
pDocs	The IPSWDocs interface

Note: The Documents property is intended for internal use. If you wish to examine the list of open documents in PS/Workshop, use the m_InterfaceVsDocMap member variable in the CAddonMain class.

StatusBarText

```
HRESULT   get_StatusBarText
(
--- returned arguments ---
   BSTR *pText
)
```

Returns the current text in the PS/Workshop status bar.

Returned arguments	
pText	The current status bar text

Note: It is reponsibility of the module to free the pText resource.

Example

```
// allocated previously
CComPtr<IPSWApp> pApp;

CComBSTR bsStatusText;
HRESULT hr      = pApp->get_StatusBarText(&bsStatusText);
if (SUCCEEDED( hr ) )
{
   // convert the BSTR into a more usable CString
   CString csStatusText = bsStatusText;
}
```

```
HRESULT   put_StatusBarText
(
--- received arguments ---
   BSTR Text
)
```

Sets the text in the PS/Workshop status bar.

Returned arguments	
Text	The new status bar text

Example

```
// allocated previously
CComPtr<IPSWApp> pApp;

CComBSTR bstrStatusText( OLESTR("Set Status Text") );
// or alternatively
// Cstring csStatusText = _T("Set Status Text");
// bstrStatusText.AllocSysString( bstrStatusText );

if (SUCCEEDED( pApp->put_StatusBarText ) )
{
   // convert the BSTR into a more usable CString
   CString csStatusText = bsStatusText;
}
```

Version

```
HRESULT   get_Version
(
--- returned arguments ---
   double *pVersion
)
```

Returns the current version of PS/Workshop.

Returned arguments	
pVersion	The version.

A.2.3 IPSWApp Functions

AddMenuItem

```
HRESULT   AddMenuItem
(
--- received arguments ---
   IPSWAddIn *pAddIn,
   BSTR CommandName,
   long CommandID,
   PSW_Menu_Mode  mode
)
```

Adds an item to a PS/Workshop menu. If the menu does not already exist then it is created first.

Received arguments	
pAddIn	IPSWAddIn interface of module
CommandName	Menu/Menu Item to add
CommandID	The ID associated with the item
mode	How to add the item

CommandName must be a NULL separated string.

- The last part of the string contains the text of the menu item to add.
- The first part of the string contains the text of the menu to add.

You can add a cascading menu to PS/Workshop by specifying a string that contains more than two parts, as follows:

```
Menu\nCommand\nSub-Command
```

You can use the & character in CommandName to specify keyboard mnemonics for your menu and menu item that the user can use to access the new command. The character immediately after any & character in CommandName is used as the keyboard mnemonic for that part of the string. For example, the string

```
&Menu\nC&ommand
```

would create a **Menu** menu containing a **Command** item. A PS/Workshop user could access the new **Command** by typing Alt+M+O.

Note: You must take care to specify keyboard mnemonics that are unique and consistent. Specifying both `&Debug` and `D&ebug` in different calls to AddMenuItem results in two **Debug** menus being added to PS/Workshop: one accessible using Alt+D and the other accessible using Alt+E. You must also take care not to specify mnemonics that have already been used at that level.

`CommandID` contains the ID to be associated with the menu item. This is subsequently passed back to the module during the IPSWEvents::OnCommand event when the user chooses the menu item.

Note: The CommandID should be unique within a module.

The `mode` argument may have the following values:

Value	Description
PSW_Menu_App	Add the menu to the application level (such a menu is available before any document is open).
PSW_Menu_Doc	Add the menu at a document level (the menu is only available once a document is open).

See Chapter 7, “Adding a New Menu Item to PS/Workshop” for more information about adding menu items to PS/Workshop.

Specific Errors	
E_INVALIDARG	One of the arguments is invalid
PSWERR_ITEMALREADYEXISTS	The item on the specified menu already exists

Example The following example adds a **Debug** menu to PS/Workshop that contains an **Analyse** command. When the user chooses this command the OnCommand message on the module is called with the `CommandID` ID_ON_ANALYSE. The & characters specify that the user can press Alt+D+A on the keyboard to access the Analyse command.

```
HRESULT hr = E_FAIL;
CComBSTR bstrMenuItem( OLESTR("&Debug\n&Analyse") );
hr = m_pAppInterface->AddMenuItem( pApp, bstrMenuItem,
ID_ON_ANALYSE, PSW_Menu_Doc);
if ( SUCCEED(hr) )
{
   // … do something
}
```

AddMenuItem2

```
HRESULT   AddMenuItem
(
--- received arguments ---
   IPSWAddIn *pAddIn,
   BSTR  CommandName,
   long CommandID,
   PSW_Menu_Mode mode
   long menuPos,
   long itemPos
)
```

Adds an item to a PS/Workshop menu, specifying the precise position of the menu and menu item. If the menu does not already exist then it is created first.

Received arguments	
`pAddIn`	IPSWAddIn interface of module
`CommandName`	Menu/Menu Item to add
`CommandID`	The ID associated with the item
`mode`	How to add the item
`menuPos`	The position in the existing PS/Workshop menu to add the new menu
`itemPos`	The position in the menu to add the new menu item to

`CommandName` must be a NULL separated string.

- The last part of the string contains the text of the menu item to add.
- The first part of the string contains the text of the menu to add.

You can add a cascading menu to PS/Workshop by specifying a string that contains more than two parts, as described in the documentation for AddMenuItem.

You can use & to specify keyboard mnemonics for menus and menu items, as described in the documentation for AddMenuItem.

`CommandID` contains the ID to be associated with the menu item. This is subsequently passed back to the module during the IPSWEvents::OnCommand event when the user chooses the menu item.

Note: The CommandID should be unique within a module.

The `mode` argument may have the following values:

Value	Description
PSW_Menu_App	Add the menu to the application level (such a menu is available before any document is open).
PSW_Menu_Doc	Add the menu at a document level (the menu is only available once a document is open).

See Chapter 7, “Adding a New Menu Item to PS/Workshop” for more information about adding menu items to PS/Workshop.

Specific Errors	
E_INVALIDARG	One of the arguments is invalid
PSWERR_ITEMALREADYEXISTS	The item on the specified menu already exists

The `menuPos` argument lets you specify the position of the new menu in the PS/Workshop menu bar. Setting this to 0 makes the new menu the first one on the menu bar (at the left), setting it to 1 makes it the second menu, and so on. A value of -1 appends the new menu to the right hand end of the menu bar.

Note: Although you can place new menus anywhere in the PS/Workshop menu bar, you should ensure that you conform to GUI design guidelines when designing your module.

The `itemPos` argument lets you specify the position of the new menu item in the PS/Workshop menu. Setting this to 0 makes the new item the first one on the menu, setting it to 1 makes it the second item, and so on. A value of -1 appends the new item to the end of the specified menu.

Example The following example adds a **Blend** menu to PS/Workshop that contains a **Notch** command. When the user chooses this command the OnCommand message on the module is called with the `CommandID` ID_ON_NOTCH. The **Blend** menu is placed at position 5, which ensures that it appears between the **Window** and **Help** menus in PS/Workshop, and the **Notch** command is placed

at position 0, which ensures it is the first command in the **Blend** menu, regardless of other commands that may have been added.

```
HRESULT hr = E_FAIL;
CComBSTR bstrMenuItem( OLESTR("Bl&end\nNotch") );
hr = m_pAppInterface->AddMenuItem2( pApp, bstrMenuItem,
ID_ON_NOTCH, PSW_Menu_Doc, 5, 0);
if ( SUCCEEDED(hr) )
{
   // … do something
}
```

OpenDocument

```
HRESULT   OpenDocument
(
--- received arguments ---
   BSTR fileName,
--- returned arguments ---
   IPSWDoc **ppDoc
)
```

Loads a new part from file and opens a corresponding module document.

Received arguments	
`fileName`	The partfile to open
Returned arguments	
`ppDoc`	Pointer to the newly opened document interface

Note: It is a module's responsibility to ensure that the `ppDoc` interface is correctly managed

Example

```
// Previously initialised pointer
CComPtr<IPSWApp>         pApp;

// our variable to store the returned interface
CcomPtr<IPSWDoc>  pDoc = NULL;
HRESULT hr = pApp->OpenDocument("c:\\TestDoc.x_b", &pDoc );
if ( SUCCEEDED(hr))
{
   //… do something with the pDoc
}
```

OpenNewDocument

```
HRESULT   OpenNewDocument
(
--- returned arguments ---
   IPSWDoc **ppDoc
)
```

Opens a new document in PS/Workshop.

Returned arguments	
ppDoc	Pointer to the newly opened document interface

Note: It is a module's responsibility to ensure that the ppDoc interface is correctly managed

Usage:

```
// Previously initialised pointer
CComPtr<IPSWApp>         pApp;

// our variable to store the returned interface
CcomPtr<IPSWDoc>  pDoc = NULL;
HRESULT hr = pApp->OpenNewDocument(&pDoc);
If ( SUCCEEDED(hr))
{
   //… do something with the pDoc
}
```

CloseDocument

```
HRESULT   CloseDocument
(
--- received arguments ---
   IPSWDoc  *pDoc
)
```

Closes the specified document.

Received arguments	
pDoc	The interface of the document to close

Specific Errors	
E_INVALID_ARG	The given pDoc could not be found

QueryPSWModuleInterface

```
HRESULT   QueryPSWModuleInterface
(
--- received arguments ---
   GUID *riid,
--- returned arguments ---
   IUnknown **ppUnknown
)
```

Queries if a module currently loaded in PS/Workshop supports a particular interface.

Received arguments	
riid	Reference identifier of the interface being requested
Returned arguments	
ppUnknown	Address of pointer which is filled if the query is successful

This function behaves in a similar way to IUnknown::QueryInterface. It allows one module to communicate with another module, so that modules can provide functionality to one another.

Note: It is a module's responsibility to ensure that the ppUnknown interface is correctly released.

RegisterFileOpenFunction

```
HRESULT   RegisterFileOpenFunction
(
--- received arguments ---
   IPSWAddIn *pAddIn,
   BSTR description,
   long CommandID
)
```

Registers a function to act as a FileOpen handler, and adds an entry to the PS/Workshop File Open dialog to handle a particular file extension.

Received arguments	
pAddIn	The IPSWAddIn interface of the module.
description	The type of files to associate with this function
CommandID	The ID associated with the function

If a registered filetype is subsequently opened from the PS/Workshop File Open dialog, the IPSWEvents::OnCommand function is called on the module with the `CommandID` of the registered function and a LPCSTR giving the full path name of the file to open.

The supplied `description` is a specially formatted string that contains information about the file types to associate with the registered function and the accompanying text that appears in the PS/Workshop File Open dialog.

The string must be formatted in the following way:

```
File Description to appear in the dialog |
File extensions to be associated with the function |
```

For example, Parasolid text XT files might have the following description:

```
"Parasolid Files (*.x_t;*.xmt_txt)|*.x_t;*.xmt_txt|"
```

Note: The CommandID argument should be unique within any given module.

For more information about registering handlers for filetypes, see Chapter 8, “Registering Handlers for Different Filetypes”.

Specific Errors	
PSW_ALREADY_REGISTERED	At least one of the given file types is already registered

RegisterFileSaveAsFunction

```
HRESULT   RegisterFileSaveAsFunction
(
--- received arguments ---
   IPSWAddIn *pAddIn,
   BSTR description,
   long CommandID
)
```

Registers a function to act as a FileSaveAs handler, and adds an entry to the PS/Workshop File Save As dialog to handle a particular file extension.

Received arguments	
`pAddIn`	The IPSWAddIn interface of the module.
`description`	The type of files to associate with this function
`CommandID`	The ID associated with the function

If the PS/Workshop File Save As dialog is subsequently used to save a registered filetype, the IPSWEvents::OnCommand function is called on the

module with the `CommandID` of the registered function and a LPCSTR giving the full path name of the file to save.

The supplied `description` is a specially formatted string that contains information about the file types to associate with the registered function and the accompanying text that appears in the PS/Workshop File Save As dialog.

The string must be formatted in the following way:

```
File Description to appear in the dialog |
File extensions to be associated with the function |
```

For example, JPEG files might have the following description:

```
"JPG Files (*.jpg;*.jpeg)|*.jpg;*.jpeg|"
```

Note: The CommandID argument should be unique within any given module.

For more information about registering handlers for filetypes, see Chapter 8, “Registering Handlers for Different Filetypes”.

Specific Errors	
PSW_ALREADY_REGISTERED	At least one of the given file types has already been registered

A.3 IPSWDoc

The IPSWDoc interface represents each of the PS/Workshop parts (documents) currently open. For each open document there exists a corresponding IPSWDoc interface pointer. Those functions called from IPSWDoc only affect the associated PS/Workshop document.

A.3.1 Summary

The following is a summary of the IPSWDoc interface.

Properties

Property	Type	Description
Colour	COLORREF	The colour of an entity
DocumentTitle	BSTR	The title of the associated document
DrawList	IPSWDrawList*	The IPSWDrawList associated with this document
PartList	IPSWParts*	The IPSWParts interface associated with this document
Rollback	IPSWRollback*	The IPSWRollback associated with this document
SelectionList	IPSWSelectionList*	The IPSWSelectionList associated with this document

Functions

Function	Description
SaveAsBMP	Saves the document as a Windows BMP
SaveAsXGL	Saves the document in RealityWave format
SaveAsWMF	Saves the document in Extended MetaFile Format
Update	Forces an update of the document in PS/Workshop

A.3.2 IPSWDoc Properties

Colour

```
HRESULT   get_Colour
(
--- received arguments ---
   PK_ENTITY_t pkEnt,
--- returned arguments ---
   COLORREF *pColour
)
```

Returns the colour of the given pkEnt.

Received arguments	
pkEnt	An entity
Returned arguments	
pColour	The colour of the entity

This works on the following classes of Parasolid entity:

- PK_CLASS_face, PK_CLASS_edge, PK_CLASS_vertex, PK_CLASS_body
- PK_CLASS_geom and all subclasses, if the entities are in a draw list

Attempting to return the colour of a geometric entity not in the draw list returns E_FAIL.

Note: If the entity is in a draw list then the colour returned may not be same as the colour the entity is currently displayed in.

Specific Errors	
PSWERR_NOTANENTITY	The given `pkEnt` is not a valid entity
PSWERR_INVALIDCLASS	The given `pkEnt` is not a valid class

```
HRESULT   put_Colour
(
--- received arguments ---
   PK_ENTITY_t pkEnt,
   COLORREF pColour
)
```

Sets the colour of the given `pkEnt`.

Received arguments	
`pkEnt`	An entity
Returned arguments	
`pColour`	The colour of the entity

This works on the following classes of Parasolid entity:

- PK_CLASS_face, PK_CLASS_edge, PK_CLASS_vertex, PK_CLASS_body
- PK_CLASS_geom and all subclasses, if the entities are in a draw list

Attempting to set the colour of a geometric entity not in the draw list returns E_FAIL.

Note: The entity is only drawn in the specified colour if it is not contained in a draw list.

Specific Errors	
PSWERR_NOTANENTITY	The given `pkEnt` is not a valid entity

DocumentTitle

```
HRESULT   get_DocumentTitle
(
--- returned arguments ---
   BSTR *pDocTitle
)
```

Returns the title of the current document.

Returned arguments	
`pDocTitle`	The title of the document

The title of the document appears in the title bar of the PS/Workshop document window, and is also used in the list of currently open documents in the **Window** menu, and the list of recently used files in the **File** menu.

Note: It is a module's responsibility to ensure that the `pDocTitle` resource is correctly freed.

```
HRESULT   put_DocumentTitle
(
--- received arguments ---
   BSTR DocTitle
)
```

Sets the title of the current document.

Received arguments	
`DocTitle`	The new title of the document

The title of the document appears in the title bar of the PS/Workshop document window, and is also used in the list of currently open documents in the **Window** menu, and the list of recently used files in the **File** menu.

DrawList

```
HRESULT   get_DrawList
(
--- returned arguments ---
   IPSWDrawList **ppDrawList
)
```

Returns the IPSWDrawList associated with the document.

Returned arguments	
ppDrawList	The draw list associated with the document

Note: It is a module's responsibility to ensure that the lifetime of `ppDrawList` is correctly handled.

PartList

```
HRESULT   get_PartList
(
--- returned arguments ---
   IPSWParts **ppPartList
)
```

Returns the IPSWParts associated with the document.

Returned arguments	
ppPartList	The part list associated with the document

Note: It is a module's responsibility to ensure that the lifetime of `ppPartList` is correctly handled.

Rollback

```
HRESULT   get_Rollback
(
--- returned arguments ---
   IPSWRollback **ppRollback
)
```

Returns the IPSWRollback interface associated with the document.

Returned arguments	
ppRollback	The IPSWRollback associated with the document

Note: It is a module's responsibility to ensure that the lifetime of ppRollback is correctly handled.

SelectionList

```
HRESULT   get_SelectionList
(
--- returned arguments ---
   IPSWSelectionList **ppSelectionList
)
```

Returns the IPSWSelectionList associated with the document

Returned arguments	
ppSelectionList	The selection list associated with the document

Note: It is a module's responsibility to ensure that the lifetime of ppSelectionList is correctly handled.

A.3.3 IPSWDoc Functions

SaveAsBMP

```
HRESULT   SaveAsBMP
(
--- received arguments ---
   LPCTSTR lpszPathName
)
```

Saves the current document in Windows bitmap format (*.bmp).

Received arguments	
lpszPathName	Name to save the file to

SaveAsXGL

```
HRESULT   SaveAsXGL
(
--- received arguments ---
   LPCTSTR lpszPathName
)
```

Saves the parts in the document in RealityWave format (`*.xgl`).

Received arguments	
`lpszPathName`	Name to save the file to

SaveAsWMF

```
HRESULT   SaveAsWMF
(
--- received arguments ---
   LPCTSTR lpszPathName
)
```

Saves the parts in the document in Enhanced Metafile format (`*.emf`).

Received arguments	
`lpszPathName`	Name to save the file to

Update

```
HRESULT   Update
(
--- received arguments ---
   BOOL render
)
```

Tells PS/Workshop that the parts in the document have changed and forces an update.

Received arguments	
`render`	Whether to force a redraw of the parts in the document

The `render` option controls whether the parts in the document should be completely redrawn, and is included for performance reasons. Setting `render` to FALSE allows you to improve the performance of an operation by ensuring that the document is not redrawn unnecessarily.

The final call in any sequence of calls to Update should pass `render` as TRUE to ensure that the display is correctly updated.

A.4 IPSWParts

This interface allows control over the number of Parasolid parts in a PS/Workshop document. You can use the functionality in this interface to add, remove, or interrogate parts in a document. It can be obtained from the IPSWDoc interface, and so only affects the associated document.

As an alternative to this interface, you can use direct calls to Parasolid. You might find it easier to use Parasolid functionality directly rather than use this interface. So long as parts have been created in a partition associated with a document, calling IPSWDoc::Update correctly handles the part list whenever necessary. For example, a part can be removed from the part list by calling PK_ENTITY_delete, and then calling Update.

A.4.1 Summary

The following is a summary of the IPSWParts interface.

Properties

Property	Type	Description
`_NewEnum`	`IUnknown*`	Returns the IPSWEnumParts interface
`Count`	`int`	The number of items in the part list
`Item`	`PK_PART_t`	An index into the part list

Functions

Function	Description
`AddItems`	Adds entities to the part list
`IsEmpty`	Tests for an empty part list condition
`IsMember`	Tests if an entity is a member of the list
`RemoveAll`	Removes all items from the list
`RemoveItems`	Removes a number of items from the list
`ReplaceItems`	Replace a number of items in the list

A.4.2 IPSWParts properties

_NewEnum

```
HRESULT   get__NewEnum
(
--- returned arguments ---
   IUnknown **ppunkEnum
)
```

Returns the IPSWEnumParts interface associated with the IPSWParts interface.

Returned arguments	
ppunkEnum	The returned IPSWEnumParts

This differs from get_Item in that get__NewEnum can be used to return more than one item at a time, so using this property in preference to get_Item could improve performance in some situations.

Note: It is a module's responsibility to manage the lifetime of ppunkEnum.

You can use the IPSWEnumParts interface to enumerate the parts in the IPSWParts interface. For more information see the definition of IPSWEnumParts.

The returned interface only represents a snapshot of the part list. It is not updated if the part list changes.

Count

```
HRESULT   get_Count
(
--- returned arguments ---
   int *pCount
)
```

Returns the number of entities in the part list.

Returned arguments	
pCount	The number of entities in the part list

Item

```
HRESULT   get_Item
(
---received arguments ---
   int index,
--- returned arguments ---
   PK_PART_t *pEnt
)
```

Returns the `pEnt` at position `index` in the part list.

Received arguments	
`index`	The index into the part list
Returned arguments	
`pEnt`	The entity at `index`

The part list is a zero based index system: `index` >= 0 and `index` < *n* - 1 where *n* is the number of entities in the list.

This can only be used to return a single item in the part list at a time. To return more than one item, use get__NewEnum.

Specific Errors	
E_INVALIDARG	`index` is not in range

A.4.3 IPSWParts functions

AddItems

```
HRESULT   AddItems
(
--- received arguments ---
   int nParts
   PK_PART_t *pkParts
)
```

Adds a list of parts to the part list.

Received arguments	
`nParts`	The number of parts
`pkParts`	The parts

IsEmpty

```
HRESULT   IsEmpty
(
--- returned arguments ---
   BOOL *Empty
)
```

Indicates whether the part list contains any elements.

Returned arguments	
Empty	Whether the list is empty or not

Empty is TRUE if the list is empty, FALSE otherwise.

IsMember

```
HRESULT   IsMember
(
--- received arguments ---
   PK_PART_t pkPart,
--- returned arguments ---
   BOOL *member
)
```

Indicates whether pkPart is a member of the part list.

Received arguments	
pkPart	The part to check
Returned arguments	
member	Whether it is a member

Member is TRUE if pkPart is a member of the list, FALSE otherwise.

RemoveAll

```
HRESULT   RemoveAll
(
)
```

Removes all the parts in the part list.

RemoveItems

```
HRESULT   RemoveItems
(
--- received arguments ---
   int nParts
   PK_PART_t *pkParts
)
```

Removes the specified parts from the part list.

Received arguments	
nParts	The number of parts
pkParts	The parts

ReplaceItems

```
HRESULT   ReplaceItems
(
--- received arguments ---
   int nNewPartCount
   PK_PART_t *nNewPartCount
)
```

Replaces all the parts in the list with the specified pNewParts.

Received arguments	
nNewPartCount	The number of parts
nNewPartCount	The parts

A.5 IPSWSelectionList

This interface allows control over the selection of entities in a PS/Workshop document. It can be obtained from the IPSWDoc interface, and so only affects the associated document.

A.5.1 Summary

The following is a summary of the IPSWSelectionList interface.

Properties

Property	Type	Description
_NewEnum	IUnknown*	Returns the IPSWEnumSelectionList interface
Colour	COLORREF	The colour of an item
Count	int	The number of items in the list
Item	PK_ENTITY_t	An item in the list

Functions

Function	Description
AddItems	Adds entities to the list
IsEmpty	Tests for an empty list condition
IsMember	Tests if an entity is a member of the list
RemoveAll	Removes all items from the list
RemoveItems	Removes a number of items from the list
ResetColour	Resets the selection colour
ResetColour2	Resets the selection colour
SetColour	Sets the selection colour
SetColour2	Sets the selection colour

A.5.2 IPSWSelectionList Properties

_NewEnum

```
HRESULT   get__NewEnum
(
--- received arguments ---
   PK_CLASS_t pkClass
--- returned arguments ---
   IUnknown **ppunkEnum
)
```

Returns the IPSWEnumSelectionList interface associated with the IPSWSelectionList.

Received arguments	
pkClass	The class of requested entities
Returned arguments	
ppunkEnum	The returned IPSWEnumSelectionList

This differs from get_Item in that get__NewEnum can be used to return more than one item at a time, so using this property in preference to get_Item could improve performance in some situations.

Note: It is a module's responsibility to manage the lifetime of `ppunkEnum`.

The IPSWEnumSelectionList interface can be used to enumerate the entities in the IPSWSelectionList interface. For more information see the definition of IPSWEnumSelectionList.

The returned interface only represents a snapshot of the selection list. It is not updated if the selection list changes.

The `pkClass` argument determines the type of entities in the selection list to enumerate through. This can have the following values:

Value	Description
PK_CLASS_null PK_CLASS_topol	Return all entities in the selection list.
PK_CLASS_body PK_CLASS_face PK_CLASS_edge PK_CLASS_vertex	Enumerate only those entities of the given type.

Specific Errors	
PSWERR_INVALIDCLASS	Invalid class type

Colour

```
HRESULT   get_Colour
(
--- received arguments ---
   PK_ENTITY_t pkEnt,
--- returned arguments ---
   COLORREF *pColour
)
```

Returns the selection colour of the given `pkEnt`.

Received arguments	
`pkEnt`	The entity

Returned arguments	
pColour	The selection colour of pkEnt

```
HRESULT   put_Colour
(
--- received arguments ---
   PK_ENTITY_t      pkEnt,       --- The entity
   COLORREF      Colour  --- The new selection colour for pkEnt
)
```

Sets the selection colour of pkEnt.

Received arguments	
Returned arguments	

Count

```
HRESULT   get_Count
(
--- received arguments ---
   PK_CLASS_t pkClass
--- returned arguments ---
   int *pCount
)
```

Returns the number of entities in the selection list.

Received arguments	
pkClass	The class of entities
Returned arguments	
pCount	The number of entities in the selection list

The pkClass argument determines the type of entities in the selection list to enumerate. This can have the following values:

Value	Description
PK_CLASS_null PK_CLASS_topol	All entities in the list
PK_CLASS_body PK_CLASS_face PK_CLASS_edge PK_CLASS_vertex	Only those entities of the given type

Specific Errors	
PSWERR_INVALIDCLASS	Invalid class type

Item

```
HRESULT   get_Item
(
---received arguments ---
   PK_CLASS_t pkClass
   int index,
--- returned arguments ---
   PK_PART_t *pEnt
)
```

Returns the `pEnt` at position `index` in the selection list.

Received arguments	
`pkClass`	Class of entity
`index`	The index into the selection list
Returned arguments	
`pEnt`	The entity at `index`

The selection list is a zero based index system: `index` >= 0 and `index` < *n* - 1 where *n* is the number of entities in the list.

This can only be used to return a single item in the selection list at a time. To return more than one item, use get__NewEnum.

Specific Errors	
E_INVALIDAR	`index` is not in range
PSWERR_INVALIDCLASS	`pkClass` is not a valid class

A.5.3 IPSWSelectionList Functions

AddItems

```
HRESULT   AddItems
(
--- received arguments ---
   int nEnts
   PK_ENTITY_t *pEnts
)
```

Adds a list of parts to the selection list.

Received arguments	
nEnts	The number of entities
pEnts	The entities

The class of pEnts can be one of the following:

- PK_CLASS_body
- PK_CLASS_edge
- PK_CLASS_vertex
- PK_CLASS_face

Specific Errors	
PSWERR_INVALIDCLASS	Class of one of the pEnts is invalid
PSWERR_NOTANENTITY	An item in pEnts is not an entity

IsEmpty

```
HRESULT   IsEmpty
(
--- returned arguments ---
   BOOL *Empty
)
```

Indicates whether the selection list contains any elements.

Returned arguments	
Empty	Whether the list is empty or not

Empty is TRUE if the list is empty, FALSE otherwise.

IsMember

```
HRESULT   IsMember
(
--- received arguments ---
   PK_ENTITY_t pkEnt,
--- returned arguments ---
   BOOL *member
)
```

Indicates whether `pkPart` is a member of the selection list.

Received arguments	
`pkEnt`	The entity to check
Returned arguments	
`member`	Whether it is a member

`Member` is TRUE if pkPart is a member of the list, FALSE otherwise.

Specific Errors	
PSWERR_INVALIDCLASS	Class of pkEnt is invalid
PSWERR_NOTANENTITY	pkEnt is not an entity

RemoveAll

```
HRESULT   RemoveAll
(
)
```

Removes all the entities in the selection list.

RemoveItems

```
HRESULT   RemoveItems
(
--- received arguments ---
   int nEnts
   PK_ENTITY_t *pkEnts
)
```

Removes the specified entities from the selection list.

Received arguments	
nEnts	The number of entities
Returned arguments	
pkEnts	The entities

Specific Errors	
PSWERR_INVALIDCLASS	Class of one of pkEnts is invalid
PSWERR_NOTANENTITY	One of pkEnts is not an entity

ResetColour

```
HRESULT   ResetColour
(
)
```

Resets the selection colour of all the entities in the document.

The default selection color is specified in the Default Colors tab of the Options dialog in PS/Workshop.

ResetColour2

```
HRESULT   ResetColour2
(
--- received arguments ---
   int nEnts,
   PK_ENTITY_t *pkEnts
)
```

Resets the selection colour for the specified entities.

Received arguments	
nEnts	The number of entities
pkEnts	The entities

The default selection color is specified in the Default Colors tab of the Options dialog in PS/Workshop.

SetColour

```
HRESULT   SetColour
(
--- received arguments ---
   COLORREF colour
)
```

Sets the selection colour for all entities in the selection list

Received arguments	
colour	The new selection colour

SetColour2

```
HRESULT   SetColour2
(
--- received arguments ---
   int nEnts,
   PK_ENTITY_t *pkEnts,
   COLORREF colour
)
```

Sets the selection colour for the given `pkEnts`.

Received arguments	
nEnts	The number of entities
pkEnts	The entities
colour	The new selection colour

A.6 IPSWDrawList

This interface allows control over custom drawing of entities in a PS/Workshop document. It can be obtained from the IPSWDoc interface, and so only affects the associated document.

A.6.1 Summary

The following is a summary of the IPSWDrawList interface.

Properties

Property	Type	Description
_NewEnum	IUnknown*	Returns the IPSWEnumSelectionList interface
Count	int	The number of items in the list
DrawOptions	IPSWDrawOpts*	The draw options for the list
Item	PK_ENTITY_t	An item in the list

Functions

Function	Description
AddItems	Adds entities to the list
AddItems2	Adds entities to the list
IsEmpty	Tests for an empty list condition
IsMember	Tests if an entity is a member of the list
ModifyItems	Changes the draw options
RemoveAll	Removes all items from the list
RemoveItems	Removes a number of items from the list
Update	Forces an update (re-render) for the given entities

A.6.2 IPSWDrawList properties

_NewEnum

```
HRESULT   get__NewEnum
(
--- returned arguments ---
   IUnknown **ppunkEnum
)
```

Returns the IPSWEnumDrawList interface associated with the IPSWDrawList interface.

Returned arguments	
ppunkEnum	The returned IPSWEnumDrawList

This differs from get_Item in that get__NewEnum can be used to return more than one item at a time, so using this property in preference to get_Item could improve performance in some situations.

Note: It is a module's responsibility to manage the lifetime of `ppunkEnum`.

Count

```
HRESULT   get_Count
(
--- returned arguments ---
   int *pCount
)
```

Returns the number of entities in the draw list.

Returned arguments	
`pCount`	The number of entities in the list

DrawOptions

```
HRESULT   get__DrawOptions
(
--- received arguments ---
   PK_ENTITY_t pkEnt,
--- returned arguments ---
   IPSWDrawOpts **ppDrawOpts
)
```

Returns the IPSWDrawOpts interface associated with the IPSWDrawList interface

Received arguments	
`pkEnt`	The entity to obtain the draw options
Returned arguments	
`ppDrawOpts`	The returned IPSWDrawOpts interface

Note: It is a module's responsibility to manage the lifetime of `ppDrawOpts`.

Item

```
HRESULT   get_Item
(
--- received arguments ---
   int index,
--- returned arguments ---
   PK_ENTITY_t *pEnt
)
```

Returns the `pEnt` at position `index` in the draw list.

Received arguments	
`index`	The position in the draw list of the item you require
Returned arguments	
`pEnt`	The returned item

The draw list is a zero based index system: `index` >= 0 and `index` < *n* - 1 where *n* is the number of entities in the list.

This can only be used to return a single item in the draw list at a time. To return more than one item, use get__NewEnum.

Specific Errors	
E_INVALIDARG	`index` is not in range

A.6.3 IPSWDrawList functions

AddItems

```
HRESULT   AddItems
(
--- received arguments ---
   int nEnts,
   PK_ENTITY_t *pkEnts
)
```

Add entities to the draw list using the default draw options

Received arguments	
`nEnts`	The number of entities
`pkEnts`	The entities to add to list

The following entity classes are allowed:

- PK_CLASS_body
- PK_CLASS_face
- PK_CLASS_edge
- PK_CLASS_vertex
- PK_CLASS_curve, and all subclasses
- PK_CLASS_surf, and all subclasses
- PK_CLASS_point

Any given entity must appear once and only once in the draw list. Attempting to add an item for the second time produces the error PSWERR_ITEMALREADYEXISTS

Specific Errors	
PSWERR_ITEMALREADYEXISTS	Item already exists in list
PSWERR_INVALIDCLASS	Class of one of `pkEnts` is invalid
PSWERR_NOTANENTITY	One of `pkEnts` is not a valid entity

AddItems2

```
HRESULT   AddItems2
(
--- received arguments ---
   int nEnts,
   PK_ENTITY_t *pkEnts,
   IPSWDrawOpts *pDrawOpts
)
```

Add Items to the draw list using the draw options specified by `pDrawOpts`.

Received arguments	
`nEnts`	The number of entities
`pkEnts`	The entities to add to list
`pDrawOpts`	Draw Options to apply to `pkEnts`

The following entity classes are allowed:

- PK_CLASS_body
- PK_CLASS_face
- PK_CLASS_edge
- PK_CLASS_vertex
- PK_CLASS_curve, and all subclasses
- PK_CLASS_surf, and all subclasses
- PK_CLASS_point

Any given entity must appear once and only once in the draw list. Attempting to add an item for the second time produces the error PSWERR_ITEMALREADYEXISTS

Specific Errors	
PSWERR_ITEMALREADYEXISTS	Item already exists in list
PSWERR_INVALIDCLASS	Class of one of `pkEnts` is invalid
PSWERR_NOTANENTITY	One of `pkEnts` is not a valid entity

IsEmpty

```
HRESULT   IsEmpty
(
--- returned arguments ---
   BOOL *Empty
)
```

Indicates whether the draw list contains any elements.

Returned arguments	
`Empty`	Whether the list is empty or not

`Empty` is TRUE if the list is empty, FALSE otherwise.

IsMember

```
HRESULT   IsMember
(
--- received arguments ---
   PK_ENTITY_t pkEnt,
--- returned arguments ---
   BOOL *member
)
```

Indicates whether pkEnt is a member of the list.

Received arguments	
`pkEnt`	The entity to check
Returned arguments	
`member`	Whether it is a member

`Member` is TRUE if pkPart is a member of the list, FALSE otherwise.

ModifyItems

```
HRESULT   ModifyItems
(
--- received arguments ---
   int nEnts,
   PK_ENTITY_t *pkEnts,
   IPSWDrawOpts *pDrawOpts
)
```

Modifies the draw options for the specified `pkEnts`.

Received arguments	
`nEnts`	The number of entities
`pkEnts`	The entities
`pDrawOpts`	The new draw options for the pkEnts

Specific Errors	
PSWERR_NOTINLIST	At least one of pkEnts is not in the list

RemoveAll

```
HRESULT   RemoveAll
(
)
```

Removes all the entities in the draw list.

RemoveItems

```
HRESULT   RemoveItems
(
--- received arguments ---
   int nEnts,
   PK_ENTITY_t *pkEnts
)
```

Removes the given entities from the draw list.

Received arguments	
`nEnts`	The number of entities
`pkEnts`	The entities

Specific Errors	
PSWERR_NOTINLIST	At least one of pkEnts is not in the selection list

Update

```
HRESULT   Update
(
--- received arguments ---
   int nEnts,
   PK_ENTITY_t *pkEnts
)
```

Forces an update (re-render) of the given entities in the draw list.

Received arguments	
nEnts	The number of entities
pkEnts	The entities

Specific Errors	
PSWERR_NOTINLIST	At least one of pkEnts is not in the draw list

A.7 IPSWDrawOpts

This interface can be obtained from IPSWDrawList (in which case it contains the current draw options). It provides control over how particular entities are displayed in the draw list.

IPSWDrawOpts is somewhat different from other interfaces, in that it can also be instantiated and passed to the IPSWDrawList interface (in which case it contains the new draw options). To instantiate this interface, you need to use the CoCreateInstance function, as described in Section 3.2, “Creating a COM object”.

A.7.1 Summary

The following is a summary of the IPSWDrawOpts interface.

Properties

Property	Type	Description
Clip	BOOL	Clip entity to part box (FALSE)
Colour	COLORREF	Colour to draw entity (-1)
DrawSense	BOOL	Whether to display sense of entity (FALSE)
N_U	int	Number of U param hatch lines (0)
N_V	int	Number of V param hatch lines (0)
Tolerance	BOOL	Whether to display tolerance of entity (FALSE)
ToleranceColour	COLORREF	Colour to draw tolerance in (-1)

Functions

Function	Description
Init	Initialise draw options
Reset	Reset draw options to their default values

A.7.2 IPSWDrawOpts Properties

Clip

```
HRESULT   get_Clip
(
--- returned arguments ---
   BOOL *pVal
)
```

Returns the clip option for the entity.

Returned arguments	
pVal	Clip status

This option is currently ignored.

```
HRESULT   put_Clip
(
--- received arguments ---
   BOOL newVal
)
```

Sets the clip option for the entity.

Received arguments	
newVal	New clip status

This option is currently ignored.

Colour

```
HRESULT   get_Colour
(
--- returned arguments ---
   COLORREF *pColour
)
```

Returns the draw colour for the entity.

Returned arguments	
pColour	The draw colour of entity

Specific Errors	
S_FALSE	The draw colour has not been set, in which case pColour is set to -1

```
HRESULT   put_Colour
(
--- received arguments ---
   COLORREF Colour
)
```

Sets the draw colour for the entity.

Received arguments	
Colour	The new draw colour for entity

Specifying Colour as -1 draws the entity in the default colour.

DrawSense

The DrawSense flag is only valid for entities of the following classes:

- PK_CLASS_face
- PK_CLASS_edge
- PK_CLASS_curve
- PK_CLASS_surf

If this flag is set for other class types it is ignored.

In the case of faces and surfaces, setting DrawSense to TRUE displays a face/surface normal in the same colour as the face/surface. For edges/curves an arrow is drawn depicting the direction going from the low to the high parameter.

```
HRESULT   get_DrawSense
(
--- returned arguments ---
   BOOL *pValue
)
```

Gets the draw sense option for the entity

Returned arguments	
pValue	Whether to draw the sense of the entity

```
HRESULT   put_DrawSense
(
--- received arguments ---
   BOOL newVal
)
```

Sets the draw sense for the entity

Received arguments	
newVal	The new draw sense for entity

N_U

The N_U flag is only valid for entities of class PK_CLASS_surf. It is ignored for all other entity classes.

```
HRESULT   get_N_U
(
--- returned arguments ---
   int *pVal
)
```

Returns the number of u parameter lines for the entity.

Returned arguments	
pVal	The number of U parameter lines

```
HRESULT   put_N_U
(
--- received arguments ---
   int newVal
)
```

Sets the number of U parameter lines for the entity.

Received arguments	
newVal	The new number of U parameter lines

Specific Errors	
E_INVALIDARG	The pVal argument < 0

N_V

The N_V flag is only valid for entities of class PK_CLASS_surf. It is ignored for all other entity classes.

```
HRESULT   get_N_V
(
--- returned arguments ---
   int *pVal
)
```

Returns the number of v parameter lines for the entity.

Returned arguments	
pVal	The number of V parameter lines

```
HRESULT   put_N_V
(
--- received arguments ---
   int newVal
)
```

Sets the number of V parameter lines for the entity.

Received arguments	
newVal	The new number of V parameter lines

Specific Errors	
E_INVALIDARG	The pVal argument < 0

Tolerance

This option is only valid for entities of class PK_CLASS_edge and class PK_CLASS_vertex. It is ignored for all other entity classes. In addition, this option only has a visible effect if the edge or vertex is tolerant.

- For tolerant edges, a tube is displayed around the edge with the same radius as the edge tolerance.
- For tolerant vertices, a sphere based on the point of the vertex is displayed with the same radius as the vertex tolerance.

```
HRESULT   get_Tolerance
(
--- returned arguments ---
   BOOL *pVal
---
)
```

Returns whether the tolerance of an entity should be displayed.

Returned arguments	
pVal	Whether to display the tolerance of an entity

```
HRESULT   put_Tolerance
(
--- received arguments ---
   BOOL newVal
)
```

Sets whether the tolerance of an entity should be displayed

Received arguments	
newVal	Whether to display the tolerance of an entity

ToleranceColour

This option is only used if the Tolerance option is set. As with Tolerance , this option is only valid for entities of class PK_CLASS_edge and PK_CLASS_vertex. It is ignored for all other entity classes.

If the ToleranceColour is not set then the tolerance is displayed in the same colour as the entity.

```
HRESULT   get_ToleranceColour
(
--- returned arguments ---
   COLORREF *pVal
)
```

Returns the colour in which to draw the tolerance of an entity.

Returned arguments	
pVal	The colour

This function is currently not implemented and returns E_NOTIMPL.

```
HRESULT   put_ToleranceColour
(
--- received arguments ---
   COLORREF newVal
)
```

Sets the colour in which to draw the tolerance of an entity.

Received arguments	
newVal	The colour

This function is currently not implemented and returns E_NOTIMPL.

A.7.3 IPSWDrawOpts Functions

Init

```
HRESULT   Init
(
--- received arguments ---
   int n_u,
   int n_v,
   COLORREF col,
   BOOL clipToPartBox,
   BOOL drawSense,
   BOOL tolerance
)
```

Allows a number of draw options to be set in one function call.

Received arguments	
n_u	N_U
n_v	N_V
col	Colour
clipToPartBox	Clip
drawSense	DrawSense
tolerance	Tolerance

Specific Errors	
E_INVALIDARG	Either n_u or n_v is < 0

Reset

```
HRESULT   Reset
(
)
```

Resets the draw options to their default values.

A.8 IPSWRollback

This interface provides access to Parasolid partitioned rollback functionality. It can be obtained from the IPSWDoc interface, and so only affects the associated document.

Note: You should use these wrapper functions rather than calling Parasolid PK_PMARK functions directly.

A.8.1 Summary

The following is a summary of the IPSWRollback interface.

Functions

Function	Description
`DeletePMark`	Deletes an existing pMark
`MakePMark`	Creates a new pMark
`RollbackTo`	Rollbacks to an existing pMark

A.8.2 IPSWRollback functions

DeletePMark

```
HRESULT   DeletePMark
(
--- received arguments ---
   PK_PMARK_t pMark
)
```

Deletes an existing `pMark`.

Received arguments	
`pMark`	The pMark

MakePMark

```
HRESULT   MakePMark
(
--- returned arguments ---
   PK_PMARK_t *pMark
)
```

Creates a new `pMark`.

Returned arguments	
`pMark`	The new pMark

RollbackTo

```
HRESULT   RollbackTo
(
--- received arguments ---
   PK_PMARK_t *pMark,
   BOOL del
)
```

Rollback the document to the specified pMark.

Received arguments	
pMark	The pMark to roll to
del	Whether to delete pMark

The del argument controls whether to delete pMark after rolling back to it. If del is TRUE then pMark is deleted, and also returned by the function as PK_ENTITY_null.

Note: This function can be used to either rollback or rollforward to a pMark.

A.9 IPSWEnumDrawList

Provides enumeration over the entities in the draw list

This interface can be obtained from the IPSWDrawList interface using the IPSWDrawList::_NewEnum property. Once obtained the interface can be used to enumerate over all the entities in the IPSWDrawList. Note that the _NewEnum property returns a snapshot of the IPSWDrawList and is not updated if the draw list is changed externally.

A.9.1 Summary

The following is a summary of the IPSWEnumDrawList interface.

Functions

Function	Description
Clone	Creates a copy of this enumeration
Next	Returns the next set of items in the enumeration
Reset	Resets the enumeration sequence to the beginning
Skip	Skips over the next specified number of elements in the enumeration

A.9.2 IPSWEnumDrawList functions

Clone

```
HRESULT   Clone
(
--- returned arguments ---
   IPSWEnumDrawList **ppenum
)
```

Creates another enumerator that contains the same enumeration state as the current one.

Returned arguments	
ppenum	The returned IPSWEnumDrawList

Note: It is a module's responsibility to ensure that the lifetime of ppenum is correctly handled

If the function succeeds then ppenum is a pointer to a new IPSWEnumDrawList. If the function fails then the value of ppenum is undefined.

Next

```
HRESULT   Next
(
--- received arguments ---
   ULONG celt,
--- returned arguments ---
   PK_ENTITY_t *rgelt,
   ULONG *pceltFetched
)
```

Retrieves the next celt items in the enumeration sequence.

Received arguments	
celt	The number of elements requested
Returned arguments	
rgelt	Returned entities
pceltFetched	The number of entities actually returned (may be NULL)

If there are fewer than the requested items left in the sequence, this function retrieves the remaining elements. The number of elements actually retrieved is returned in the `pceltFetched` argument.

Reset

```
HRESULT   Reset
(
)
```

Resets the enumeration sequence to the beginning.

Skip

```
HRESULT   Skip
(
--- received arguments ---
   ULONG celt
)
```

Skips over the next specified number of elements in the enumeration sequence.

Received arguments	
`celt`	The number of elements to skip

A.10 IPSWEnumParts

Provides enumeration over the entities in the part list

This interface can be obtained from the IPSWParts interface using the IPSWParts::_NewEnum property. Once obtained the interface can be used to enumerate over all the entities in the IPSWParts. Note that the _NewEnum property returns a snapshot of the IPSWParts and is not updated if the number of parts in the document are changed externally.

A.10.1 Summary

The following is a summary of the IPSWEnumParts interface.

Functions

Function	Description
`Clone`	Creates a copy of this enumeration
`Next`	Returns the next set of items in the enumeration

Function	Description
Reset	Resets the enumeration sequence to the beginning
Skip	Skips over the next specified number of elements in the enumeration

A.10.2 IPSWEnumParts functions

Clone

```
HRESULT   Clone
(
--- returned arguments ---
   IPSWEnumParts **ppenum
)
```

Creates another enumerator that contains the same enumeration state as the current one

Returned arguments	
ppenum	The returned IPSWEnumParts

Note: It is a module's responsibility to ensure that the lifetime of ppenum is correctly handled.

If the function succeeds then ppenum is a pointer to a new IPSWEnumParts. If it fails then the value of ppenum is undefined.

Next

```
HRESULT   Next
(
--- received arguments ---
   ULONG celt,
--- returned arguments ---
   PK_PART_t *rgelt,
   ULONG *pceltFetched
)
```

Retrieves the next celt items in the enumeration sequence.

Received arguments	
celt	The number of elements requested
Returned arguments	
rgelt	Returned entities
pceltFetched	The number of entities actually returned (may be NULL)

If there are fewer than the requested items left in the sequence, this function retrieves the remaining elements. The number of elements actually retrieved is returned in the pceltFetched argument.

Reset

```
HRESULT   Reset
(
)
```

Resets the enumeration sequence to the beginning.

Skip

```
HRESULT   Skip
(
--- received arguments ---
   ULONG celt
)
```

Skips over the next specified number of elements in the enumeration sequence.

Received arguments	
celt	The number of elements to skip

A.11 IPSWEnumSelectionList

Provides enumeration over the entities in the selection list.

This interface can be obtained from the IPSWSelectionList interface using the IPSWSelectionList::_NewEnum property. Once obtained the interface can be used to enumerate over all the entities in the IPSWSelectionList. It should be noted that the _NewEnum property returns a snapshot of the IPSWSelectionList and is not updated if the selection list is changed externally.

A.11.1 Summary

The following is a summary of the IPSWEnumSelectionList interface.

Functions

Function	Description
`Clone`	Creates a copy of this enumeration
`Next`	Returns the next set of items in the enumeration
`Reset`	Resets the enumeration sequence to the beginning
`Skip`	Skips over the next specified number of elements in the enumeration

A.11.2 IPSWEnumSelectionList functions

Clone

```
HRESULT   Clone
(
--- returned arguments ---
   IPSWEnumSelectionList **ppenum
)
```

Creates another enumerator that contains the same enumeration state as the current one.

Returned arguments	
`ppenum`	The returned IPSWEnumSelectionList

Note: It is a module's responsibility to ensure that the lifetime of `ppenum` is correctly handled

If the function succeeds then `ppenum` is a pointer to a new IPSWEnumSelectionList. If it fails then the value of `ppenum` is undefined.

Next

```
HRESULT   Next
(
--- received arguments ---
   ULONG celt,
--- returned arguments ---
   PK_ENTITY_t *rgelt,
   ULONG *pceltFetched
)
```

Retrieves the next `celt` items in the enumeration sequence.

Received arguments	
`celt`	The number of elements requested
Returned arguments	
`rgelt`	Returned entities
`pceltFetched`	The number of entities actually returned (may be NULL)

If there are fewer than the requested items left in the sequence, this function retrieves the remaining elements. The number of elements actually retrieved is returned in the `pceltFetched` argument.

Reset

```
HRESULT   Reset
(
)
```

Resets the enumeration sequence to the beginning.

Skip

```
HRESULT   Skip
(
--- received arguments ---
   ULONG celt
)
```

Skips over the next specified number of elements in the enumeration sequence.

Received arguments	
`celt`	The number of elements to skip

A.12 IPSWView

This interface is superseded by IPSWView2.

A.13 IPSWView2

Each document open in PS/Workshop has one or more views attached. Each IPSWView is associated with an IPSWDoc.

> **Note:** In the current version of PS/Workshop there is only one IPSWView associated with each IPSWDoc.

A.13.1 Summary

The following is a summary of the IPSWView interface.

Properties

Property	Type	Description
`ViewMatrix`	`PK_TRANSF_t`	The view matrix
`CurrentOperation`	`pswCurrentOperation`	The current operation
`ViewStyle`	`pswViewStyle`	The current view style
`SelectionFilter`	`pswSelectionFilter`	The current selection filter
`RenderFacetOpts`	`void*`	The facet rendering options
`RenderLineOpts`	`void*`	The line rendering options
`ViewCentre`	`double*`	The current view centre

Functions

Function	Description
`FitToScreen`	Resizes the view to fit the part
`ResetRenderOptions`	Resets the render options to their default
`ScaleView`	Scales the view by a given factor
`ZoomToEntities`	Zooms the view to the given entities
`RotateView`	Rotates the view about a given axis
`Update`	Forces an updated (re-render) of the given entities

A.13.2 IPSWView2 properties

ViewMatrix

```
HRESULT   get_ViewMatrix
(
--- returned arguments ---
   PK_TRANSF_t *pVal
)
```

Returns the current view matrix.

Returned arguments	
pVal	The view matrix

```
HRESULT   put_ViewMatrix
(
--- received arguments ---
   PK_TRANSF_t newVal
)
```

Sets the current view matrix.

Received arguments	
newVal	The new view matrix

CurrentOperation

```
HRESULT   get_CurrentOperation
(
--- returned arguments ---
   pswCurrentOperation *pVal
)
```

Returns the current PS/Workshop operation.

Returned arguments	
pVal	The current operation

pVal may be one of the following values:

- pswCurrentOperationIdle
- pswCurrentOperationRotate
- pswCurrentOperationZoom
- pswCurrentOperationZoomWindow
- pswCurrentOperationPan

```
HRESULT   put_CurrentOperation
(
--- received arguments ---
   pswCurrentOperation newVal
)
```

Sets the current PS/Workshop operation.

Received arguments	
newVal	The new operation

newVal may be one of the following values:

- pswCurrentOperationIdle
- pswCurrentOperationRotate
- pswCurrentOperationZoom
- pswCurrentOperationZoomWindow
- pswCurrentOperationPan

ViewStyle

```
HRESULT   get_ViewStyle
(
--- returned arguments ---
   pswViewStyle *pVal
)
```

Returns the current view style.

Returned arguments	
pVal	The view style

pVal may be one of:

- pswViewStyleShaded
- pswViewStyleWireframe
- pswViewStyleWireAndSils
- pswViewStyleShadedWireAndSils
- pswViewStyleHidden
- pswViewStyleShadedWireframe

```
HRESULT   put_ViewStyle
(
--- received arguments ---
   pswViewStyle newVal
)
```

Sets the current view style.

Received arguments	
newVal	The view style

newVal may be one of:

- pswViewStyleShaded
- pswViewStyleWireframe
- pswViewStyleWireAndSils
- pswViewStyleShadedWireAndSils
- pswViewStyleHidden
- pswViewStyleShadedWireframe

SelectionFilter

```
HRESULT   get_SelectionFilter
(
--- returned arguments ---
   pswSelectionFilter *pVal
)
```

Returns the current selection filter.

Returned arguments	
pVal	The selection filter

pVal may be any combination of the following flags:

- pswSelectionFilter_None
- pswSelectionFilter_Edge
- pswSelectionFilter_Face
- pswSelectionFilter_Vertex
- pswSelectionFilter_Body
- pswSelectionFilter_All

```
HRESULT   put_SelectionFilter
(
--- received arguments ---
   pswSelectionFilter newVal
)
```

Sets the current selection filter.

Received arguments	
newVal	The selection filter

newVal may be any combination of the following flags:

- pswSelectionFilter_None
- pswSelectionFilter_Edge
- pswSelectionFilter_Face
- pswSelectionFilter_Vertex
- pswSelectionFilter_Body
- pswSelectionFilter_All

RenderFacetOpts

```
HRESULT   get_RenderFacetOpts
(
--- returned arguments ---
   void *pVal
)
```

Returns the current facetting options.

Returned arguments	
pVal	The facetting options

Note: pVal must to be cast to a PK_TOPOL_facet_mesh_o_t* structure.

```
HRESULT   put_RenderFacetOpts
(
--- returned arguments ---
   void *newVal
)
```

Sets the current facetting options.

Returned arguments	
newVal	The facetting options

Note: newVal must be a PK_TOPOL_facet_mesh_o_t* structure.

RenderLineOpts

```
HRESULT   get_RenderLineOpts
(
--- returned arguments ---
   void *pVal
)
```

Returns the current line rendering options.

Returned arguments	
pVal	The line rendering options

Note: pVal must be cast to a PK_TOPOL_render_line_o_t* structure.

```
HRESULT   put_RenderLineOpts
(
--- returned arguments ---
   void *newVal
)
```

Sets the current line rendering options.

Returned arguments	
newVal	The line rendering options

Note: newVal must be a PK_TOPOL_render_line_o_t* structure.

ViewCentre

```
HRESULT   get_ViewCentre
(
--- returned arguments ---
   double pVal[3]
)
```

Returns the current view centre

Returned arguments	
pVal	The view centre

pVal is the current view centre where:

- pVal[0] represents the x component
- pVal[1] represents the y component
- pVal[2] represents the z component

```
HRESULT   put_ViewCentre
(
--- received arguments ---
   double newVal[3]
)
```

Sets the current view centre

Received arguments	
newVal	The view centre

newVal is the current view centre where:

- newVal[0] represents the x component
- newVal[1] represents the y component
- newVal[2] represents the z component

A.13.3 IPSWView2 functions

FitToScreen

```
HRESULT   FitToScreen
(
)
```

Recalculates the view to fit all the parts in the document on the screen.

ResetRenderOptions

```
HRESULT   ResetRenderOptions
(
)
```

Resets the render options to their default values.

This function returns full control of the rendering to PS/Workshop.

ScaleView

```
HRESULT   ScaleView
(
--- received arguments ---
   double factor
)
```

Scales the view by the given `factor`.

Received arguments	
`factor`	Scale to apply to view

If `factor` < 1 then the view zooms out. If `factor` is > 1 then the view zooms in.

Specific Errors	
E_INVALIDARG	factor must be > 0

ZoomToEntities

```
HRESULT   ZoomToEntities
(
---received arguments ---
   int nEnts,
   PK_ENTITY_t *pkEnts
)
```

Zooms to the given entities.

Received arguments	
nEnts	The number of entities
pkEnts	The entities to zoom to

The following classes of entities are supported:

- PK_CLASS_assembly
- PK_CLASS_body
- PK_CLASS_face
- PK_CLASS_edge
- PK_CLASS_vertex
- PK_CLASS_point
- PK_CLASS_curve, and all subclasses
- PK_CLASS_surf, and all subclasses

Specific Errors	
PSWERR_INVALIDCLASS	At least one of pkEnts is of the wrong class
PSWERR_NOTANENTITY	At least one of pkEnts is not a valid entity

RotateView

```
HRESULT   RotateView
(
--- received arguments ---
   double angle,
   double *axis
)
```

Rotates the view about the given axis.

Received arguments	
`angle`	Angle to rotate view in radians
`axis`	Axis to rotate view about

- `axis[0]` represents the x component of axis
- `axis[1]` represents the y component of axis
- `axis[2]` represents the z component of axis

Specific Errors	
E_INVALIDARG	axis is not a unit vector

Update

```
HRESULT   Update
(
--- received arguments ---
   int nEnts,
   PK_ENTITY_t *pkEnts
)
```

Forces an update (re-render) of the given entities.

Received arguments	
`nEnts`	The number of entities
`pkEnts`	The entities

This function updates the folllowing Parasolid entity classes:

- PK_CLASS_body
- PK_CLASS_assembly
- PK_CLASS_face
- PK_CLASS_edge
- PK_CLASS_vertex
- PK_CLASS_curve, and all subclasses (if in the draw list)
- PK_CLASS_surf, and all subclasses (if in the draw list)
- PK_CLASS_point (if in the draw list)

A.14 IPSWAddIn

Provides the connection/disconnection mechanism for a module. Every PS/Workshop module must support this interface.

A.14.1 Summary

The following is a summary of the IPSWAddIn interface.

Functions

Function	Description
OnConnection	Called when a module is first loaded
OnDisconnection	Called when a module is unloaded

A.14.2 IPSWAddIn functions

OnConnection

```
HRESULT   OnConnection
(
--- received arguments ---
   IDispatch *pIPSWApp,
   IDispatch *pIPSWAddIn,
   PswConnectMode eConnectMode
)
```

This function is called whenever a module is loaded.

Received arguments	
pIPSWApp	The PS/Workshop IDispatch interface
pIPSWAddIn	The PS/Workshop IPSWAddIn interface
eConnectMode	How the module is loaded

The eConnectMode argument determines how the module was loaded. It can have one of the following values:

Value	Description
pswConnectAtStartUp	The module has been loaded at startup
pswConnectByUser	A user has requested this module to be loaded
pswConnectExternally	The module has been loaded from an external source

PS/Workshop calls this function on a module when it attempts to load it. If this returns an error code then PS/Workshop unloads the module. In the default implementation this calls CAddonMain:StartModule.

OnDisconnection

```
HRESULT   OnDisconnection
(
--- received arguments ---
   pswDisconnectMode DisconnectMode
)
```

This function is called when the module is disconnected.

Received arguments	
DisconnectMode	How the module came to be unloaded

The pswDisconnectMode argument determines how the module came to be unloaded. It may have the following values:

Value	Description
pswDisconnectAtShutdown	The module is being disconnected as a result of PS/Workshop shutting down
pswDisconnectByUser	The module has been unloaded at the request of the user
pswDisconnectExternally	The module has been unloaded at the request of some external process

This function on a module is called immediately before the module is unloaded. It is suggested that during this function the module ensures that all references to PS/Workshop interfaces are released.

A.15 IPSWEvents

Provides notification of PS/Workshop events to a module

A.15.1 Summary

The following is a summary of the IPSWEvents interface.

Functions

Function	Descriptoin
OnCommand	Called whenever a menu item added by a module is selected
OnCommandHelp	Called to provide help on a menu item added by a module

Function	Descriptoin
`OnCommandUpdateUI`	Called to update the UI of a menu item added by a module
`OnDocClose`	Called whenever a PS/Workshop document is closed
`OnDocOpen`	Called whenever a PS/Workshop document is opened
`OnPartChange`	Called whenever the parts in a document have changed
`OnSelectTopols`	Called whenever a selection event occurs
`OnViewClose`	Called whenever a view is closed
`OnViewOpen`	Called whenever a view is opened

A.15.2 IPSWEvents functions

OnCommand

```
HRESULT   OnCommand
(
--- received arguments ---
   long CommandID,
   void *lparam
)
```

Called whenever a menu item added by a module is selected, or a file of a registered filetype is chosen.

Received arguments	
`CommandID`	The ID associated with the command
`lparam`	Extra information relating to the command

This event may occur in the following cases:

Case	Comments
The user chose a menu item previously added by the module.	In this case, `CommandID` is the ID previously passed to IPSWApp::AddMenuItem or AddMenuItem2. The `lparam` argument is NULL in this instance.
The user has selected a file of a registered filetype from either the File Open or File Save As dialog.	In this case, `CommandID` is the ID of the registered FileOpen or FileSaveAs handler. The `lparam` argument contains the filename to open or save, as a LPCSTR. The module must have already registered a custom FileOpen or FileSaveAs handler using IPSWApp::RegisterFileOpenFunction or RegisterFileSaveAsFunction for this case to be handled.

OnCommandHelp

```
HRESULT   OnCommandHelp
(
--- received arguments ---
   long hFrameWnd,
   long HelpCommandID,
   long CommandID
)
```

Called whenever help is requested about a menu item added by a module.

This function is currently not implemented and returns E_NOTIMPL.

OnCommandUpdateUI

```
HRESULT   OnCommandUpdateUI
(
--- received arguments ---
   long CommandID,
--- returned arguments ---
   long *CommandFlags,
   BSTR *MenuItemText,
   long *BitmapID
)
```

Called to update the user interface for a menu item added by a module.

Received arguments	
`CommandID`	The ID associated with the command
Returned arguments	
`CommandFlags`	Whether to enable/disable this command
`MenuItemText`	The text to associate with this menu entry
`BitmapID`	Not used

This function is called by the framework to update the status of user added menu items. `CommandID` is the ID previously passed to IPSWApp::AddMenuItem or AddMenuItem2 for a particular menu item.

You can use `CommandFlags` to change the status of a menu item. It can take any of the following values:

Value	Description
pswCmdUI_Enable	Enable the menu item if it is currently disabled.
pswCmdUI_Disable	Disable the menu item if it is currently enabled.
pswCmdUI_Checked	Place a tick mark next to the menu item.
pswCmdUI_Unchecked	Remove the tick mark next to the menu item.
pswCmdUI_ChangeText	Use the text specified in `MenuItemText` as the text for the menu item.
pswCmdUI_UseBmp	Use the bitmap specified in `BitmapID` as the bitmap associated with the `CommandID`.

The `MenuItemText` argument controls the text that is associated with the menu item, so that the text seen by the user can be changed on the fly, in conjunction with the value of `CommandFlags`.

Note: The BitmapID argument is currently ignored.

OnDocClose

```
HRESULT   OnDocClose
(
--- received arguments ---
   IPSWDoc* pDoc
)
```

Called whenever a PS/Workshop document is closed.

Received arguments	
pDoc	The IPSWDoc interface associated with this document

This event only occurs after the associated OnViewClose event.

OnDocOpen

```
HRESULT   OnDocOpen
(
--- received arguments ---
   IPSWDoc* pDoc
)
```

Called whenever a PS/Workshop document is opened.

Received arguments	
pDoc	The IPSWDoc interface associated with this document

This event occurs before the associated OnViewOpen event.

OnPartChange

```
HRESULT   OnPartChange
(
)
```

Called whenever the parts in a document change.

This function is called whenever the number of parts in any document change, or the parts themselves change. It is not called if any sub-entities of the part are modified.

OnSelectTopols

```
HRESULT   OnSelectTopols
(
)
```

Called whenever a selection event occurs inside PS/Workshop.

This function is called whenever an entity is selected or deselected in any document. The selected entities can then be found using the IPSWSelectionList interface.

OnViewClose

```
HRESULT   OnViewClose
(
--- received arguments ---
   IPSWDoc *pDoc,
   IPSWView *pView
)
```

Called whenever the PS/Workshop view is closed.

Received arguments	
pDoc	The IPSWDoc interface associated with this view
pView	The IPSWView interface associated with this view

This function is called before any subsequent call to OnDocClose.

As of PS/Workshop 2.1, pView can be query-interfaced to IPSWView.

OnViewOpen

```
HRESULT   OnViewOpen
(
--- received arguments ---
   IPSWDoc *pDoc,
   IPSWView *pView
)
```

Called whenever the PS/Workshop view is opened.

Received arguments	
pDoc	The IPSWDoc interface associated with this view
pView	The IPSWView interface associated with this view

This function is called after any call to OnDocOpen.

Known Issues B

This chapter lists known issues with the current version of PS/Workshop.

Modules written for previous versions of PS/Workshop and linked to `edsps111.lib` will need to be recompiled and linked to `pskernel.lib` to load successfully

Developers of modules for earlier versions of PS/Workshop should note that the structure of the new COM template module has been kept as close as possible to the original template module.

The default module created by the PS/Workshop AppWizard uses the delayload mechanism when it is built. Doing this means that you can successfully build the module without having `pskernel.dll` present in the same directory as the module DLL when the module is first created. It does, however, cause a warning to be generated when the module is compiled for the first time.