OLE Document Objects Specifications

Version 1.0, Also Known as "DocObjects"

OLE Design Team and Office Design Team
30 April, 1996

Distribution: Public

Contents

Introduction

The software industry has entered an era in which customers rely on many products to complete their work. For example, when a new company creates its business plan, it may rely on Microsoft Word to create the basic proposal, Excel to create a summary of projected financial performance, and PowerPoint to create a slide show for potential investors. Although customers rely on several distinct products to complete their projects, they think of each project as a single entity: a business plan, a sales proposal, a book, and so forth.

Office for Windows 95 introduced a new application called the Binder, that makes it easier for customers to complete projects that contain heterogeneous documents (that is, a variety of documents created by many distinct applications). And it makes it easy for them to use the standard Office applications as they do so. It may be helpful to think of the Binder as a an electronic paper clip: it holds together text files, spreadsheets, graphics presentations and other documents so that the user can manipulate them as a single entity. From another perspective, the Binder is a sophisticated “viewer” that can host a variety of heterogeneous documents which let the user create, edit, save, print, and view distinctly different kinds of information.

“Document Objects” (DocObjects for short) is the core technology that makes Office Binder work. While originally developed as a proprietary technology for Microsoft Office, Microsoft believe that DocObjects represents an important step forward and that is will benefit customers in many more ways than just Office and Office-Compatible applications. Most notably is that the technology is flexible enough to support document containers other than Office Binder, and can support document servers other than just Office and Office Compatible applications.

One obvious application for this technology is within the domain of “Internet browsers”, where the adoption of Binder technology will not only facilitate the presentation of Internet-based information (Web pages and so forth) but will, at the same time with the same implementation, enable the browser to present documents from Office and Office Compatible applications. In short, the user need only go to one navigation tool to browse and view all documents whether local or network-based.

This specification explains the Document Objects architecture in terms of the container and the server side of the technology.

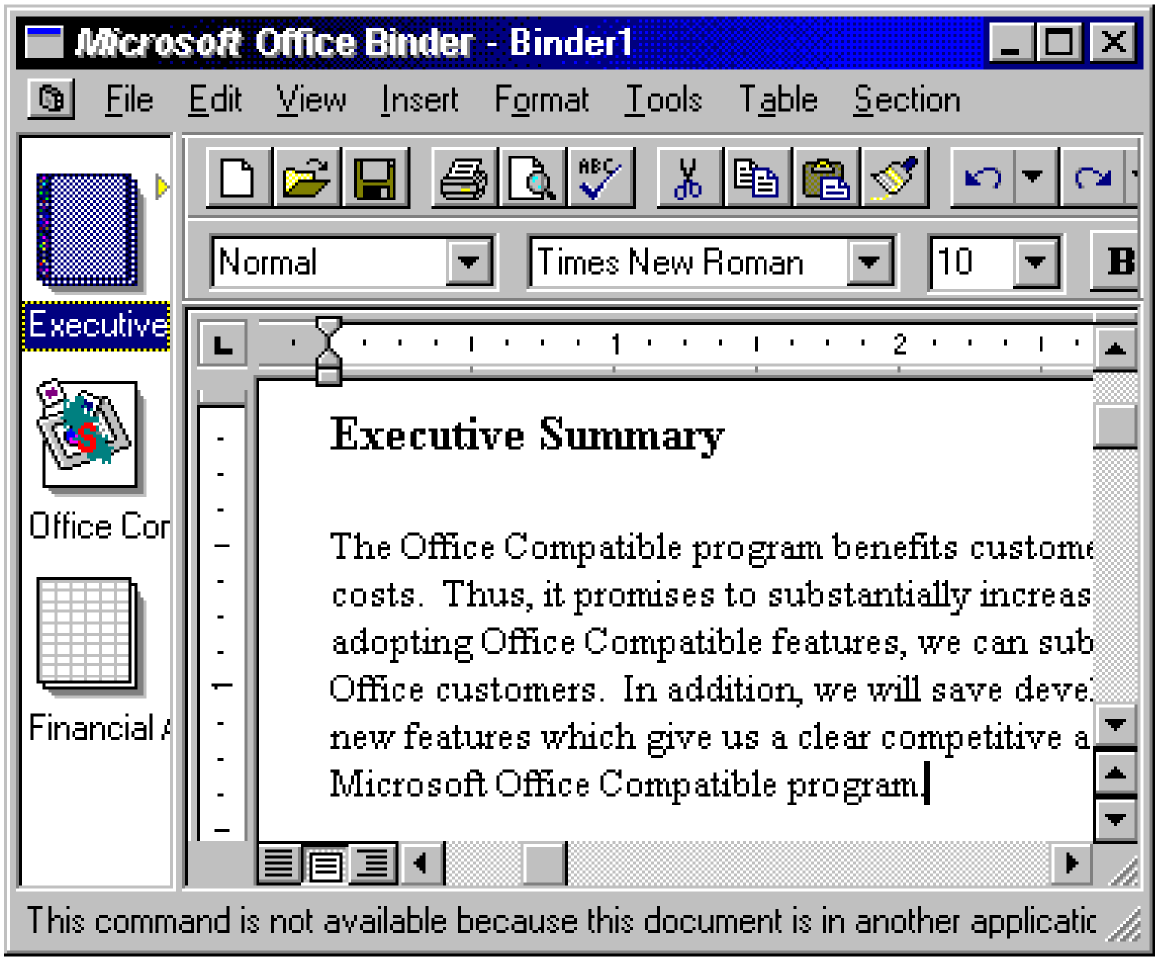
Note: This specification has been written from the perspective of an advanced OLE programmer. For additional information about OLE issues discussed below, please consult the OLE Programmer’s Reference and related publications.

The current Internet Explorer does not implement all features described in this specification. It’s features are as follows:

1. *Internet Explorer only supports one document view as a container. It also views only one document at a time (per instance of IE).*
2. *Internet Explorer makes no use of IPrint.*
3. *Internet Explorer does route a number of commands through IOleCommandTarget.*

Feature Description

The following picture illustrates the Office 95 Binder, which for the purposes of this document serves merely as an illustration of a “document container.”



Executive Summary

The Office Compatible program benefits custome
costs. Thus, it promises to substantially increas
adopting Office Compatible features, we can sub
Office customers. In addition, we will save deve
new features which give us a clear competitive a
Microsoft Office Compatible program.

This command is not available because this document is in another applicatic

As this illustration shows, the Binder includes two primary panes, as will most containers. The left pane shows thumbnails that correspond to the section of the Binder. For example, the preceding Binder contains a Word document, a PowerPoint slide show, and an Excel spreadsheet. Users can click these images to activate the corresponding “document object” or “DocObject.” The right pane of the Binder shows the document on which the user is currently working (a Word document in the preceding illustration). In an “Explorer” type of container, the left pane would display a hierarchical tree list of a drive or network while the right pane would display the available document or page in that point of the hierarchy.

When a document is “activated” in the right hand pane, it looks and acts, for all intents and purposes, as if the user was running the stand-alone application that normally manages that particular document type, complete with toolbars, menus, and all other user interface elements. A container like Office Binder thus provides a single frame in which to work with documents, instead of forcing the user to multiple application frames for each document type. (This is also different than working with embeddings in a compound document where only a single piece of content is being activated; here we are activating an entire document, that is, an entire application, within the context of a single frame).

While the Document Objects technology allows an application like Office Binder to present literally a “three-ring binder” paradigm, DocObjects is generic enough to accommodate many other possible user interfaces that have the same requirements.

A General Overview of Document Objects

The DocObjects technology is a set of extensions to OLE Documents, the compound document technology of OLE. The extensions are in the form of additional interfaces that allow what mostly looks like an embeddable in-place object to represent an entire document instead of a single piece of embedded content. As with OLE Documents, DocObjects involve a container that provides the display space for DocObjects and servers that provide the user interface and manipulation capabilities for DocObjects themselves.

A DocObject server is a product that supports one or more document object classes, where each object itself supports the extension interfaces that allow the object to be activated in a suitable container, such as Binder. A DocObject is best understood by distinguishing it from a standard OLE embedded object. Following the OLE convention, an embedded object is one which is displayed within the "page" of the document that "owns it" where the document is managed by an OLE container. The container stores the embedded object's data with the rest of the document.

However, embedded objects are limited in that they do not control the page on which they appear. By necessity they tend to be rather small objects: a picture that supplements the surrounding text (provided by the container), a spreadsheet that clarifies its supporting analysis (again provided by the container), and so forth.

By contrast, a document object provided from a DocObject server is essentially a full-scale, conventional document which is embedded as an object within another DocObject container (Binder, browsers, etc.). Unlike embedded objects, DocObjects have complete control over their pages, and the full power of the application is available to the user to edit them. Thus, unlike embedded objects, DocObjects tend to be full-scale, robust documents that exploit the complete native functionality of the server (application) that creates them. Users can create documents (called sections within the Binder, for example) using the full horsepower of their favorite applications (if they are DocObject enabled), yet they can treat the resulting project as a single entity, which can be uniquely named, saved, transmitted to coworkers for review or editing, printed as a single entity, and so forth. In the same way, a user of an Internet browser (such as a future Explorer) can treat the entire network as well as local file systems as a single document-storage entity with the ability to browse the documents in that storage from a single location.

Summary of Requirements for Document Object Participation

A DocObject container that wishes to integrate DocObjects must:

1. Be capable of handling object storage through the *IPersistStorage* interface, that is, it must provide an *IStorage* instance to each DocObject.
2. Support the basic embedding features of OLE Documents, necessitating "site" objects (one per document or embedding) that implements *IOleClientSite* and *IAdviseSink*.
3. Support in-place activation of embedded objects or DocObjects, requiring the container's site objects to implement *IOleInPlaceSite* and requiring the container's frame object to provide *IOleInPlaceFrame*.
4. Support the DocObjects extensions through the implementation of *IOleDocumentSite* and possibly IContinueCallback on the site object, along with *IOleCommandTarget* on the frame object.

(Note that OLE Documents support in a container implies more than just interface implementations: it also requires knowledge of using the interfaces of an embedded object. Same applies to DocObjects extensions where the container must also know how to use those extension interfaces on the DocObjects themselves.)

Correspondingly, a "document object" that wishes to work within a DocObject container must:

1. Use OLE's Compound Files as their storage mechanism, that is, implement *IPersistStorage*.
2. Support the basic embedding features of OLE Documents, including "Create From File." This necessitates the interfaces *IPersistFile*, *IOleObject*, and *IDataObject*.
3. Support the in-place activation extension of OLE Documents, that is, *IOleInPlaceObject* and *IOleInPlaceActiveObject* (using the container's *IOleInPlaceSite* and *IOleInPlaceFrame* interfaces).
4. Support the DocObjects extensions that involves these new interfaces: *IOleDocument*, *IOleDocumentView*, *IOleCommandTarget*, and *IPrint*.

Again, knowledge of when and how to use the container-side interfaces is implied in these requirements.

The remainder of this document will describe the architecture of Document Objects and how the new interfaces of *IOleDocument*, *IEnumOleDocumentViews, IOleDocumentView*, *IOleDocumentSite*, *IPrint*, *IContinueCallback,* and *IOleCommandTarget* work together to achieve the document-activation features described earlier. First will be the architectural details of Document Objects including special requirements for Help menu merging (an extension to OLE Documents), programmatic printing, and "command targets." This is followed by implementation notes for DocObject containers and servers, followed by the complete interface reference and an appendix describing additional details of the Office Binder's implementation this architecture.

Architectural Details of Document Objects

Overview

In general, a particular product owns a set of data, the storage in which they are saved, and the views through which they are displayed for the user. OLE Documents introduced technology that let applications store their documents (including data and information about the way they should be displayed) in an abstract manner. That is, the application was freed from the need to understand the granular aspects of its storage vehicle, and it could, instead, deal with a variety of storage sites in a consistent fashion (that is, without regard for their underlying properties). Products that exploited this OLE "abstract storage technology", could save their documents into files, databases, and other types of storage in a uniform way. In addition, OLE enabled applications (via *IPersist** interfaces) to permanently save their documents in storage that they did not own. Thus, applications that support OLE can treat storage abstractly.

The OLE Documents architecture also took an important step toward letting applications treat their views (that is, the port through which their data are displayed for the user) abstractly. Products that supported in-place activation could display their content in a "foreign" frame that they did not own. While this represented an important step forward, it had its limitations. In particular, in-place activation supports an object view of the data rather than a document view. That is, the OLE container is responsible for siting the object's view port, for controlling the overall display of its pages, for printing them, and so forth. To overcome some of these limitations, the embedded object can be opened so that more of the native functionality of its parent (that is, the OLE server) can be used. Nevertheless, some important limitations remained.

Among these limitations, it is worth noting that a defining characteristic of a document, as distinguished from an embedded object, is that it owns the printed page. It can have headers, footers, footnotes, endnotes, revision marks, and so forth, and it knows where to place them on the page and how to display them for the user. Embedded objects (that is, object views) do not control the page on which they appear. Instead, they must live within a containment hierarchy whose root container is an actual document.

The Document Objects architecture defines an abstraction for *views* and their management, so that objects can function within containers and yet retain control over their display and some important printing functions. This architecture makes it possible to display documents both in foreign frames (such as Binder or Explorer) and in native frames (such as the product's own view ports).

A view can be divided into two components: the view frame component and the view component. The view frame could consist of just the frame window in the case of an Single Document Interface (SDI) product, or it could include the frame and the Multiple Document Interface (MDI) window in the case of an MDI product. The view frame component provides the space for menus, toolbars, a status bar, and so forth. The frame component also provides the view port within which adornments such as rulers, scroll bars, and similar tools can be displayed. Note that the frame does not "tell the document" how big it should be, nor does it care. On the contrary, it merely conveys to the contained application the view port size that was selected by the user. The view port, by contrast, is the region within which the data themselves are displayed. If the frame were an SDI frame, for example, then the view port could be the client area of the frame window minus the space allocated for tool bars, status bar, and such. In an MDI setting, the view port would be the client area of the MDI document window minus any other frame level user interface elements (for example, space for tab bar in case of workbook).

This breakdown of components offers several advantages:

- The view frame can be of any type: SDI, MDI, Workbook, Form, and so forth. A single implementation of the document's view can be used within many different types of frames.
- Multiple applications can have the same frame level UI and functionality, while supporting distinctly different data sets. In principle, this offers a great advantage to vendors who wish to develop a core user interface as a frame that is used throughout their entire product line. In essence, they can build the frame once and re-use it as appropriate.

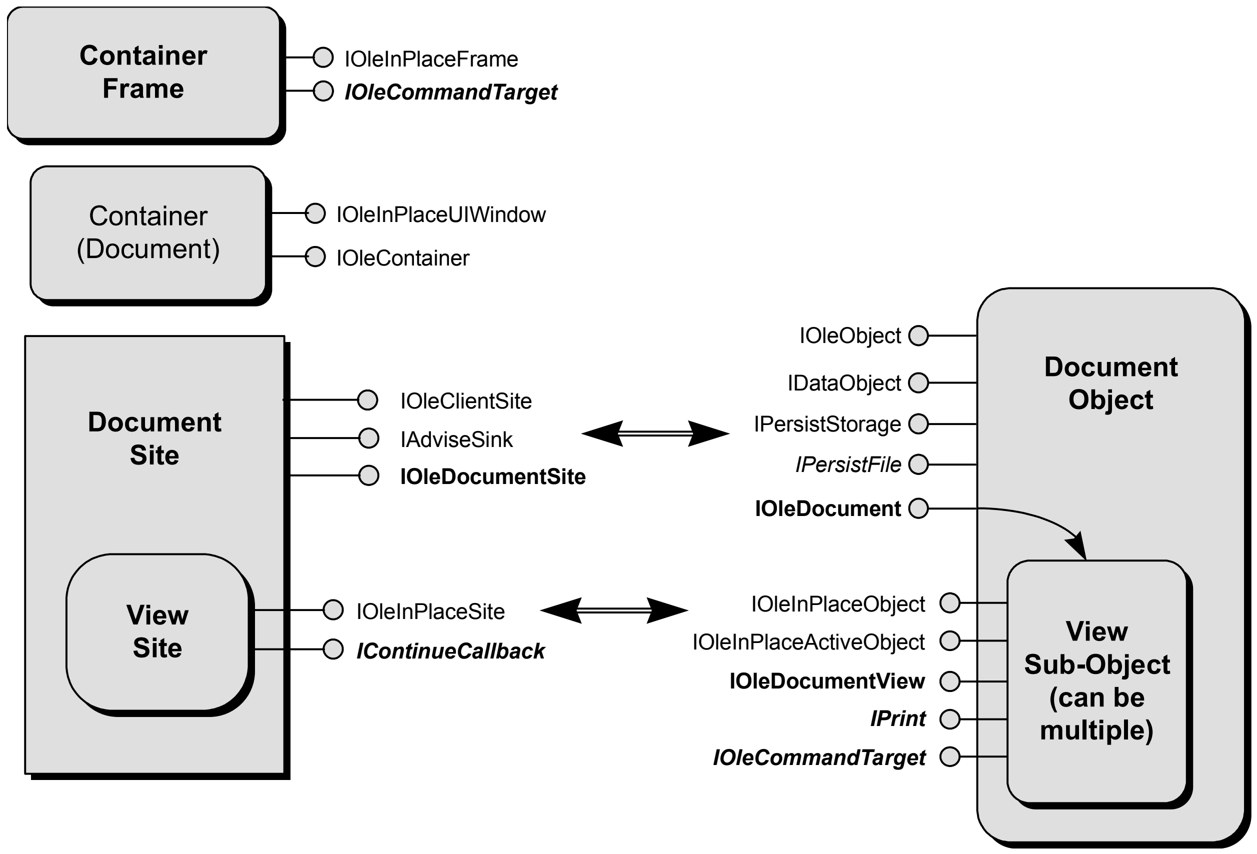
Special attention should be given to the fact that the view and storage aspects of a data set are two entirely orthogonal aspects of the document to which they belong. The storage provider and the view frame provider could be the same, or they could be different. In any case, the application can proceed with its work in a standard manner that isolates it from the need to understand either its storage or its view port in specific detail. In some sense, this separation of views and storage is present within OLE, since when an embedded object is "open edited", the server application provides the view frame while its container provides the storage. However, the Document Objects architecture takes matters much further, since it lets the document's storage container (or some other container) provide the view frame. In addition, moniker binders can also provide view frames, and this can enable in-place activation of links.

Relevant Objects and the Interfaces They Must Implement

A DocObject can support one or more views, each of which is capable of in-place activation. The document component of the object must support standard OLE Document interfaces, but in addition it must support new interfaces such as *IOleDocument*. The view component must also support certain standard OLE interfaces (*IOleInPlaceObject* and *IOleInPlaceActiveObject*), and in addition it must support the new *IOleDocumentView* interface, too. DocObject containers must implement *IOleDocumentSite* along with OLE container interfaces, and they must implement *IOleInPlaceSite* on each view site. Finally, the frame object, the view object(s), and the container object can optionally implement *IOleCommandTarget* to support the dispatch of certain commands (as discussed below).

View and container objects can also optionally implement *IPrint* and *IContinueCallback* (discussed below), to support programmatic printing.

The following illustration shows the conceptual relationships between a container and its components (on the left) and the DocObject and its views (on the right), where the DocObject manages storage and data and the view displays and possibly prints that data. Interfaces in bold are those required for DocObject participation; those bold and italic are optional. All others are required according to OLE Document rules:



Note that a document that supports only a single view can implement both the view and document components (that is, their corresponding interfaces) on a single concrete class. In addition, a container site that only supports one view at a time can combine the document site and the view site into a single concrete site class. The container's frame object, however, must remain distinct, and the container's document component is merely shown here to complete the OLE Documents architecture. This piece of the container is not affected by the Document Objects architecture.

A DocObject is one that has some data and one or more views associated with it. The Document Objects architecture formalizes the relationship between the document, its views, and their view sites/frames.

Document Objects (Server)

The DocObject owns a set of data and has access to storage where the data can be saved and retrieved. It can create and manage one or more views on its data. In addition to supporting the usual embedding and in-place activation interfaces according to OLE Documents, the DocObject communicates its ability to create views through *IOleDocument*. Through this interface the container can ask to create (and possibly enumerate) the views that the DocObject can display. Through this interface, the DocObject can also provide miscellaneous information about itself, such as whether it supports multiple views or complex rectangles.

```
interface IOleDocument : IUnknown
    {
    HRESULT CreateView([in] IOleInPlaceSite *pIPSite, [in] IStream *pstm
        , [in] DWORD dwReserved, [out] IOleDocumentView **ppView);
    HRESULT GetDocMiscStatus([out] DWORD *pdwStatus);
    HRESULT EnumViews([out] IEnumOleDocumentViews **ppEnum
        , [out] IOleDocumentView **ppView);
    }
```

Every DocObject must have a view frame provider with this interface. If the document is not embedded within a container, then the DocObject server itself must provide the view frame. However, when the DocObject is embedded in a DocObject container then the container provides the view frame.

The *IEnumOleDocumentViews* interface is a standard OLE enumerator for *IOleDocumentView* * types.

Views Objects (Server)

A DocObject can create one or more types of views (for example, normal, outline, page layout, etc.) of its data. From a functional perspective, views act like filters through which the data can be seen.[1]

Even if the document has only one type of view, it may still wish to support multiple views as a means of supporting "Window/New Window" functionality (for example, the Window menu in Office applications). Functionally these views are like ports onto a particular method for displaying the data.

To be represented within the a DocObject container, a view component must support *IOleInPlaceObject* and *IOleInPlaceActiveObject* in addition to *IOleDocumentView*:

```
interface IOleDocumentView : IUnknown
    {
    HRESULT SetInPlaceSite([in] IOleInPlaceSite *pIPSite);
    HRESULT GetInPlaceSite([out] IOleInPlaceSite **ppIPSite);
    HRESULT GetDocument([out] IUnknown **ppunk);
    [input_sync] HRESULT SetRect([in] LPRECT prcView);
    HRESULT GetRect([in] LPRECT prcView);
    [input_sync] HRESULT SetRectComplex([in] LPRECT prcView
        , [in] LPRECT prcHScroll, [in] LPRECT prcVScroll
        , [in] LPRECT prcSizeBox);
    HRESULT Show([in] BOOL fShow);
    HRESULT UIActivate([in] BOOL fUIActivate);
    HRESULT Open(void);
    HRESULT CloseView([in] DWORD dwReserved);
    HRESULT SaveViewState([in] IStream *pstm);
    HRESULT ApplyViewState([in] IStream *pstm);
    HRESULT Clone([in] IOleInPlaceSite *pIPSiteNew, [out] IOleDocumentView **ppViewNew);
    }
```

[1] There is some question as to when a server would support multiple views. At the time Document Objects were specified, Microsoft had some plansas to how multiple views might be used, and thus the interfaces were designed to be flexible without putting overhead on the single view implementor. When multiple views will be used is hard to say.

Every view has an associated view site, which encapsulates the view frame and the view port (HWND and a rectangular area in that window). The site exposes this functionality though the standard *IOleInPlaceSite* interface. Note that it is possible to have more than one view port on a single HWND.

Typically each type of view has a different printed representation. Hence views and the corresponding view sites should implement the printing interfaces if *IPrint* and *IContinueCallback,* respectively. The view frame must negotiate with the view provider through *IPrint* when printing begins, so that headers, footers, margins, and related elements are printed correctly. The view provider notifies the frame of printing-related events through *IContinueCallback*. For more information on the use of these interfaces, see "Programmatic Printing" later in this document.

Do note that if a DocObject only supports a single view, then the DocObject and that single view can be implemented using a single concrete class. *IOleDocument::CreateView* simply returns the same object's *IOleDocumentView* interface pointer. In short, it is not necessary that there be two separate object instances when only one view is required.

A view object can also choose to be a command target by implement *IOleCommandTarget*. This allows it to easily receive commands that originate in the container's user interface (such as File New, Open, SaveAs, Print; Edit Copy, Paste, Undo, etc.). For more information, see "Command Targets" later in this document.

Document Site Objects (Container)

In the Document Objects architecture, a document site is the same as a client site object in OLE Documents with the addition of the *IOleDocument* interface:

```
interface IOleDocumentSite : IUnknown
    {
    HRESULT ActivateMe(IOleDocumentView *pViewToActivate);
    }
```

The document site is conceptually the container for one or more "view site" objects that are each associated with individual view objects of the document managed by the document site. If the container only supports a single view per document site, then it can implement the document site and the view site with a single concrete class.

View Site Objects (Container)

A container's "view site" object manages the display space for a particular view of a document. In addition to supporting the standard *IOleInPlaceSite* interface, a view site also generally implements *IContinueCallback* for programmatic printing control. (Note that the view object never queries for *IContinueCallback* so it can actually be implemented on any object the container desires).

A container that supports multiple views must be able to create multiple view site objects within the document site. This provides each view with separate activation and deactivation services as provided through *IOleInPlaceSite.*

Frame Object (Container)

The container's frame object is, for the most part, the same frame that is used for in-place activation in OLE Documents, that is, the one that handles menu and toolbar negotiation. A view object has access to this frame object through *IOleInPlaceSite::GetWindowContext* which also provides access to the container object representing the container document (which can handle pane-level toolbar negotiation and contained object enumeration).

In Document Objects, a container can augment the frame by adding *IOleCommandTarget*. This allows it to receive commands that originate in the DocObject's user interface in the same way that this interface can allow a container to send the same commands (such as File New, Open, SaveAs, Print; Edit Copy, Paste, Undo, etc.) to a DocObject. For more information, see "Command Targets" later in this document.

Help Menu Merging: An Extension to OLE Documents

When an object is active within a container, the menu merging protocol of OLE Documents gives the object complete control of the Help menu. As a result, the container's Help topics are not available unless the user deactivates the object. The Document Objects architecture expands on the rules for in-place menu merging to allow both the container and an active DocObject to share the menu. The new rules are simply additional conventions about what component owns what part of the menu and how the shared menu is constructed.

The new convention is simple. In DocObjects, the Help menu has two top-level menu items organized as follows:

```
Help
   Container Help >
   Object Help    >
```

For example, when a Word section is active in the Office Binder, then the Help menu would appear as follows:

```
Help
   Binder Help >
   Word Help   >
```

Both menu items are cascade menus under which any additional menu items specific to the container and the object are provided to the user. What items appear here will vary with the container and objects involved.

To construct this merged Help menu, the Document Objects architecture modifies the normal OLE Documents procedure. According to OLE Documents, the merged menu bar can have 6 groups of menus, namely *File, Edit, Container, Object, Window, Help*, in that order, and in each group there can be 0 or more menus. The groups *File, Container* and *Window* belong to the container and the groups *Edit*, *Object* and *Help* belong to the object. When the object wants to do menu merging it creates a blank menu bar and hands it over to the container, to let it insert its menus, by calling *IOleInPlaceFrame::InsertMenus*. The object also hands over a structure which is an array of six LONGs (OLEMENUGROUPWIDTHS). After inserting the menus, container would mark how many menus he added in each one of its groups, and then returns. Then the object inserts its menus paying attention to the count of menus in each container group. Then finally object passes the merged menu bar and the array (which contains the count of menus in each group) to OLE, which returns an opaque "menu descriptor" handle. Later the object passes that handle and the merged menu bar to the container, via *IOleInPlaceFrame::SetMenu*. At this time container displays the merged menu bar and also passes the handle to OLE, so that OLE can do proper dispatching of menu messages.

In the modified DocObject procedure, the object must first initialize the OLEMENUGROUPWIDTHS elements to zero before passing it to the container. Then the container would do what it normally does in its menu insertion code with one exception. The container inserts a **Help** pop-up menu as the last item and stores a value of 1 in the last (sixth) entry of the OLEMENUGROUPWIDTHS array (that is, *width[5]* which belongs to the object's Help group). This **Help** popup menu will have only one item which is another popup menu, the "Container Help >" cascade menu as described above.

The object then does its normal menu insertion code, except that before inserting its help menu, it checks the sixth entry of the OLEMENUGROUPWIDTHS array. If the value is 1 and the name of the last menu is **Help** *(or the appropriate localized string)*, then the object inserts its help popup menu as sub-menu of container's **Help** popup menu.

The object then sets the sixth element of OLEMENUGROUPWIDTHS to zero and increments the fifth element by one. This lets OLE know that the **Help** menu belongs to the container and the menu messages corresponding that menu (and its sub menus) should be routed to the container. It is then the container's responsibility to forward WM_INITMENUPOPUP, WM_SELECT, WM_COMMAND, and other menu-related messages that belong to the object's portion of the help menu.[2] The container should use the window returned from the object's *IOleInPlaceActiveObejct::GetWindow* function as the destination for these messages.

If the object detects a zero in the sixth element of OLEMENUGROUPWIDTHS it otherwise proceeds according to the normal OLE Documents rules. This will take care of containers that do participate in help menu merging as well as those which do not.

When the object calls *IOleInPlaceFrame::SetMenu,* before displaying the merged menu bar, the container checks whether his **Help** popup menu has an additional sub-menu, in addition to what it has inserted. If so the container would

[2] This is accomplished by using WM_INITMENU to clear a flag that tells the container whether or not the user has navigated into the object's Help menu. The container then watches WM_MENUSELECT for entry into or exit from any item on the Help popup that the container did not add itself. On entry, it means the user has navigated into an object popup, so the container sets the "in object Help menu" flag and uses the state of that flag to forward any WM_MENUSELECT, WM_INITMENUPOPUP, and WM_COMMAND messages, as a minimum, to the object window. On exit, the container clears the flag and then processes these same messages itself.

leave his **Help** popup menu in the merged menu bar, else he will remove it from the merged menu bar. This will take care of the objects that do participate in help menu merging as well as those that do not.

Finally, during menu disassembling time, the object would remove the inserted help menu, in addition to removing the other inserted menus. And when container gets a chance to removed its menus, it will remove its help popup menu in addition to the other menus that it has inserted.

Programmatic Printing (IPrint & IContinueCallback)

OLE provided the means to uniquely identify persistent documents (*GetClassFile*) and load them into their associated code (*CoCreateInstance*, *QueryInterface(IID_IPersistFile/IID_IPersistStorage...), IPersistFile/IPersistStorage::Load)*. To further enable printing of documents, Document Objects (using an existing OLE design not shipped with OLE 2.0 originally) introduces a base-standard printing interface, *IPrint,* generally available through any object which can load the persistent state of the document type. Each view of a document object in the Document Objects architecture can optionally support the *IPrint* interface to provide these capabilities.

The *IPrint* interface is defined as follows:

```
interface IPrint : IUnknown
    {
    HRESULT SetInitialPageNum([in] LONG nFirstPage);
    HRESULT GetPageInfo([out] LONG *nFirstPage, [out] LONG *pcPages);
    HRESULT Print([in] DWORD grfFlags, [in,out] DVTARGETDEVICE **pptd
        , [in,out] PAGESET ** ppPageSet , [in,out] STGMEDIUM **ppstgmOptions
        , [in] IContinueCallback* pCallback, [in] LONG nFirstPage
        ,[out] LONG *pcPagesPrinted, [out] LONG *pnPageLast);
    };
```

Clients and containers simply use *IPrint::Print* to instruct the document to print itself once that document is loaded, specifying printing control flags, the target device, the pages to print, and additional options. The client can also control the continuation of printing through the interface *IContinueCallback* (see below).

In addition, *IPrint::SetInitialPageNum* supports the ability to print a series of documents together as if they were one by numbering pages seamlessly, obviously a benefit for DocObject containers like Office Binder. *IPrint::GetPageInfo* simply allows the caller to retrieve the starting page number previously passed to *SetInitialPageNum* (or the document's internal default starting page number) and the number of pages in the document, useful for displaying pagination information.

Objects that support *IPrint* mark themselves in the registry with the "Printable" key stored under the object's CLSID:

```
HKEY_CLASSES_ROOT\CLSID\{...}\Printable
```

IPrint is usually implemented on the same object supporting either *IPersistFile* or *IPersistStorage*. Callers note the capability to programmatically print the persistent state of some class by looking in the registry for the "Printable" key. At the time being, "Printable" indicates support for at least *IPrint*; other interfaces may be defined in the future which would then be available through *QueryInterface* where *IPrint* simply represents the base level of support.

During a print procedure, the client or container that initiated the printing may wish to control whether or not the printing should continue. For example, the container may support a "Stop Print" command that should terminate the print job as soon as possible. To support this capability, the client of a printable object can implement a small notification sink object with the interface *IContinueCallback:*

```
interface IContinueCallback : IUnknown
    {
    HRESULT FContinue(void);
    HRESULT FContinuePrinting([in] LONG cPagesPrinted, [in] LONG nCurrentPage
        , [in] LPOLESTR pszPrintStatus);
    };
```

This interface is designed to be useful as a generic continuation callback function which takes the place of the various continuation procedures in the Win32 API (such as the *AbortProc* for printing and the *EnumMetafileProc* for metafile enumeration). Thus this interface design is useful in a wide variety of time-consuming processes.

In the most generic cases, *IContinueCallback::FContinue* function is called periodically by any lengthy process. The sink object returns S_OK to continue the operation, S_FALSE to stop the procedure as soon as possible.

FContinue, however, is not used in the context of *IPrint::Print*; rather, printing uses *IContinueCallback::FContinuePrint*. Any printing object should periodically call *FContinuePrinting* passing the number of pages that have been printing, the number of the page being printed, and an additional string describing the print status that the client may choose to display to the user (such as "Page 5 of 19").

Complete details of these interfaces is given in the reference section at the end of this document.

Command Targets

The command dispatch interface *IOleCommandTarget* defines a simple and extensible mechanism to query and execute commands. This mechanism is simpler than OLE Automation's *IDispatch* because it relies entirely on a standard set of commands, commands rarely have arguments, and no type information is involved (type safety is diminished for command arguments as well).

In this design, each command belongs to a "command group" which is itself identified with a GUID. Therefore anyone can define a new group and define all the commands within that group without any need to coordinate with Microsoft nor any other vendor.[3]

IOleCommandTarget handles the following scenarios:

1. When an object is in-place activated, only the object's toolbars are typically displayed and the object's toolbars may have buttons for some of the container commands like "Print," "Print Preview," "Save," "New," "Zoom," etc.[4] Currently there is no mechanism for the object to dispatch these commands to the container.
2. When a DocObject is embedded in a DocObject container (such as Binder), the container may need to send commands such "Print," "Page Setup," "Properties," etc. to the contained DocObject.

Obviously this simple command routing could be handled through existing OLE Automation standards and *IDispatch.* However, the overhead involved with *IDispatch* is more than is necessary here, so *IOleCommandTarget* provides a simpler means to achieve the same ends:

```
interface IOleCommandTarget : IUnknown
    {
    HRESULT QueryStatus([in] GUID *pguidCmdGroup, [in] ULONG cCmds
        , [in,out][size_is(cCmds)] OLECMD *prgCmds, [in,out] OLECMDTEXT *pCmdText);
    HRESULT Exec([in] GUID *pguidCmdGroup, [in] DWORD nCmdID, [in] DWORD nCmdExecOpt
        , [in] VARIANTARG *pvaIn, [in,out] VARIANTARG *pvaOut);
    }
```

The *QueryStatus* method here tests whether a particular set of commands, the set being identified with a GUID, is supported. This call fills an array of OLECMD values (structures) with the supported list of commands as well as returning text describing the name of a command and/or status information. When the caller wishes to invoke a command, it can pass the command (and the set GUID) to *Exec* along with options and arguments, getting back a return value.

For more information on this interface, see the reference section at the end of this document.

[3] This is essentially the same means of definition as a dispinterface plus dispIDs in OLE Automation. There is overlap here, although this command routing mechanism is just for command routing and not for scripting/programmability on a large scale as OLE Automation handles.
[4] In-place activation standards recommend that objects remove such buttons from their toolbars, or at least disable them. This design allows those commands to be enabled and yet routed to the right handler.

Implementation Notes

Becoming a DocObject Server

This section discusses issues related to server side implementation of the Document Objects architecture, specifically the implementation of a DocObject and its view.

A DocObject can be implemented as an in-process object or as a local object (in an EXE). The Document Objects architecture has been designed so that it is relatively easy to transform an existing in-place implementation into a DocObject. The document object itself must support those interfaces described earlier which will require existing object implementations to slightly modify their code in several places: *IOleObject::DoVerb*, *IOleObject::SetClientSite*, and in-place activation functions. The following sections describe these issues in more detail.

IOleObject::SetClientSite

An object must be able to determine whether it can and should activate as a DocObject. This will depend on whether the client site (that is, the container) supports *IOleDocumentSite.* When an object's *IOleObject::SetClientSite* is called, it should query the given pointer for *IOleDocumentSite* as the following code illustrates:

```
HRESULT IOleObject::SetClientSite(IOleClientSite *pSite)
    {
    //Perform regular SetClientSite processing.

    // If we currently have a document site pointer, release it.
    if (NULL!=m_pDocSite)
        {
        ReleaseInterface(m_pDocSite);  //Macro to Release and NULL
        m_fDocObj=FALSE;
        }

    if (NULL!=pSite)
        {
        if (SUCCEEDED(pSite->QueryInterface(IID_IOleDocumentSite, &m_pDocSite)))
            m_fDocObj=TRUE;
        }
    }
```

IOleObject::DoVerb

When a DocObject's *IOleObject::DoVerb* is called, it will know whether to activate itself as a DocObject or not as determined in *IOleObject::SetClientSite*. One DocObject support is acknowledged, various verbs are handled differently than a normal embedded object would handle them.

Verb	Handling Procedure
OLEIVERB_SHOW	The object calls *IOleDocumentSite::ActivateMe.* The object does not call *IOleClientSite::ShowObject* nor *IOleClientSite::OnShowWindow* at this time because it waits until calls to *IOleDocumentView* for specific activation instructions.
OLEIVERB_OPEN	Same as OLEIVERB_SHOW—note that this is *not recommended* for containers.
OLEIVERB_UIACTIVATE	Same as OLEIVERB_SHOW.
OLEIVERB_HIDE	The object should return an error (E_INVALIDARG)

In-Place Activation Differences

When activating as a DocObject, the object should behave as follows:

- Bypass displaying the in-place hatch border and object adornments (such as sizing handles etc.)
- Do not generate *IOleInPlaceSite::OnPosRectChange* calls (no need for them)
- Ignore *IOleObject::SetExtent* calls

- Draw scroll bars within the view rectangle (see *IOleDocumentView::SetRect* and *SetRectComplex*) as opposed to drawing them outside that rectangle (as in normal in-place activation)
- Do not call *IOleClientSite::ShowObject* during activation.

Storage requirements.

The storage format of a DocObject must be the same whether it opens the file on its own and writes the data or whether it saves that data into storage provided by its container. In short, the DocObject must depends on *IStorage* and *IStream* for its persistence mechanisms. This enables a DocObject container to take the data in the object's storage and create a file out of it. Binder, for example, uses this mechanism to move the bound sections on to the shell.

In standard OLE, when *IPersistFile::Save* method is called with NULL for the file name, then the object must save itself into the file that it currently owns. The frame provider, which is not the storage provider, can use this mechanism to ask the document to save itself into the storage it currently owns.

Registration

Every DocObject server should include the "DocObject" key in the registry entries of its supported classes. This key indicates Document Objects support. For example:

```
HKCR\Word.Document.6\DocObject = 0
HKCR\CLSID\{<CLSID for Word Document>} = Microsoft Word 7.0 Document
HKCR\CLSID\{<CLSID for Word Document>}\DocObject = 0
```

(HKCR is short for HKEY_CLASSES_ROOT.)

The DocObject subkey should appear under both the server's ProgID and its CLSID. The value of the "DocObject" key indicates whether the DocObject can create multiple views and whether it can accept complex rectangles. See *IOleDocument::GetDocMiscStatus* for more information.

The DocObject must also use the "DefaultExtension" key to register the default extension used by its files along with a descriptive string that can be used in a File Open or File Save As dialog. For example:

```
\<CLSID for Word Document>\DefaultExtension=.doc, Word Documents (*.doc)
```

Finally, if the object supports the *IPrint* interface, it must register the "Printable" key. For example:

```
\<CLSID…>\Printable
```

Limiting Embedding Support

All DocObjects will be embeddable due to the fact that they implement all the relevant interfaces for OLE Documents (*IOleObject, IDataObject, IPersistFile*, and *IPersistStorage*). However, they can choose to limit the embedding functionality they support. This can be done as follows:

- Do not register the "Insertable" key to prevent compound document containers from listing the document object class in the Insert Object dialog.[5]
- Do not offer "Embed Source" or "Embedded Object" formats in data exchange operations. This prevents the object from being pasted into compound document containers.
- Set the OLEMISC_CANTLINKINSIDE bit in your MiscStatus key of the registry to prevent linking to embedded DocObjects.
- Set the OLEMISC_ICONICONLY bit to force the document to appear as an icon in any container that might still receive the object through the Insert From File dialog (an option in Insert Object) or when the file is dropped on a container from the system shell. Because it is only displayed as an icon, there is no need to worry about generating metafiles nor in handling *IOleObject::SetExtent* calls, etc.

[5] Also do not register a "\protocol\StdFileEditing\server" to prevent inclusion in an OLE 1 container's Insert Object dialog.

Becoming a DocObject Container

This section discusses issues related to container or host side implementation of the Document Objects architecture. It goes without saying that a container supports the necessary interfaces as described in the architecture. However, there are a number of other considerations:

1. Storage provisions and user interface
2. Creation and initialization of a DocObject
3. Activation of a DocObject
4. DocObject saving and shutdown
5. Support for other OLE features, completeness of interface implementations

The following sections describe each of these topics in more detail. These are the core pieces of a DocObject container that require more comment than is found elsewhere in this specification, and the following discussion is not intended to touch on every container-side detail. As such, specific items like Help menu merging and command targets are not described here and are left for sample code to demonstrate.

Storage Provisions and User Interface

A DocObject container is generally a container that manages multiple “documents” (from the user perspective) in a single data store of some kind. Now, that data store could be something as complex as an entire file system, or it could be something simple like an individual compound file. In general, the container’s methods for dealing with the ultimate storage of documents edited as DocObjects will in many ways determine the type of user interface that the container supports.

The Office Binder, for example, uses a single “Binder” file, an OLE Compound File, as its own data store. Within that single Binder file, the Binder can embed any number of other documents as “sections” in the binder. Technically speaking, while the Binder file itself is a single root instance of *IStorage*, each section is then given the *IStorage* of a sub-storage within the root. Each embedded DocObject is handed this sub-storage pointer through *IPersistStorage::InitNew* or *IPersistStorage::Load* (either at creation or reloading time, respectively) and stores all of its data directly in that storage.

What the Office Binder does for a user interface, then, is provide a left-hand pane that displays the “documents” or sections in the binder, activating them one at a time in the right-hand pane as if they were being opened in their respective applications. However, one never leaves the binder paradigm as one changes from section to section. Each so-called document is just a sub-storage in the entire binder.

Now a container that browses a file system, on the other hand, will see the whole file system, or the World Wide Web for that matter, as a single “file” or “binder” in which are found many individual documents. This kind of container would have the browsing UI in the left hand pane and would individually activate DocObjects within a viewing pane of that browser. In this case the *IStorage* handed to each DocObject is the root *IStorage* for the entire document on the file system itself.

One must not confuse the use of an *IStorage* in the DocObjects architecture with the use of streams to save and re-load view states through *IOleDocumentView::SaveViewState* and *IOleDocumentView::ApplyViewState* as described in more detail below.

Creation and Initialization

However a container wishes to create an embedded DocObject is up to that container. This will generally involve one of the OLE API functions *OleCreate, OleCreateFromData, OleCreateFromFile*, and *OleLoad*. *OleCreate*, of course, is used to create a new, uninitialized DocObject—when that object is activated the user starts with a clean slate. *OleCreateFromData* and *OleCreateFromFile*, on the other hand, create new instances of objects with a state initialized from either the contents of a data object (clipboard, drag and drop, etc.) or from the contents of a file, respectively. Once a DocObject is saved to its *IStorage* via *OleSave*, it can then be reloaded with *OleLoad*, of course.

The full initialization sequence for a DocObject will depend on the exact nature of the container. As a minimum, however, it will involve these steps after creation or loading:

1. *IPersistStorage::InitNew* (create) or *IPersistStorage::Load* (reload)

2. *IOleObject::SetClientSite*
3. *IOleObject::Advise*

These three calls will initialize the DocObject and set up communication between it and the container's *IOleClientSite* and *IAdviseSink* interfaces. Nothing more is essential, although containers that display something like an iconic rendering of the object may also include calls to *IViewObject2* members such as *GetExtent* and *SetAdvise*.

Note that as described in the server section above, the container's call to *IOleObject::SetClientSite* will generate a *QueryInterface* call to the container for *IOleDocumentSite.* The object then uses this interface during activation, which is the next topic.

Activation

Activating a DocObject is largely just a matter of calling *IOleObject::DoVerb(OLEIVERB_SHOW, ...)* then responding to *IOleDocumentSite::ActivateMe*. OLEIVERB_SHOW is generally the most appropriate activation verb here, but OLEIVERB_PRIMARY and OLEIVERB_UIACTIVATE are also allowable. OLEIVERB_OPEN isn't recommended as highly because it implies separate-window activation instead of an in-place activation.

Activation of a DocObject is almost entirely self-contained within *IOleDocumenSite::ActivateMe* whose implementation generally appears as follows:

```
STDMETHODIMP CImpIOleDocumentSite::ActivateMe(IOleDocumentView *pView)
    {
    RECT                rc;

    /*
     * If we're passed a NULL view pointer, then try to get one from
     * the document object.
     */
    if (NULL==pView)
        {
        IOleDocument *pDoc;

        if (FAILED(m_pSite->m_pObj->QueryInterface(IID_IOleDocument
            , (void **)&pDoc)))
            return E_FAIL;

        if (FAILED(pDoc->CreateView(m_pSite->m_pImpIOleIPSite, NULL
            , 0, &pView)))
            return E_OUTOFMEMORY;
        }
    else
        {
        //Make sure that the view has our client site
        pView->SetInPlaceSite(m_pSite->m_pImpIOleIPSite);

        //We're holding onto the pointer, so AddRef it.
        pView->AddRef();
        }

    //Remember the type of object we have and the view pointer
    m_pSite->m_fDocObj==TRUE;
    m_pSite->m_pIOleDocView=pView;

    //This sets up toolbars and menus first
    pView->UIActivate(TRUE);

    //Set the window size sensitive to new toolbars
    GetClientRect(m_pSite->m_hWnd, &rc);
    pView->SetRect(&rc);

    //Makes it all visible
    pView->Show(TRUE);

    return NOERROR;
    }
```

This code is taken from a working DocObject container and demonstrates the proper sequence of operations for DocObject activation:

1. If *ActivateMe* is passed an *IOleDocumentView* pointer, then call *IOleDocumentView::SetInPlaceSite* followed by *AddRef* if you're holding onto the pointer (which is generally the case). Otherwise query the document object itself for *IOleDocument* and call *IOleDocument::CreateView* passing in the container's *IOleInPlaceSite* pointer. In both cases you'll end up with an *IOleDocumentView* pointer for the DocObject's view that should be released when the container no longer needs it.
2. Activate the DocObject view by calling *IOleDocumentView::UIActivate(TRUE)* which will cause it to perform menu merging, toolbar negotiation, and re-parent its display window to the window returned through *IOleInPlaceSite::GetWindow.* Part of the toolbar negotiation sequence should be for the container to remember exactly how much border space is taken up, resizing any client-area windows in the container to account for this space.
3. Call *IOleDocumentView::SetRect* (or *SetRectComplex* depending on the container) to tell the view exactly how much space to occupy in its parent. If the container manages a client-area window as the code sample above is doing, then this rectangle is simply the client area of that window. Note that this step is important to do *after* calling *UIActivate* because the container would otherwise send the view the wrong dimensions that wouldn't account for toolbar space.
4. Call *IOleDocumentView::Show(TRUE)* to make the DocObject visible. This is the last step because the DocObject view knows exactly what space it occupies and all its other tools are there.

While the DocObject remains active, it is also imperative that the container fulfill a few other requirements, some of which come from standard in-place activation rules:

1. Call *IOleInPlaceActiveObject::ResizeBorder* when the container frame is resized so the object can resize its toolbars appropriately.
2. Call *IOleDocumentView::SetRect* whenever the window used for the DocObject parent is resized. This might be the frame window, a client-area window, or a document window (in an MDI container). *SetRect* tells the DocObject to resize its view to fully occupy the parent window's client area.
3. Implement *IOleInPlaceFrame::SetStatusText* if the container has a toolbar.
4. Call *IOleInPlaceActiveObject::TranslateAccelerator* from the container's message loop.
5. Detect the F1 key to enter context-sensitive help mode as well as ESC to leave it, calling *IOleInPlaceActiveObject::ContextSensitiveHelp* with TRUE and FALSE, respectively.
6. Handle WM_SETFOCUS to the frame window by setting focus to the window returned from *IOleInPlaceActiveObject::GetWindow*.

All of these bits other than step 2 are standard in-place activation requirements.

Saving and Shutdown

When a DocObject is closed, the container should ensure that its data is saved as it would with any other embedded object. That is, the container must handle *IOleClientSite::SaveObject* in which it generates a call to the object's *IPersistStorage::Save*, usually through *OleSave,* followed by *IPersistStorage::SaveCompleted* and an *IStorage::Commit* if transactioned storage is being employed.

When the container wishes to close the object entirely, that is, unload it completely (with or without saving), then the container should first call *IOleInPlaceObject::InPlaceDeactivate*, *IOleObject::Close*, followed by *Release* calls on all interface pointers that the container is holding. When the last reference count is released, the DocObject will delete itself and its server will shut down as appropriate.

Again, all of this is standard for standard embedding scenarios in OLE Documents.

Support for Other OLE Features and Completeness of Interface Implementations

As described in the previous section, DocObject servers will never generate calls to various members of the *IOleClientSite* and *IOleInPlaceSite* interfaces, such as:

- *IOleClientSite::GetMoniker*
- *IOleClientSite::GetContainer*

- *IOleClientSite::RequestNewObjectLayout*
- *IOleClientSite::OnShowWindow*
- *IOleClientSite::ShowObject*
- *IOleInPlaceSite::OnPosRectChange*
- *IOleInPlaceSite::Scroll*
- *IOleInPlaceSite::ContextSensitiveHelp* (if container has no support for this)

Therefore strictly DocObject containers can simply return E_NOTIMPL from these members. In addition, most of the *IAdviseSink* members need no implementation, with *IAdviseSink::OnClose* being the only one of probable interest; in some cases a container may not need *IAdviseSink* at all, and thus would never need to call *IOleObject::Advise* (or *IViewObject::SetAdvise* when the container doesn't display anything for the object visually).

All other members of *IOleClientSite* and *IOleInPlaceSite*, as well as those in *IOleInPlaceFrame* require some implementation which is sometimes considerable and at other times nothing more than a return of NOERROR (such as *IOleInPlaceSite::CanInPlaceActivate*).

Of course, the container may support more than just DocObjects—it might also support normal OLE compound document linking and embedding in which case it will completely implement these interfaces as specified for OLE Documents. The container may also support OLE Controls in which case it would have *IDispatch*, event handlers, *IOleControlSite*, and so on. It should be noted, however, that DocObjects do not interfere with support for these other types of objects, provided that the container maintains a variable (like *m_fDocObj* as described in the activation section above) that tells the rest of its code that certain operations won't be needed when a DocObject is in use.

Finally, there is also the separate-window activation model available through *IOleDocumentView* which can be employed as a container sees fit. Support for that model is not a requirement for all DocObject containers, however.

Document Objects Interface Reference

This section lists all interfaces, related structures, and related enumerations that are defined by this architecture. The section has a detailed description of interface member functions and their arguments.

The IOleDocument *Interface*

By implementing this interface alongside other interfaces relating to OLE Documents, an object indicates its ability to act as a "document object." Through this interface, a container for DocObjects can ask the object to create views of itself as well as to enumerate those views and to retrieve MiscStatus bits related to the document object.

IDL:

```
[
uuid(B722BCC5-4E68-101B-A2BC-00AA00404770)
    , object, pointer_default(unique)
]
interface IOleDocument : IUnknown
    {
    HRESULT CreateView([in] IOleInPlaceSite *pIPSite, [in] IStream *pstm
        , [in] DWORD dwReserved, [out] IOleDocumentView **ppView);
    HRESULT GetDocMiscStatus([out] DWORD *pdwStatus);
    HRESULT EnumViews([out] IEnumOleDocumentViews **ppEnum
        , [out] IOleDocumentView **ppView);
    }
```

IOleDocument::CreateView

HRESULT IOleDocument::CreateView([in] IOleInPlaceSite *pIPSite, [in] IStream *pstm
, [in] DWORD dwReserved, [out] IOleDocumentView **ppView)

Ask the document object to create a new view sub-object, returning that view object's *IOleDocumentView* interface pointer. Optionally this call can also initialize the view from the contents a given stream. A container calls this function to both create new views as well as to reload previously saved views. The view must wait for calls to either *IOleDocumentView::Show* or *IOleDocumentView::UIActivate* before showing itself.

Argument	Type	Description
pIPSite	*IOleInPlaceSite **	A pointer to the container's "view site" object associated with the new view. May be NULL in which case the caller must initialize the view with a call to *IOleDocumentView::SetInPlaceSite*.
pstm	*IStream **	A pointer to the stream from which the view should initialize itself. If NULL, then this function creates a new view with a default state.
dwReserved	*DWORD*	Reserved for future use. Must be zero.[6]
ppView	*IOleDocumentView **	Address of the variable to receive the interface pointer to the new view. If *CreateView* succeeds, the caller is responsible for calling *Release* through this pointer when the view object is no longer needed.

Return Value	Meaning
S_OK	The view was created successfully.
E_POINTER	The address in *ppView* is NULL.
E_OUTOFMEMORY	There is not enough memory to create the new view.
E_UNEXPECTED	An unknown error occurred.

[6] In the future this parameter could be used to specify the type of view that needs to be created. Currently there are no defined values for this argument.

E_FAIL	This document object only supports a single view which has already been created.

Comments:
This function must be completely implemented in any document object; therefore E_NOTIMPL is not an acceptable return code.

As with all new interface pointers, *CreateView* calls *AddRef* on the pointer in **ppView* before returning. The caller is responsible for calling *Release* through this pointer when it is no longer needed.

If *pIPSite* is non-NULL, then the document object should pass the pointer to the new view through *IOleDocumentView::SetInPlaceSite*. If NULL, the caller is responsible for this same call. In addition, if *pstm* is non-NULL, then the object should initialize the view object by passing *pstm* to *IOleDocumentView::ApplyViewState*.

IOleDocument::GetDocMiscStatus

HRESULT IOleDocument::GetDocMiscStatus([out] DWORD *pdwStatus)

Returns miscellaneous status bits describing the document object, such as whether the object can create multiple views and accept complex rectangles.[7] These values are also stored in the registry as the value of the "DocObject" key:

```
typedef enum
    {
    DOCMISC_CANCREATEMULTIPLEVIEWS   = 1, //Object supports multiple views
    DOCMISC_SUPPORTCOMPLEXRECTANGLES = 2, //IOleDocumentView::SetRectComplex is supported
    DOCMISC_CANTOPENEDIT             = 4, //IOleDocumentView::Open is unsupported
    DOCMISC_NOFILESUPPORT            = 8  //Object does not support file read/write
    } DOCMISC;
```

The bits DOCMISC_CANTOPENEDIT, DOCMISC_NOFILESUPPORT need further explanation. There can be objects which can only be embedded, can only be in-place activated, and which do not have files of their own, regardless of whether they are implemented as in-process or local servers. Objects which have limited UI for activation purposes should set DOCMISC_CANTOPENEDIT. Those that only support *IPersistStorage* as a persistence mechanism should specify DOCMISC_NOFILESUPPORT. Otherwise and object must also implement *IPersistFile* implementation.

If an object desires none of these status bits it must return a zero in **pdwStatus*.

Argument	**Type**	**Description**
pdwStatus	*DWORD **	The address of the variable to receive the status bits about this document object.

Return Value	**Meaning**
S_OK	The status bits were returned successfully.
E_POINTER	The address in *pdwStatus* is NULL.

Comments:
This function must be completely implemented in any document object even if a zero is returned; therefore E_NOTIMPL is not an acceptable return code.

IOleDocument::EnumViews

HRESULT IOleDocument::EnumViews([out] IEnumOleDocumentViews **ppEnum
    , [out] IOleDocumentView **ppView)

Creates an enumerator object that enumerates the IOleDocumentView interface pointers of the views of the document object. The enumerator supports the interface *IEnumOleDocumentViews,* a pointer to which is returned in **ppEnum*. An object that supports only a single view (that is, DOCMISC_CANCREATEMULTIPLEVIEWS is not specified

[7] One rectangle each for view, horizontal scroll bar, vertical scroll bar and size box. See *IOleDocumentView::SetRectComplex*.

through *IOleDocument::GetMiscStatus*) does not create an enumerator but instead returns the single view pointer through **ppView*.

Argument	Type	Description
ppEnum	*IEnumOleDocumentViews* **	The address of the variable to receive the interface pointer of the enumerator.
ppView	*IOleDocumentView* **	The address of the variable to receive the interface pointer of a single view.

Return Value	Meaning
S_OK	If the object supports multiple views, then **ppEnum* contains the enumerator pointer. Otherwise **ppEnum* is NULL and **ppView* contains the interface pointer to the single view..
E_POINTER	The address in *ppEnum* or *ppView* is invalid. The caller must pass pointers for both arguments.
E_OUTOFMEMORY	The enumerator could not be created because there is insufficient memory.

Comments:
This function must be completely implemented in any document object; therefore E_NOTIMPL is not an acceptable return code.

The IEnumOleDocumentViews *Interface*

A document object can be asked to enumerate its views through *IOleDocument::EnumViews*. The resulting enumerator returned from this member implements the interface *IEnumOleDocumentViews* through which a client can access all the individual view sub-objects supported within the document object itself, where each view implements *IOleDocumentView.*

Therefore *IEnumOleDocumentViews* is a standard enumerator interface typed for *IOleDocumentView* *.

IDL:

```
[
uuid(B722BCC8-4E68-101B-A2BC-00AA00404770)
    , object, pointer_default(unique)
]
interface IEnumOleDocumentViews : IUnknown
    {
    HRESULT Next([in] ULONG cViews
        , [out, max_is(cViews)] IOleDocumentView **rgpView
        , [out] ULONG *pcFetched);

    HRESULT Skip([in] ULONG cViews);
    HRESULT Reset(void);
    HRESULT Clone([out] IEnumOleDocumentViews **ppEnum);
    }
```

IEnumOleDocumentViews::Next

HRESULT IEnumOleDocumentViews::Next([in] ULONG cViews , [out, max_is(cViews)] IOleDocumentView **rgpView, [out] ULONG *pcFetched);

Enumerates the next *cViews* elements in the enumerator's list, returning them in *rgpView* along with the actual number of enumerated elements in *pcFetched.* The caller is responsible for calling *IOleDocumentView::Release* through each pointer returned in *rgpView*.

Argument	Type	Description
cViews	*ULONG*	Specifies the number of *IOleDocumentView** values to return in the array pointed to by *rgpView*. This argument must be 1 if *pcFetched* is NULL.
rgpView	*IOleDocumentView* *	A pointer to a caller-allocated *IOleDocumentView* * array of size *cViews* in which to return the enumerated document views. The caller is responsible for calling *IOleDocumentView::Release* through each pointer enumerated into the array once this method returns successfully. If *cViews* is greater than 1 the caller must also pass a non-NULL pointer passed to *pcFetched* to know how many pointers to release.
pcFetched	*ULONG*	A pointer to the variable to receive the actual number of document views enumerated in *rgpView*. This argument can be NULL in which case the *cViews* argument must be 1.

Return Value	Meaning
S_OK	The requested number of elements has been returned and **pcFetched* (if non-NULL) is set to *cViews* if
S_FALSE	The enumerator returned fewer elements than *cViews* because there were not that many elements left in the list.. In this case, unused elements in *rgpView* in the enumeration are not set to NULL and **pcFetched* holds the number of valid entries, even if zero is returned.

E_POINTER	The address in *rgpView* is not valid (such as NULL)
E_INVALIDARG	The value of *cViews* is not 1 when *pcFetched* is NULL; or the value of *cViews* is zero.
E_UNEXPECTED	An unknown error occurred.
E_OUTOFMEMORY	There is not enough memory to enumerate the elements.

Comments:
E_NOTIMPL is not allowed as a return value. If an error value is returned, no entries in the *rgpView* array are valid on exit and require no release.

IEnumOleDocumentViews::Skip

```
HRESULT IEnumOleDocumentViews::Skip([in] ULONG cConnections);
```

Instructs the enumerator to skip the next *cViews* elements in the enumeration such that the next call to *IEnumOleDocumentViews::Next* will not return those elements.

Argument	Type	Description
cViews	*ULONG*	Specifies the number of elements to skip in the enumeration.

Return Value	Meaning
S_OK	The number of elements skipped is *cViews*.
S_FALSE	The enumerator skipped fewer than *cViews* because there were not that many left in the list. The enumerator will, at this point, be positioned at the end of the list such that subsequent calls to *Next* (without an intervening *Reset*) will return zero elements.
E_INVALIDARG	The value of *cViews* is zero, which is not valid.
E_UNEXPECTED	An unknown error occurred.

IEnumOleDocumentViews::Reset

```
HRESULT IEnumOleDocumentViews::Reset(void);
```

Instructs the enumerator to position itself back to the beginning of the list of elements.

Argument	Type	Description
NA	NA	NA

Return Value	Meaning
S_OK	The enumerator was successfully reset to the beginning of the list.
S_FALSE	The enumerator was not reset to the beginning of the list.
E_UNEXPECTED	An unknown error occurred.

Comments:
There is no guarantee that the same set of elements will be enumerated on each pass through the list: it depends on the collection being enumerated. It is too expensive for some collections, such as files in a directory, to maintain this condition.

IEnumOleDocumentViews::Clone

```
HRESULT IEnumOleDocumentViews::Clone([out] IEnumOleDocumentViews **ppEnum);
```

Creates another view enumerator with the same state as the current enumerator, which iterates over the same list. This makes it possible to record a point in the enumeration sequence in order to return to that point at a later time.

Argument	Type	Description
ppEnum	*IEnumOleDocumentViews**	The address of the variable to receive the *IEnumOleDocumentViews* interface pointer to the newly created enumerator. The caller must release this new enumerator separately from the first enumerator.

Return Value	Meaning
S_OK	Clone creation succeeded.
E_NOTIMPL	Cloning is not supported for this enumerator.
E_POINTER	The address in *ppEnum* is not valid (such as NULL)
E_UNEXPECTED	An unknown error occurred.
E_OUTOFMEMORY	There is not enough memory to create the clone enumerator.

The IOleDocumentSite *Interface*

By implementing this interface on an client site alongside other client site interfaces required by OLE Documents, a container indicates its support for document object activation to any such objects associated with this site. The interface allows a document object to ask the container to activate it as a document instead of as an in-place embedded object. The document object can alternately specify which view to activate.

The view site encapsulates the view port (the HWND and a rectangle in that HWND) and the frame context of the view port. There can be multiple view ports in a single window. A view site is attached to a view through the *pIPSite* argument of *IOleDocument::CreateView* or through *IOleDocumentView::SetInPlaceSite.*

IDL:

```
[
uuid(B722BCC7-4E68-101B-A2BC-00AA00404770)
    , object, pointer_default(unique)
]
interface IOleDocumentSite : IUnknown
    {
    HRESULT ActivateMe([in] IOleDocumentView *pViewToActivate);
    };
```

IOleDocumentSite::ActivateMe

HRESULT IOleDocumentSiteActivateMe([in] IOleDocumentView *pViewToActivate)

When a document object is asked to in-place activate through *IOleObject::DoVerb*, a document object bypasses the normal in-place activation sequence of OLE Documents and instead calls *IOleDocumentSite::ActivateMe* to become active as a document. This should be done in the OLEIVERB_OPEN, OLEIVERB_SHOW, OLEIVERB_INPLACEACTIVATE, and OLEIVERB_UIACTIVATE cases.

The document object can specify which view to activate by passing that view's *IOleDocumentView* pointer in *pViewToActivate*. The container in this case will proceed and activate that view through that pointer. Otherwise, the container calls the object's *IOleDocument::CreateView* to obtain the view it wishes to activate.

Argument	**Type**	**Description**
pViewToActivate	*IOleDocumentView* **	If non-NULL, specifies the view to bring forward. The caller does not call *AddRef* on this pointer before passing it in—if the receiver wishes to hold the pointer outside of this member function it must call *pViewToActivate->AddRef();*

Return Value	**Meaning**
S_OK	The container activated the view successfully.
E_OUTOFMEMORY	*pViewToActivate* is NULL and the container's call to *IOleDocument::CreateView* failed with E_OUTOFMEMORY.
E_FAIL	Another error occurred in either view creation or activation.

Comments:
This function must be completely implemented in a container; therefore E_NOTIMPL is not an acceptable return code.

The IOleDocumentView *Interface*

Each view of a document object is a sub-object that implements *IOleDocumentView* alongside *IOleInPlaceObject, IOleInPlaceActiveObject,* and other optional interfaces like *IPrint* and *IOleCommandTarget*. This interface provides all the necessary operations for a container to manipulate, manage, and activate a view.

IDL:

```
[
uuid(B722BCC6-4E68-101B-A2BC-00AA00404770)
    , object, pointer_default(unique)
]
interface IOleDocumentView : IUnknown
    {
    //import "unknwn.idl";

    HRESULT SetInPlaceSite([in] IOleInPlaceSite *pIPSite);
    HRESULT GetInPlaceSite([out] IOleInPlaceSite **ppIPSite);
    HRESULT GetDocument([out] IUnknown **ppunk);
    [input_sync] HRESULT SetRect([in] LPRECT prcView);
    HRESULT GetRect([out] LPRECT prcView);
    [input_sync] HRESULT SetRectComplex([in] LPRECT prcView
        , [in] LPRECT prcHScroll, [in] LPRECT prcVScroll
        , [in] LPRECT prcSizeBox);
    HRESULT Show ([in] BOOL fShow);
    HRESULT UIActivate([in] BOOL fUIActivate);
    HRESULT Open(void);
    HRESULT CloseView([in] DWORD dwReserved);
    HRESULT SaveViewState([in] IStream *pstm);
    HRESULT ApplyViewState([in] IStream *pstm);
    HRESULT Clone([in] IOleInPlaceSite *pIPSiteNew
        , [out] IOleDocumentView **ppViewNew);
    }
```

The members *SetInPlaceSite* and *GetInPlaceSite* manage the *IOleInPlaceSite* interface pointer for the container's view site associated with this view. The semantics of *SetInPlaceSite* are encompassed in the *pIPSite* argument of *IOleDocument::CreateView.*

GetDocument provides access to the *IUnknown* pointer of the document object that owns this view.

The *SetRect* and *GetRect* members manage the simple rectangle that the view will occupy in the container. *SetRectComplex* allows the container to specify not only the simple rectangle but also the spaces that should be occupied by the view's scrollbars and size box. An view specifies whether it understands *SetRectComplex* through the DOCMISC_SUPPORTCOMPLEXRECTANGLES status bit (see *IOleDocument::GetMiscStatus*).

The view's visual state is managed through the pair *Show* and *UIActivate* as well as *Open*. *Show* instructs the view to activate or deactivate itself in-place; when the view is active, *UIActivate* instructs the view to activate or deactivate its user interface elements such as menus, toolbars, and accelerators. *Show* and *UIActivate* in this interface are thus equivalents of the *IOleInPlaceObject* members of *InPlaceActivate, InPlaceDeactivate, UIActivate, and UIDeactivate* that are used for control of an in-place embedding.

The *Open* member, on the other hand, works with activation in a separate window (as happens with embeddings in OLE Documents when in-place is not supported). DocObjects marked with DOCMISC_CANTOPENEDIT (see *IOleDocument::GetMiscStatus*) do not support this form of activation. If support is present, however, *Open* instructs the view to activate in a separate window similar to *IOleObject::DoVerb(OLEIVERB_OPEN)*. At this point *Show* instructs the view to show and hide this window.

In all cases, *CloseView* instructs the view to deactivate the view, destroying any separate window and releasing the view site pointer passed previously to *IOleDocumentView::SetInPlaceSite.* This functionality is similar to that described for *IOleObject::Close*.

A view's internal state can be saved to a stream through *SaveViewState* and later reloaded from a stream with the same contents through *ApplyViewState*. The semantics of *ApplyViewState* are encompassed in the *pstm* argument of *IOleDocument::CreateView*.

Finally, a container can create a duplicate view object to the current one with *Clone*.

IOleDocumentView::SetInPlaceSite

HRESULT IOleDocumentView::SetInPlaceSite([in] IOleInPlaceSite *pIPSite)

Associates a view site object with this view. If this member is called and the view already has an associated view site, the view must first deactivate itself in that site, release that site, then remember the new pointer if that pointer is non-NULL (save the value and call *AddRef* on the pointer). The container will tell the view when to activate itself in the new site.

Argument	Type	Description
pIPSite	*IOleInPlaceSite* *	The interface pointer of the site to associate with this view. Can be NULL in which case the view loses all association with the container.

Return Value	Meaning
S_OK	The site was successfully associated (or disassociated if *pIPSite* is NULL)
E_FAIL	Another error occurred.

Comments:
This function must be completely implemented in a view; therefore E_NOTIMPL is not an acceptable return code.

IOleDocumentView::GetInPlaceSite

HRESULT IOleDocumentView::GetInPlaceSite([out] IOleInPlaceSite **ppIPSite)

Returns the most recent *IOleInPlaceSite* pointer passed to *SetInPlaceSite*, or NULL if *SetInPlaceSite* has not yet been called. The view will call *AddRef* on this pointer before returning it, thus the caller must later call *Release.*

Argument	Type	Description
ppIPSite	*IOleInPlaceSite* **	The address in which to return the current view site interface pointer associated with this view object. The caller becomes responsible for this pointer.

Return Value	Meaning
S_OK	The site was successfully returned. The caller must call *Release* through this pointer when it is no longer needed.
E_FAIL	Another error occurred.

Comments:
This function must be completely implemented in a view; therefore E_NOTIMPL is not an acceptable return code.

IOleDocumentView::GetDocument

HRESULT IOleDocumentView::GetDocument([out] IUnknown **ppunk)

Returns the *IUnknown* interface pointer of the document object that owns this view. As a document owning the view must always exist, this function will always succeed, calling *AddRef* on the pointer stored in **ppunk* before returning.

Argument	Type	Description
ppunk	*IUnknown* **	The address in which to return the *IUnknown* pointer of the document object that owns this view. The caller becomes responsible for this pointer.

Return Value	Meaning
S_OK	The document object's interface pointer was successfully returned. This is the only valid return code for this function.

IOleDocumentView::SetRect

[input_sync] HRESULT IOleDocumentView::SetRect([in] LPRECT prcView)

Sets the rectangular coordinates of the view port in the client coordinates of the view window (the window is obtained through *IOleInPlaceSite::GetWindow*). The view must resize itself to view the new coordinates.

This member function is defined with the [input_sync] attribute, hence the implementing object cannot yield or make another non input_sync RPC call while executing this method.

Argument	Type	Description
prcView	*LPRECT*	Points to a RECT structure containing the coordinates of the view port in the client coordinates of the view window.

Return Value	Meaning
S_OK	The view was successfully resized to the rectangle.
E_FAIL	Some other critical error occurred that prevented resizing to occur.

Comments:
This function must be completely implemented in a view; therefore E_NOTIMPL is not an acceptable return code.

IOleDocumentView::GetRect

HRESULT IOleDocumentView::GetRect([out] LPRECT prcView)

Returns the rectangular coordinates of the view port in the client coordinates of the view window, as was last specified through *IOleDocumentView::SetRect* or *IOleDocumentView::SetRectComplex* .

Argument	Type	Description
prcView	*LPRECT*	Points to a RECT structure to receive the current view coordinates.

Return Value	Meaning
S_OK	The view was successfully resized to the rectangle.
E_UNEXPECTED	This view has not yet seen a call to *IOleDocumentView::SetRect* or *IOleDocumentView::SetRectComplex*, thus it has no rectangle to return.

Comments:
This function must be completely implemented in a view; therefore E_NOTIMPL is not an acceptable return code.

IOleDocumentView::SetRectComplex

[input_sync] HRESULT IOleDocumentView::SetRectComplex([in] LPRECT prcView, [in] LPRECT prcHScroll, [in] LPRECT prcVScroll, [in] LPRECT prcSizeBox)

Sets the rectangular coordinates of the view port, horizontal and vertical scroll bars, and the size box. This method typically gets used by the view frames which have a workbook metaphor. However, not all DocObjects support these detailed specifications; those that do mark themselves with DOCMISC_SUPPORTCOMPLEXRECTANGLES as described in *IOleDocument::GetMiscStatus.* DocObjects that do not support this member can return E_NOTIMPL.

Within this member, the view should resize itself according to *prcView* and fit its scrollbars and size box to the areas described in *prcHScroll, prcVScroll,* and *prcSizeBox,* respectively.

This member function is defined with the [input_sync] attribute, hence the implementing object cannot yield or make another non input_sync RPC call while executing this method.

Argument	Type	Description
prcView	[in] LPRECT	Points to a RECT structure containing the coordinates of the view port in client coordinates of the view window.
prcHScroll	[in] LPRECT	Points to a RECT structure containing the coordinates of the horizontal scroll bar in client coordinates of the view window.
prcVScroll	[in] LPRECT	Points to a RECT structure containing the coordinates of the vertical scroll bar in client coordinates of the view window.
prcSizeBox	[in] LPRECT	Points to a RECT structure containing the coordinates of the size box in client coordinates of the view window.

Return Value	Meaning
S_OK	The view was successfully resized to the rectangle.
E_NOTIMPL	The document object that owns this view does not support complex rectangle specifications.
E_FAIL	Some other critical error occurred that prevented resizing of the view or placement of the scrollbars and size box.

IOleDocumentView::Show

HRESULT Show ([in] BOOL fShow)

Instructs a view to in-place activate or in-place deactivate itself as described in the following pseudo-code:

```
if (fShow)
    {
    in-place activate the view but do not UI activate it.
    Show the view window.
    {
else
    {
    call IOleDocumentView::UIActivate(FALSE) on this view
    Hide the view window
    }
```

Argument	Type	Description
fShow	BOOL	TRUE instructs the view to show itself, FALSE instructs the view to hide itself.

Return Value	Meaning
S_OK	The view was successfully shown or hidden.
E_OUTOFMEMORY	There was not enough memory to activate or hide the view.
E_FAIL	Some other critical error occurred that prevented activation or hiding.
E_UNEXPECTED	This member was called before a call to *IOleDocumentView::SetInPlaceSite.*

Comments:
All views of a document object must at least support the in-place activation mode, therefore E_NOTIMPL is not allowed as a return value.

IOleDocumentView::UIActivate

HRESULT IOleDocumentView::UIActivate([in] BOOL fUIActivate)

Instructs the view to activate or deactivate its user interface elements (menus, toolbars, accelerators) as described in the following pseudo-code:

```
if (fActivate)
    {
    UI activate the view (do menu merging, show frame level tools, process accelerators)
    Take focus, and bring the view window forward.
    }
else
    call IOleInPlaceObject::UIDeactivate() on this view
```

The view may, and should, participate in extended Help menu merging if it desires.

Argument	Type	Description
fActivate	BOOL	TRUE instructs the view to activate its UI, FALSE instructs the view to deactivate its UI.

Return Value	Meaning
S_OK	The view's UI was successfully activated or deactivated.
E_OUTOFMEMORY	There was not enough memory to activate the UI elements.
E_FAIL	Some other error occurred that prevented success.
E_UNEXPECTED	This member was called before a call to *IOleDocumentView::SetInPlaceSite.*

Comments:
All views of a document object must at least support the in-place activation mode, therefore E_NOTIMPL is not allowed as a return value.

IOleDocumentView::Open

HRESULT IOleDocumentView::Open(void)

Asks the view to display itself in a separate popup window with semantics equivalent to *IOleObject:;DoVerb(OLEIVERB_OPEN)*. If the document object specified DOCMISC_CANTOPENEDIT through *IOleDocument::GetMiscStatus,* this call can return E_NOTIMPL. Otherwise implementation generally calls the view's own *IOleInPlaceObject::InPlaceDeactivate* after which the view shows its separate popup window and brings that window to the foreground.

Contrary to the normal in-place deactivation sequence for OLE Documents, a view *continues to hold* the *IOleInPlaceSite* pointer that it obtained in *IOleDocumentView::SetInPlaceSite* (likewise the view site continues to hold the view's interface pointers, obviously). This pointer is only released through *IOleDocumentView::SetInPlaceSite(NULL)* or in *IOleDocumentView::CloseView*.

When the user closes the view's window (via File.Close), then the view should not shut itself down. Instead it should call *pIPSite->OnInPlaceActivate.* The view site then decides whether to UI activate the view at that time or at a later time.

When the container decides that the view window is no longer needed, it calls *IOleDocumentView::CloseView.* The view uses that call to determine when to release the site pointer and destroy the window.

If is legal for the container to call *IOleDocumentView::Show(FALSE)* when the view is in this Open mode. In this case the view hides its window. Similarly, *IOleDocumentView::Show(TRUE)* instructs the view to show the window again and bring it to the foreground.

Argument	Type	Description
NA	NA	NA

Return Value	Meaning
S_OK	The view successfully created its separate window.
E_OUTOFMEMORY	There was not enough memory to activate the view in a separate window.
E_FAIL	Some other error occurred that prevented success.
E_NOTIMPL	The document object that owns this view does not support separate window activation.
E_UNEXPECTED	This member was called before a call to *IOleDocumentView::SetInPlaceSite.*

IOleDocumentView::CloseView

HRESULT IOleDocumentView::CloseView([in] DWORD dwReserved)

Asks the view to close down and release its *IOleInPlaceSite* pointer obtained in *IOleDocumentView::SetInPlaceSite*. The container must call this method before it wants to delete the view (that is, release its last reference to the view). In general, implementation of this member will call *IOleDocumentView::Show (FALSE)* to hide the view if it's not already, then call *IOleDocumentView::SetInPlaceSite(NULL)* to deactivate itself and release the view site pointer.

Argument	Type	Description
dwReserved	*DWORD*	Reserved. Must be zero.

Return Value	Meaning
S_OK	The view successfully closed itself.

Comments:
Because *CloseView* is called when the container wishes to completely shut down the view, this member must be implemented and has no reason to fail.

IOleDocumentView::SaveViewState

HRESULT IOleDocumentView::SaveViewState([in] IStream *pstm)

Instructs the view to save its state into the given stream, where the state includes properties like the view type, zoom factor, insertion point, and so on. The container typically calls this function before deactivating the view. The stream can then later be used to reinitialize a view of the same document to this saved state through *IOleDocumentView::ApplyViewState.*

The view must write its CLSID as the first element in the stream according to the rules that apply to *IPersistStream.* Any cross-platform file format compatibility issues that apply to the document's storage representation also apply to this context.

Argument	Type	Description
pstm	[in] IStream *	ask the view to save the view state into this stream.

Argument	Type	Description
pstm	*IStream **	The stream in which the view should save its state.

Return Value	Meaning
S_OK	The view successfully saved its state to the stream.
E_POINTER	The value in *pstm* is NULL.
E_NOTIMPL	This view has no meaningful state to save; this should be a rare case as most views will have at least some information.

IOleDocumentView::ApplyViewState

HRESULT IOleDocumentView::ApplyViewState([in] IStream *pstm)

Instructs a view to reinitialize itself according to the data in a stream that was previously written through *IOleDocumentView::SaveViewState.* Typically this function is called when the view is being displayed for first time after its instantiation. It is the responsibility of the view to validate the data in the view stream as the container does not attempt to interpret view state stream data in any way.

Argument	Type	Description
pstm	*IStream **	The stream from which the view should load its state.

Return Value	Meaning
S_OK	The view successfully loaded its state from the stream.
E_POINTER	The value in *pstm* is NULL.
E_NOTIMPL	This view has no meaningful state that it would load; this should be a rare case as most views will have at least some information.

IOleDocumentView::Clone

HRESULT IOleDocumentView::Clone([in] IOleInPlaceSite *pIPSiteNew, [out] IOleDocumentView **ppViewNew)

Creates a duplicate view object with an identical internal state to the current view. This is useful for creating a new view with a different view port and view site but with the same view context as the view being cloned. Typically this will be used to implement the “Window-New window” functionality.

Argument	Type	Description
pipsiteClone	[in] IOleInPlaceSite *	pointer to the in-place site for the clone
ppviewClone	[out] IOleDocumentView **	the location where the pointer to the new view should be returned.

Argument	Type	Description
pIPSiteNew	*IOleInPlaceSite **	The *IOleInPlaceSite* pointer of the view site to associate with the clone. The view being cloned should pass this to the new view’s *IOleDocumentView::SetInPlaceSite* member. This can be NULL in which case the caller is responsible for calling *SetInPlaceSite* on this new view directly.
ppViewNew	*IOleDocumentView **	The address of the variable to receive the pointer to the new view’s *IOleDocumentView* interface. The caller is responsible for this pointer any must call *Release* through it when it is no longer needed.

Return Value	Meaning
S_OK	The view successfully cloned. The caller is responsible for the pointer in **ppViewNew.*
E_POINTER	The value in *ppViewNew* is NULL.
E_FAIL	The document object only supports one view. E_NOTIMPL can also

be used.

The IPrint *Interface*

Any object that wishes to support programmatic printing can implement the *IPrint* interface. Through this interface a caller can tell the object to print, set the initial page number (for printing multiple documents together), and retrieve print-related information from the object:

IDL:

```
[
uuid(B722BCC9-4E68-101B-A2BC-00AA00404770)
    , object, pointer_default(unique)
]
interface IPrint : IUnknown
    {
    typedef [unique] IPrint *LPPRINT;

    typedef enum
        {
        PRINTFLAG_MAYBOTHERUSER        = 1,
        PRINTFLAG_PROMPTUSER           = 2,
        PRINTFLAG_USERMAYCHANGEPRINTER = 4,
        PRINTFLAG_RECOMPOSETODEVICE    = 8,
        PRINTFLAG_DONTACTUALLYPRINT    = 16,
        PRINTFLAG_FORCEPROPERTIES      = 32,
        PRINTFLAG_PRINTTOFILE          = 64
        } PRINTFLAG;

    typedef struct tagPAGERANGE
        {
        LONG nFromPage;
        LONG nToPage;
        } PAGERANGE;

    typedef struct tagPAGESET
        {
        ULONG cbStruct;
        BOOL  fOddPages;
        BOOL  fEvenPages;
        ULONG cPageRange;
        [size_is(cPageRange)] PAGERANGE rgPages[];
        } PAGESET;


    HRESULT SetInitialPageNum([in] LONG nFirstPage);
    HRESULT GetPageInfo([out] LONG *pnFirstPage, [out] LONG *pcPages);
    HRESULT Print([in] DWORD grfFlags, [in,out] DVTARGETDEVICE **pptd
        , [in,out] PAGESET **ppPageSet
        , [unique][in,out] STGMEDIUM *pstgmOptions
        , [in] IContinueCallback *pcallback, [in] LONG nFirstPage
        , [out] LONG *pcPagesPrinted, [out] LONG *pnLastPage);
    };

#define PAGESET_TOLASTPAGE ((WORD)(-1L))
```

The structures of this interface will be described first, followed by the member functions.

PAGERANGE Structure

Identifies a single range of pages. Note that is *nFromPage* is greater than *nToPage,* the pages are printed in the reverse order.

Member	Type	Description
nFromPage	*LONG*	The first page to print. The first page of a document is 1.
nToPage	*LONG*	The last page to print. A special value of PAGESET_TOLASTPAGE indicates that all the remaining pages should be printed.

PAGESET Structure

Identifies a series of page-ranges and optionally identifies only the even or odd pages as part of this PAGESET.

Member	Type	Description
cbStruct	*ULONG*	The number of bytes in this instance of the PAGESET structure. Must be a multiple of 4.
fOddPages	*BOOL*	If true, then only the odd-numbered pages in the page-set indicated by *rgPages* are to be printed.
fEvenPages	*BOOL*	If true, then only the even-numbered pages in the page-set indicated by *rgPages* are to be printed.
cPageRange	*ULONG*	The number of page-range pairs specified in *rgPages*.
rgPages	*PAGERANGE **	Specifies the pages to be printed. The page ranges must be sorted in increasing order and non-overlapping. It is an error to attempt to print a page which does not exist.

PRINTFLAG Enumeration

A combination of values from PRINTFLAG is passed in as *grfFlags* to *IPrint::Print*.

Value	Description
PRINTFLAG_MAYBOTHERUSER	Specifies whether any interaction is permitted with the user at all. Unless this flag is set, no part of the printing process may interact with the user.
PRINTFLAG_PROMPTUSER	Only valid if PRINTFLAG_MAYBOTHERUSER is specified. Prompt the user for job-specific printing options using the normal print dialog for the object. Support for this option is required.
PRINTFLAG_USERMAYCHANGEPRINTER	Only valid if PRINTFLAG_PROMPTUSER is specified. Indicates that the user may change the printer to be printed to; in the absence of this flag, the user must print on the printer provided.
PRINTFLAG_RECOMPOSETODEVICE	Indicates that the object should attempt to recompose itself to the indicated target device. In the absence of this flag, the object should retain any existing compositional-device association that it may happen to presently have if at all possible.
PRINTFLAG_DONTACTUALLYPRINT	Carry out any indicated user-prompting and object-recomposing actions as indicated, but don't actually carry out the printing operation.
PRINTFLAG_PRINTTOFILE	The object should print to the file, name of which is passed through "portname" field of DVTARGETDEVICE.

IPrint::SetInitialPageNum

HRESULT IPrint::SetInitialPageNum([in] LONG nFirstPage)

Attempt to set the number of the first page of this document. Note that setting a negative first page number is legal: this may be useful in printing a portion of the document with offset page numbers from what it would normally print. Note also that not all implementations permit the initial page number to be set, as some implementations simply lack the information as to how this page information should be reflected in the final output.

Argument	Type	Description
nFirstPage	*LONG*	The desired first page number.

Return Value	Meaning
S_OK	The first page was set as requested.
E_FAIL	The first page could not be set to the indicated value.
E_UNEXPECTED	An unknown error occurred.

IPrint::GetPageInfo

```
HRESULT IPrint::GetPageInfo([out] LONG *nFirstPage, [out] LONG *pcPages)
```

Return information about the pages in the document.

Argument	Type	Description
pnFirstPage	*LONG**	Location to return the page number of the first page. May be NULL, indicating the caller doesn't need this number. If *IPrint::SetInitialPageNum* has been called, this should contain the same value passed to that method. Otherwise the value is the document's internal first page number.
pcPages	*LONG**	Location to return the total number of pages in this document. May be NULL, indicating the caller doesn't need this number.

Return Value	Meaning
S_OK	Success.
E_UNEXPECTED	An unexpected error occurred.

IPrint::Print

```
HRESULT IPrint::Print([in] DWORD grfFlags, [in,out] DVTARGETDEVICE **pptd
    , [in,out] PAGESET **pppageset, [unique][in,out] STGMEDIUM *pstgmOptions
    , [in] IContinueCallback *pcallback, [in] LONG nFirstPage, [out] LONG *pcPagesPrinted
    , [out] LONG *pnLastPage)
```

Print this object on the printer indicated by the DVTARGETDEVICE structure in *ptd*. The DEVMODE in the target device indicates whole-job printer-specific options, such as number of copies, paper size, print quality, etc. It may or may not also contain orientation information in the *dmOrientation* field (this is indicated in the *dmFields* field). If present, then this paper orientation should be used; if absent, then natural orientation as determined by the object content is to be used.

Due to the possibility of user input, the parameters *pptd* and *ppPageSet* are both [in,out] structures. In the absence of user interaction (that is, without PRINTFLAG_PROMPTUSER), both the target device and the page set will necessarily be the same on input and output. However, if the user is prompted for print options, then the object returns target device and page set information appropriate to what the user has actually chosen during interaction.

ppstgmOptions is an [in,out] parameter. On exit, the object should return through **ppstgmOptions* any object-specific information that it would need to reproduce this exact print job. Examples might include whether the user selected "sheet, notes, or both" in a spreadsheet application. The data returned is in the format of a serialized property set. The returned data can usually only be usefully used by passing it back in a subsequent call to the same object; however, that call may have different user interaction flags, different target device, etc. Thus, the caller can cause the exact same document to be printed multiple times in slightly different printing contexts.

Argument	Type	Description
grfFlags	*DWORD*	A bit field whose values are taken from the enumeration PRINTFLAG.
pptd	*DVTARGETDEVICE***	The target device on which the printing is to occur.
ppPageSet	*PAGESET***	Indicates which pages are to be printed.
ppstgmOptions	*STGMEDIUM***	Contains object-specific printing options in the form of a serialized OLE property set. May be NULL in one or both directions.
pCallback	*IContinueCallback**	A callback interface which is to be periodically polled at human-response speeds to determine whether printing should be abandoned. May be NULL.

nFirstPage	*LONG*	The starting page number to print. This overrides any value previously passed to *IPrint::SetInitialPageNum*.
pcPagesPrinted	*LONG**	The place at which the object is to return the actual number of pages that were successfully printed.
pnLastPage	*LONG**	The place at which the object is to return the last legal page number.

Return Value	**Meaning**
S_OK	Success
PRINT_E_CANCELLED	The print process was canceled. **pcPagesPrinted* indicates the number of pages that were in fact successfully printed before this occurred.
PRINT_E_NOSUCHPAGE	An attempt has been made to print a page which does not exist.
E_UNEXPECTED	An unexpected error occurred.

The IContinueCallback *Interface*

This interface is a generic callback mechanism for interruptible processes that should periodically ask an object with this interface whether to continue the process.

IDL:

```
[
uuid(B722BCCA-4E68-101B-A2BC-00AA00404770)
    , object, pointer_default(unique)
]
interface IContinueCallback : IUnknown
    {
    HRESULT FContinue(void);
    HRESULT FContinuePrinting([in] LONG nCntPrinted
        , [in] LONG nCurPage, [unique][in] wchar_t *pszPrintStatus);
    }
```

The *FContinue* function is a generic continuation request. *FContinuePrinting* carries extra information pertaining to a printing process and is used in the context of *IPrint.*

IContinueCallback::FContinue

HRESULT IContinueCallback::FContinue(void)

Answer as to whether a given generic operation should continue.

Argument	Type	Description
NA	NA	NA

Return Value	Meaning
S_OK	Continue the operation.
S_FALSE	Cancel the operation as soon as possible

IContinueCallback::FContinuePrinting

HRESULT IContinueCallback::FContinuePrinting(cPagesPrinted, nCurrentPage, wszPrintStatus)

Answer as to whether a given lengthy printing operation should continue. Implementations of *IPrint* call back on this method at periodic intervals during the printing process. The *IPrint* implementation should call back at least after printing each page, so that its client can display useful visual feedback to the user. Further, the implementation can legally call back multiple times with the same *cPagesPrinted* and *nCurrentPage* values; this is sometimes useful when a page being printed is complex and it is appropriate to give the user a chance to cancel mid-page.

Argument	Type	Description
cPagesPrinted	*LONG*	The total number of pages printed so far.
nCurrentPage	*LONG*	The page number of the current page being printed.
pszPrintStatus	*LPOLESTR*	Status message about the print job which the recipient of this call may choose to display to the user. May be NULL.

Return Value	Meaning
S_OK	Continue printing.
S_FALSE	Cancel the print job as soon as possible
E_UNEXPECTED	An unknown error occurred.

The IOleCommandTarget *Interface*

The command dispatch interface *IOleCommandTarget* defines a simple and extensible mechanism to query and execute commands which are defined as integer identifiers in a group. The group is identified itself with a GUID. The interface allows a caller to both query for support of commands within a group as well as to instruct the object to execute those commands.

IDL:

```
[
uuid(B722BCCB-4E68-101B-A2BC-00AA00404770)
    , object, pointer_default(unique)
]
interface IOleCommandTarget : IUnknown
    {
    typedef [unique] IOleCommandTarget *LPOLECOMMANDTARGET;

    typedef enum
        {
        OLECMDF_SUPPORTED   = 0x00000001,
        OLECMDF_ENABLED     = 0x00000002,
        OLECMDF_LATCHED     = 0x00000004,
        OLECMDF_NINCHED     = 0x00000008
        } OLECMDF;

    typedef struct _tagOLECMD
        {
        ULONG cmdID;
        DWORD cmdf;
        } OLECMD;

    typedef enum
        {
        OLECMDTEXTF_NONE   = 0,
        OLECMDTEXTF_NAME   = 1,
        OLECMDTEXTF_STATUS = 2
        } OLECMDTEXTF;

    typedef struct  _tagOLECMDTEXT
        {
        DWORD cmdtextf;
        ULONG cwActual;
        ULONG cwBuf;
        [size_is(cwBuf)] wchar_t rgwz[];
        } OLECMDTEXT;

    typedef enum
        {
        OLECMDEXECOPT_DODEFAULT      = 0,
        OLECMDEXECOPT_PROMPTUSER     = 1,
        OLECMDEXECOPT_DONTPROMPTUSER = 2,
        OLECMDEXECOPT_SHOWHELP       = 3
        } OLECMDEXECOPT;

    typedef enum
        {
        OLECMDID_OPEN           = 1,
        OLECMDID_NEW            = 2,
        OLECMDID_SAVE           = 3,
        OLECMDID_SAVEAS         = 4,
        OLECMDID_SAVECOPYAS     = 5,
        OLECMDID_PRINT          = 6,
        OLECMDID_PRINTPREVIEW   = 7,
        OLECMDID_PAGESETUP      = 8,
        OLECMDID_SPELL          = 9,
        OLECMDID_PROPERTIES     = 10,
        OLECMDID_CUT            = 11,
        OLECMDID_COPY           = 12,
        OLECMDID_PASTE          = 13,
        OLECMDID_PASTESPECIAL   = 14,
        OLECMDID_UNDO           = 15,
        OLECMDID_REDO           = 16,
        OLECMDID_SELECTALL      = 17,
        OLECMDID_CLEARSELECTION = 18,
        OLECMDID_ZOOM           = 19,
        OLECMDID_GETZOOMRANGE   = 20,
        OLECMDID_UPDATECOMMANDS = 21,
        OLECMDID_REFRESH        = 22,
        OLECMDID_STOP           = 23,
        OLECMDID_HIDETOOLBARS   = 24,
        OLECMDID_SETPROGRESSMAX = 25,
        OLECMDID_SETPROGRESSPOS = 26,
        OLECMDID_SETPROGRESSTEXT= 27,
        OLECMDID_SETTITLE       = 28
        } OLECMDID;

    [input_sync] HRESULT QueryStatus([unique][in] const GUID *pguidCmdGroup
        , [in] ULONG cCmds, [in,out][size_is(cCmds)] OLECMD *prgCmds
        , [unique][in,out] OLECMDTEXT *pCmdText);
    HRESULT Exec([unique][in] const GUID *pguidCmdGroup
        , [in] DWORD nCmdID, [in] DWORD nCmdExecOpt
        , [unique][in] VARIANTARG *pvaIn
        , [unique][in,out] VARIANTARG *pvaOut);
    };
```

OLECMDF Enumeration

Values from the OLECMDF enumeration are used to fill the value of the *cmdf* field in OLECMD structures as passed to *IOleCommandTarget::QueryStatus*.

Flag	Description
OLECMDF_SUPPORTED	The command is supported by this object.
OLECMDF_ENABLED	The command is available and enabled.
OLECMDF_LATCHED	The command is an on-off toggle and is currently on.
OLECMDF_NINCHED	The command is an on-off toggle but the state cannot be determined because the attribute of this command is found in both on and off states in the relevant selection. This state corresponds to an "indeterminate" state of a 3-state checkbox, for example.

OLECMD Structure

The OLECMD structure is used to associate command flags from the OLECMDF enumeration with a command identifier through *IOleCommandTarget::QueryStatus*.

Field	Type	Description
cmdID	ULONG	A command identifier.
cmdf	DWORD	Flags associated with *cmdID* taken from the OLECMDF enumeration.

OLECMDTEXTF Enumeration

Values from the OLECMDTEXTF enumeration are used to describe what a command target object should store in the OLECMDTEXT structure passed to *IOleCommandTarget::QueryStatus*. One value from this enumeration is stored in the *cmdtextf* of the structure to indicate the desired information.

Flag	Description
OLECMDTEXTF_NONE	No extra information is requested.
OLECMDTEXTF_NAME	The object should return the localized name of the command.
OLECMDTEXTF_STATUS	The object should return a localized status string for the command.

OLECMDTEXT Structure

Used to return a text name or a status string for a single command identifier when used with *IOleCommandTarget::QueryStatus*.

Field	Type	Description
cmdtextf	*DWORD*	Filled on input; a value from the OLECMDTEXTF enumeration describing the information the caller wishes to receive in return.
cwActual	*ULONG*	Filled on output; the number of characters actually written into the *rgwz* buffer before the function returns.
cwBuf	*ULONG*	Filled on input; the size of the string buffer in *cwBuf.*
rgwz	*wchar_t*	A caller allocated array of wide characters to receive the string on output.

OLECMDEXECOPT Enumeration

Flag	Description
OLECMDEXECOPT_PROMPTUSER	Execute the command after taking user input.
OLECMDEXECOPT_DONTPROMPTUSER	Execute the command without prompting the user (for example, clicking on the Print toolbar button, causes the document to be immediately printed without requiring the user input).
OLECMDEXECOPT_DODEFAULT	Caller is not sure whether the user should be prompted or not.
OLECMDEXECOPT_SHOWHELP	Object should show help for the corresponding command and not execute.

OLECMDID Enumeration

See below under “Standard Command List.”

IOleCommandTarget::QueryStatus

```
[input_sync] HRESULT QueryStatus([unique][in] const GUID *pguidCmdGroup, [in] ULONG cCmds,
    [in,out][size_is(cCmds)] OLECMD *prgCmds , [unique][in,out] OLECMDTEXT *pCmdText);
```

Queries the object for the status of one or more commands, typically used in WM_INITMENU or WM_INITMENUPOPUP messages, enabling the caller to disable those commands that would be routed to the object but that are not available. The caller passes an array of OLECMD structures in *prgCmds* that describe the commands of interest from the group specified in *pguidCmdGroup*, where each structure’s *cmdID* is set to a command identifier and the *cmdf* field is set to zero. The object receiving the call the fills the *cmdf* field for each command with bits taken from the OLECMDF enumeration to describe the status of each command.

The caller can also use this method to get the name or status text of a single command. The called object should first mark the command as described above. If the command is supported (OLECMDF_SUPPORTED) then the object should check the OLECMDTEXTF flags in the OLECMDTEXT structure. If the OLECMDFTEXF_NAME flag is specified, then the object should copy the localized name of the command (for example, “Open”, “Copy”, etc.) into the *rgwz* field of OLECMDTEXT, paying attention to the size specified by the *cwBuf* field in that same structure.

If, however, the caller specifies OLECMDFTEXTF_STATUS then the object should instead copy a localized status string for the command into the *rgwz* field. The status string is typically contextual, and it depends on the state of the command such as enabled/disabled. If the buffer is not big enough then the object should zero terminate the buffer. Whether the buffer is big enough or not the object must return the total actual size of the string(s), that he attempted to copy, via *cwActual* field.

If the command array contains more than one command, then the textual information should be returned for the first command in the command array that the object supports. Typically this functionality is used to show the status text of a command. Note that the caller can use a stack or global variable for *rgwz*, it not be dynamically allocated memory.

This member function is defined with the [input_sync] attribute, hence the implementing object cannot yield or make another non input_sync RPC call while executing this method.

Argument	Type	Description
pguidCmdGroup	*const GUID **	Unique identifier of the command group which can be NULL to specify the standard group. All the commands that are passed in the *rgCmds* array must belong to this group.
cCmds	*ULONG*	The number of commands in the *prgCmds* array.
prgCmds	*OLECMD **	An caller-allocated array of OLECMD structures where the *cmdID* fields of the structures initialized with the commands being queried.
pcmdText	*OLECMDTEXT **	Pointer to the structure in which to return name and/or status information. Can be NULL to indicate that the caller is not interested in such information.

Return Value	Meaning
S_OK	The command status as any optional strings were returned successfully.
E_POINTER	The *prgCmds* argument is NULL.
E_UNEXPECTED	An unexpected error occurred.
E_FAIL	An error occurred
OLECMDERR_E_UNKNOWNGROUP	*pguidCmdGroup* is non-NULL but does not specify a recognized command group.

Comments:
A command target must implement this function; therefore E_NOTIMPL is not a valid return code.

IOleCommandTarget::Exec

```
HRESULT Exec([unique][in] const GUID *pguidCmdGroup, [in] DWORD nCmdID
    , [in] DWORD nCmdExecOpt, [unique][in] VARIANTARG *pvaIn
    , [unique][in,out] VARIANTARG *pvaOut)
```

Executes a specified command or displays help for a command. As in the case of *IOleCommandTarget::QueryStatus,* the *pguidCmdGroup* and *nCmdID* arguments uniquely identify the command to invoke. The exact action to take is specified in *nCmdExecOpt* (see the OLECMDEXECOPT enumeration for more details).

Most of the commands take no arguments nor do they return any values. Hence, for majority of the commands the caller can pass NULLs for *pvaIn* and *pvaOut*. For the commands which expect one or more input value, the caller can declare and initialize a VARIANTARG variable and pass a pointer to that variable in *pvaIn.*[8] If the input to the command is a single value then the argument can be stored directly in the VARIANTARG and passed to the function. If the command expects multiple arguments then they must be packaged appropriately within the VARIANTARG using one of the supported types (such as *IDispatch, SAFEARRAY*, etc.).

Similarly, if a command returns one or more arguments the caller is expected to declare a VARIANTARG, initialize it to VT_EMPTY, and pass its address in *pvaOut.* If the command returns a single value then the object can store that value directly in *pvaOut.* If the command has multiple output values then it will package those in some way appropriate for the VARIANTARG.

Note that both *pvaIn* and *pvOut* are caller-allocated, thus stack variables are perfectly usable. For commands that take zero or one argument on input and return zero or one values, then no extra memory allocation is necessary.[9] the caller and callee can use stack variables.

The list of *in* and *out* arguments of a command and how they are packaged is unique to each command; such information should be documented with the specification of the command group (see the Zoom command later in this section). In the absence of any specific information the command is assumed to take no arguments and have no return value.

[8] VARIANTARG is defined in OLE Automation.
[9] Most of the types supported by VARIANTARG do not require memory allocation, few of the exceptions are SAFEARRAY and BSTR. For the complete list, see OLE documentation.

Argument	Type	Description
pguidCmdGroup	*const GUID **	Unique identifier of the command group which can be NULL to specify the standard group. The command passed in *nCmdID* must belong to this group.
nCmdID	*DWORD*	The command to execute which must be in the group specified with *pguidCmdGroup.*
nCmdExecOpt	*DWORD*	One or more values from the OLECMDEXECOPT enumeration describing how the object should execute the command.
pvaIn	*VARIANTARG **	Pointer to a VARIANTARG containing input arguments. Can be NULL.
pvaOut	*VARIANTARG **	Pointer to a VARIANTARG to receive the output return values. Can be NULL.

Return Value	Meaning
S_OK	The command was executed successfully.
E_UNEXPECTED	An unexpected error occurred.
E_FAIL	An error occurred
OLECMDERR_E_UNKNOWNGROUP	*pguidCmdGroup* is non-NULL but does not specify a recognized command group.
OLECMDERR_E_NOTSUPPORTED	The *nCmdID* argument is not recognized as a valid command in the group identified with *pguidCmdGroup*.
OLECMDERR_DISABLED	The command identified with *nCmdID* is currently disabled and cannot be executed.
OLECMDERR_NOHELP	The caller has asked for help on the command identified by *nCmdID* but no help is available.
OLECMDERR_CANCELED	The user canceled the execution of the action.

Comments:
A command target must implement this function; therefore E_NOTIMPL is not a valid return code.

Standard Command List

Following is the list of standard commands that have been defined by Office 95 which are identified as the group with a NULL GUID (that is, *pguidCmdGroup* as passed to *IOleCommandTarget::Exec* is NULL; this is not the same as GUID_NULL, which is *not* used in this context).

Identifier	Description
OLECMDID_OPEN	File Open
OLECMDID_NEW	File New
OLECMDID_SAVE	File Save
OLECMDID_SAVEAS	File Save As
OLECMDID_SAVECOPYAS	File Save Copy As
OLECMDID_PRINT	File Print
OLECMDID_PRINTPREVIEW	File Print Preview
OLECMDID_PAGESETUP	File Page Setup
OLECMDID_SPELL	Tools Spelling
OLECMDID_PROPERTIES	File Properties
OLECMDID_CUT	Edit Cut
OLECMDID_COPY	Edit Copy
OLECMDID_PASTE	Edit Paste
OLECMDID_PASTESPECIAL	Edit Paste Special
OLECMDID_UNDO	Edit Undo
OLECMDID_REDO	Edit Redo
OLECMDID_SELECTALL	Edit Select All
OLECMDID_CLEARSELECTION	Edit Clear
OLECMDID_ZOOM	View Zoom (see below for details)
OLECMDID_GETZOOMRANGE	Retrieves zoom range applicable to View Zoom (see below for details)
OLECMDID_UPDATECOMMANDS	Informs the receiver of state changes at which time the receiver (usually the frame but not limited to it) can query the status of the commands at a convenient time.
OLECMDID_REFRESH	Instructs the receiver to refresh its display.
OLECMDID_STOP	Stop all current processing
OLECMDID_HIDETOOLBARS	View Toolbar. Hide all toolbars owned by the receiving object (usually a Document Object).
OLECMDID_SETPROGRESSMAX	Sets the maximum value of a progress indicator if one if owned by the receiving object (usually a frame).
OLECMDID_SETPROGRESSPOS	Sets the current value of a progress indicator if one if owned by the receiving object (usually a frame).
OLECMDID_SETPROGRESSTEXT	Sets the text contained inside a progress indicator if one if owned by the receiving object (usually a frame). If the receiver currently has no progress indicator, this text should be displayed in the status bar, if one exists, as with IOleInPlaceFrame::SetStatusText.
OLECMDID_SETTITLE	Sets the title bar text of the receiving object (usually the frame).

The Zoom Commands

Under normal OLE Documents functionality, an object being edited in-place disabled its Zoom control on its toolbar and its View.Zoom menu are disabled, because logically the Zoom applies to the container document and not the object. With the OLECMDID_ZOOM and OLECMDID_GETZOOMRANGE commands in the standard set for *IOleCommandTarget,* the object now has a means through which it can notify the container's frame object (the one with

IOleInPlaceFrame as well as *IOleCommandTarget*, if supported) as well as retrieve the zoom range that it should display in its user interface.

OLECMDID_ZOOM

The OLECMDID_ZOOM command takes one LONG argument as input and returns one LONG argument on output. This command is used for three purposes:

- *To query the current zoom value* the caller passes OLECMDEXECOPT_DONTPROMPTUSER as the execute option in *nCmdExecOpt* and NULL for *pvIn*. The object returns the current zoom value in *pvaOut*. When the object goes UI active, it retrieves the current zoom value from the container's frame object using this same mechanism and updates its zoom control with the returned value.
- *To display the Zoom dialog box* the caller passes OLECMDEXECOPT_PROMPTUSER in *nCmdExecOpt*. The caller can optionally pass the initial value for the dialog box through *pvaIn*, otherwise *pvaIn* must be NULL. If the user presses CANCEL, the object returns OLECMDERR_E_CANCELED; if the user presses OK, then the object returns the user selected value in *pvaOut.* When user selects the View.Zoom menu item, the object calls container's frame object in the same manner. The container then zooms the document to the user selected value, and the object updates its Zoom control with that value.
- *To set a Zoom value* the caller passes OLECMDEXECOPT_DONTPROMPTUSER in *nCmdExecOpt* and passes the zoom value to apply through *pvaIn.* The object validates and normalizes the new value and returns the validated value in *pvaOut.* When the user selects a new zoom value (using the Zoom control on the toolbar for instance) the object calls the container's frame object in this manner. The container zooms the document to the normalized value and object updates the Zoom control with that value.

OLECMDID_GETZOOMRANGE

The OLECMDID_GETZOOMRANGE command is used to determine the range of valid zoom values from a command target object. The caller passes MSOCMDEXECOPT_DONTPROMPTUSER in *nCmdExecOpt* and NULL for *pvaIn*. The object returns its zoom range as a DWORD in *pvaOut* where the HIWORD contains the maximum zoom value and the LOWORD contains the minimum zoom value. Typically this command is used when the user drops down the Zoom control on the toolbar of the UI active object. The applications and objects that support this command are required to support all the integral zoom values that are within the (min,max) pair they return.

Appendix: Office Binder Issues

This short appendix describes some of the details concerning Office Binder's implementation of programmatic printing. In addition, one other note is worth mention which is that Binder allows the user to open a document object into a separate window (the semantics of *IOleDocumentView::Open*). It is recommended that Office-compatible DocObjects support separate window activation if possible.

As for printing, Binder uses *IPrint* for Binder level printing, thus DocObjects that wish to work well with Binder must implement *IPrint.* The Binder supports two distinct levels of printing. At the Binder level, users can print multiple sections. At the section level, only the selected section will be printed (implemented via the *IOleCommandTarget* interface).

Binder Level Printing

When printing a section of the Binder level, Binder will be responsible for displaying the user interface elements that are related to print progress, canceling of the print job, and so forth. This will be indicated by the absence of the PRINTFLAG_MAYBOTHERUSER flag in the call to *IPrint::Print*. Binder is always going to call *IPrint::Print* with PRINTFLAG_RECOMPOSETODEVICE bit set. Depending on the user's selection, Binder may set the DM_COLLATE and DM_COPIES bits of *dmFields* field of DVTARGETDEVICE. When DM_COPIES bit is set then the *dmCopies* field contains the number of copies that need to be printed. The document object being printed must look at these fields and use the information they contain when it prints.

When the user selects the *Print to file* option in the print dialog box, then the Binder will call *IPrint::Print* with PRINTFLAG_PRINTTOFILE and it will pass the name of the file (into which the document object must print) through the "*portname"* field of DVTARGETDEVICE. The document object can then put that file name in the DOCINFO structure, and pass it to the WIN32 *StartDoc* API as part of the printing process. This will take handle the "print to file" request.

Section Level Page Setup and Printing

A user can opt to perform Page Setup and Printing at the section level. When Page Setup is chosen, Binder will call the *IOleCommandTarget::Exec* method with OLECMDID_PAGESETUP. This indicates that the object should prompt the user for page-specific options using its Page Setup dialog.

Similarly when printing a section at the section level, the Binder will call the *IOleCommandTarget::Exec* method with OLECMDID_PRINT, indicating that printing is to be performed. The document object should prompt the user with its File/Print dialog and use it's own settings to perform the print job.

During section level printing the object should display any user interface elements that are needed by the user (that is, print job status, cancellation buttons, etc).

Calling *IContinueCallback::FContinuePrinting*

During Binder-level printing, it is important for DocObjects to call *IContinueCallback::FContinuePrinting* often, so that Binder can response quickly if the user presses the Cancel button in the Binder's print dialog box. The document object must call at least once for each page that it is printing. If a specific page will take a long time to compose and print, then the document object should call more often to assure a timely response to the user's commands.