 © Microsoft Corporation 1992-1993. All Rights Reserved.

# 10. In-place Activation

This chapter gives a brief overview of in-place activation and how it works, and a specification of the interfaces and API functions that support it. A detailed description of how to implement an application that supports in-place activation can be found in Chapter 6 of the OLE Programmers' Reference manual. That description covers activation sequences, how to avoid unnecessary and distracting repainting, menu and toolbar negotiation, accelerator dispatching, help processing and context-sensitive help.

In OLE 1 an embedded object had to be opened into its own top-level window in order to be edited by the user. In OLE 2, we provide the ability for an embedded object to be edited inside the window of its container. We call this “in-place activation.” It is also sometimes referred to as “in-place editing,” but as all objects may not support editing as their primary action (e.g. some objects like video clips may wish to play in-place), the more general term is preferred. Chapter 2 presented the user interface of in-place activation; here we look at the functions and interfaces which support it.

There are different flavors of object that might be considered for activation in place, with varying characteristics. Consider a control on a form: this is normally active for user interaction whenever it is visible, but might distinguish between form editing mode and "runtime" mode. In addition, such controls do not themselves add other visual controls or menu items, other than drop-downs etc. At the other extreme, a document might be considered an object which exposes its user interface within an MDI workspace; this is common in applications today, but the documents are all of a type managed by a single application. In future, generic task-oriented MDI workspaces might be implemented, and applications might use interfaces like this to live in that world.

The user interface recommendations in chapter 2 of this specification address an intermediate form of object: one which is designed to be a part of some container, negotiate its layout, and install menus and controls onto the containing frame when it is active in-place. Objects need not actually install menus and controls, but the activation model is expressed in terms of one-at-a-time within a container, with editing being modal with respect to the rest of the container.

The *architecture* described here allows for variations in the activation model of objects. Objects may be partially active in that they have a window, do their own repainting, and get mouse input directly, and may or may not have the focus for keyboard input. Such objects lose focus (and notify the container) when the user clicks outside, but are not deactivated and do not remove their window until the container so instructs. Multiple such objects may be active simultaneously. In this situation containers may elect to allow the object with focus to install menus and frame-level tools, or they may reserve those resources for their own use. Forms-based application builders are an example of the latter kind of container: the form designer determines the frame level UI and uses the programming interfaces to manipulate the objects on the form, rather than letting the object show its particular UI.

A description of how to implement containers and objects that behave in this way will be found in a technical note in the OLE 2 SDK materials, which should be considered logically as part of this specification.

## 10.1. Basic Approach

This section describes the basic approach to the implementation of in-place activation.

There is no API call for a container or a client site to initiate in-place activation with an object. Instead, some calls to DoVerb() which might be semantically interpreted as initiating in-place activation. The following describes what the embedded object code should do to implement this.

The activation of an embedded object for in-place activation causes a train of events:

- If the verb implies editing (i.e.: making changes to the object rather than just showing it), and the ActiveSite passed to DoVerb is not the same as the object's embedding site (see the discussion of object identity in chapter 4), then the object should request its embedding

 © Microsoft Corporation 1992-1993. All Rights Reserved.

container to show it, and continue using the EmbeddingSite pointer for its activation in place of the ActiveSite.

- An editing window is created by the embedded object. This window will be a child of the container application's window which contains it, but it may or may not belong to the container's process – this depends on whether the embedded object implementation is a DLL or an EXE implementation. The handle to the parent window of this new window can be retrieved by calling IOleInPlaceSite::GetWindow().
- The menu bar of the container is replaced by a composite menu bar which consists of some drop-down menus from the container along with some drop down menus from the embedded object implementation.
- The in-place activation window will be surrounded by the shading recommended in Chapter 2.
- The embedded object implementation may request space from the container to display one or more toolbars, formula bars, rulers, etc.
- The in-place activation window takes the keyboard focus.

The in-place activation window will directly receive keyboard and mouse input while the object is activated. When the user selects commands from the menu bar, the command and associated menu messages will be sent to the container implementation or the embedded object implementation, depending on which body of code owns the particular drop-down menu that was selected. Input through the rulers, toolbars, etc., will go directly to the embedded object since these objects are windows that belong to the embedded object.

The embedded object remains active until it voluntarily gives up the active state or until the container takes the activation away: e.g., the user clicks the mouse inside the client area of the container application but outside the in-place activation window. In particular, the object is not deactivated if the user clicks on the menu bar, the title bar, or a scroll bar belonging to the container application.

The code implementing an embedded object can be either a DLL or an EXE file. In the DLL case, the in-place activation window is just another child window in the container's window tree, and it receives input through the container application's message pump. In the EXE case, the in-place activation window belongs to the embedded object's process, but has a parent in the container's window tree. The menu bar contains items (normally drop-downs) from both processes. The input for the in-place window will appear in the embedded object's message queue and will be dispatched by the message loop in the embedded object application. The OLE libraries are responsible for making sure that menu commands and messages are dispatched correctly. The applications have some responsibility for dispatching keyboard accelerators; this is described in later sections

### 10.1.1.Window Hierarchy

When an object is active in place, it has access to several windows in the window hierarchy. In the general case, these are

| | |
|---|---|
| *frame window:* | the MDI frame or workspace window, where the main menu resides |
| *document window:* | the MDI child window or sub-window where the document is shown |
| *parent window:* | the container's window which contains the object's view, and which should be the parent window for the object's editing window. |

These windows are illustrated in Figure 94.

In many cases, these will be the same window. For example, an SDI container application will have frame window the same as document window. In these cases, the container returns NULL for the redundant window object pointer in the GetWindowContext call. Complex cases involve MDI applications, split-pane and structured views, and nested objects (where parent will not be the same as pane). Split-pane views are implemented by supplying the appropriate pane window as the document window for the editing session. The object creates its editing window as a child of the "parent" window, and uses the others to

 © Microsoft Corporation 1992-1993. All Rights Reserved.

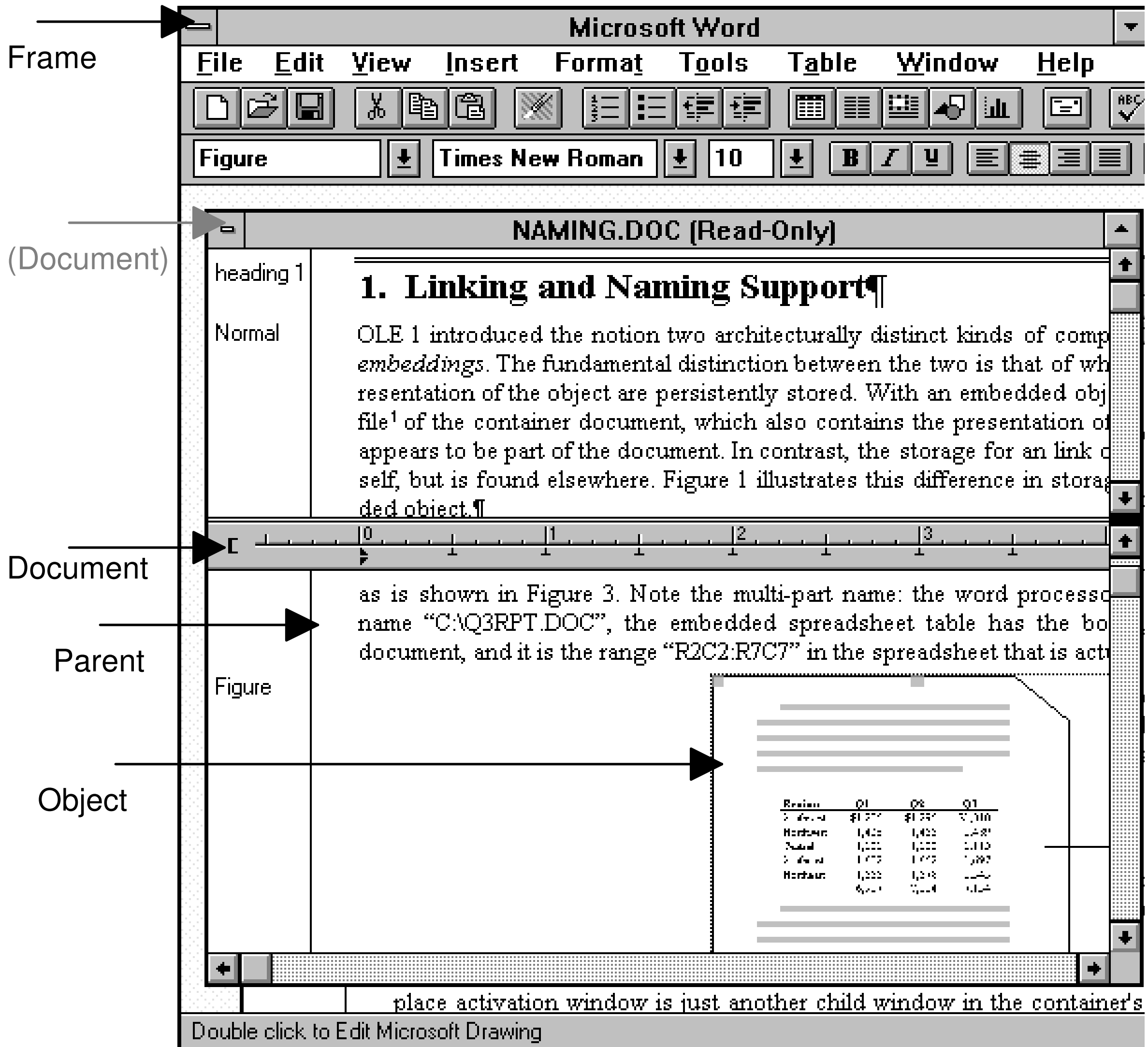



**Figure 94. In-place editing Window Hierarchy**

place the main menu bar and toolbars. An object may request installation of toolbars, rulers etc. on the frame or document windows as it sees fit. The object may also add adornments around the object, by expanding the window to make space. These latter adornments overlay the other contents in the container.

## 10.1.2.States and Transitions

An object which is active in place is modal with respect to the document in which is it contained; other than the container menu commands which remain available on the combined menu bar, the user is not able to edit other objects or data in the same document. Clicking on them will cause UI de-activation of the object, and make a new selection. In this case, the object will remove its menu and any toolbars, the shaded border will be removed, and the new selection state will determine the UI appearance. The object will not actually be shut down at this point, to permit quick reactivation if the user activates the object again, and to enable Undo. The object will be fully de-activated when the user invokes some other command that finally loses the ability to Undo

More complex is the case where the container is an MDI application. If the user switches to a different top level window, then the behavior is the same as the SDI case. If however the user switches to another document window in the same MDI workspace, then the following occurs: the containing document window is drawn as inactive; the menu bar is replaced by the menu bar appropriate for the document

 © Microsoft Corporation 1992-1993. All Rights Reserved.

being activated; any other frame-level or pop-up UI controls such as toolbars shown by the object are hidden. On switching back, the menu is restored, the toolbars are shown, focus is given to the object's window, and the document window is shown as active.

While an object is active, the user may as usual switch to other windows, leaving the object (and its containing document) in a dormant state. There are two cases to consider. First, if the container is an SDI application, the user will have switched to another top level window, and the frame window is simply shown as inactive (border, title colors). The menu bar and any toolbars or other editing controls remain visible, but disabled. On switching back, the titles are shown as active, and focus is given to the object's window.

Frame-level tools belonging to an in-place active object appear and disappear as MDI child windows are activated; this can cause a disturbing movement of windows as the frame is rearranged. For this reason, it is better to use floating palettes (pop-ups) or attach the tools to the document window rather than the MDI frame. Along the same lines, Container applications need to make careful decisions between removing toolbars to save space and leaving them visible (but disabled) to preserve stability.

## 10.2. Merging Client and Server UI Components

### 10.2.1. Installing Menus onto a Combined Menu Bar

When an embedded object is activated for in-place activation, it shares the user interface with its container as described in Chapter 2. The main control which is used by both the container and the embedded object is the menu bar. When an embedded object is activated, the container must provide only those drop-downs belonging to the container that appear on the composite menu bar. Usually, these will be the File and Window drop-downs. The embedded object's implementation will provide the drop-down menus that it supplies to the composite menu bar.

Chapter 2 describes the recommended menu arrangement, where the container supplies the File menu, and if the container is an MDI application, the Window menu. The mechanisms used to implement this are more flexible, allowing the container and object each to supply three groups of menus, which are interleaved. This does not mean that the recommended policy is changed, just that the underlying architecture could accommodate different policies. Applications should go against the recommendation only with extreme caution.

In this model, the container supplies three groups of submenus, called the File group, the Container group, and the Window group. The object supplies three groups, called the Edit group, the Object group, and the Help group. These are combined in the order File, Edit, Container, Object, Window, Help.

*Example.* If the menu groups provided by the container are

| group | *File* | *Container* | *Window* |
|---|---|---|---|
| content | **File** | empty | **Window** |

and the menu groups provided by the embedded object are

| group | *Edit* | *Object* | | | | *Help* |
|---|---|---|---|---|---|---|
| content | **Edit** | **DataSeries** | **Gallery** | **Chart** | **Format** | **Help** |

then the composite menu bar is

| File | Edit | DataSeries | Gallery | Chart | Format | Window | Help |
|---|---|---|---|---|---|---|---|
| container menu | object menus | | | | | container menu | object menu |

 © Microsoft Corporation 1992-1993. All Rights Reserved.

*More complex hypothetical example.* If the menu groups provided by the container are

| group | *File* | *Container* | *Window* |
|---|---|---|---|
| content | **File** | **Project** | **Window** |

and the menu groups provided by the embedded object are

| group | *Edit* | *Object* | | | | *Help* |
|---|---|---|---|---|---|---|
| content | **Edit** | **DataSeries** | **Gallery** | **Chart** | **Format** | **Help** |

then the composite menu bar is

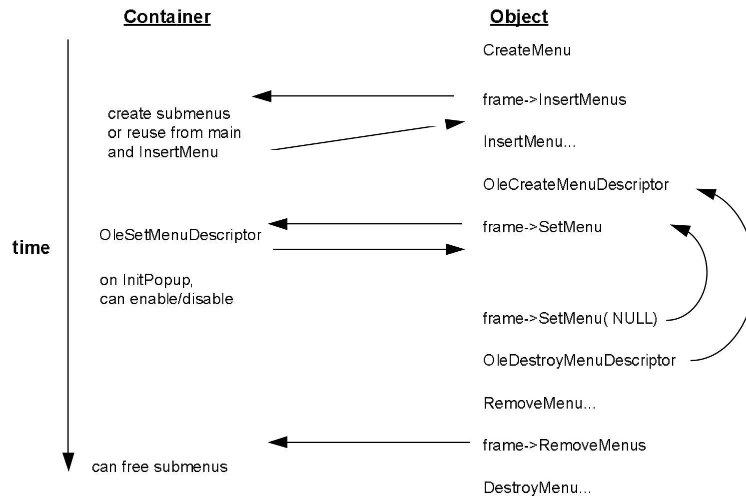
| **File** | **Edit** | **Project** | **DataSeries** | **Gallery** | **Chart** | **Format** | **Window** | **Help** |
|---|---|---|---|---|---|---|---|---|
| container menu | object menu | container menu | object menus | | | | container menu | object menu |

### 10.2.1.1. Resource ownership

The individual drop-down menus will be referenced, *not* copied. This means that the container or the embedded object implementation can use the HMENU of the individual drop-downs that it provided to check, gray, or draw menu items. Similarly, the container can use the same drop-downs for multiple object menus, provided it ensures that the lifetime of the drop-down menus exceeds that of any object menu that refers to them.

The object server creates the top level menu that is the combined menu, and destroys it when finished with it. This means that the server can in fact create several such menus, and switch among them, perhaps depending on editing mode or viewing state. The container does not have this ability: it can only supply one set of drop-down menus for the duration of an active server.

The following sketch shows the flow of control in constructing the combined menu. Notice that create/destroy and set/unset calls are paired. Each nested group of calls can be repeated as necessary, though normally only the SetMenu() calls will be repeated, to set the menu when focus change and task switching occur.




The object creates a menu (using CreateMenu), then calls the container to insert its drop-down menus. The container inserts these menus, so that the object's menu contains references to the container's drop-downs, not copies. The container may create these submenus on demand, or re-use submenus of its main menu. The server returns three counts, which are the number of items in each of the three groups, as discussed above. The object inserts its menus into the resulting menu, using the counts to determine where its groups should go in the combined menu. Then the object calls OleCreateMenuDescriptor() passing in the menu handle, and the six counts (three from the container, and three from the object). OLE constructs a descriptor which it will use to dispatch commands to the appropriate window (container or object). Finally, the object calls IOleInPlaceFrame::SetMenu(), passing in the menu handle, the menu descriptor and the window handle for the object (to which menu messages and commands will be sent). When de-activating, the object calls IOleInPlaceFrame::SetMenu(NULL, NULL, hWndFrame) to remove its menu,

 © Microsoft Corporation 1992-1993. All Rights Reserved.

and OleDestroyMenuDescriptor() to free the descriptor. Then the object removes its drop-down menus and calls the container to remove the container's menus. This indicates to the container that the combined menu no longer refers to the container's drop-downs, which might allow the container to free them. Finally, the object frees the menu with DestroyMenu(). The container must have used RemoveMenu to remove its drop-downs, otherwise the call to DestroyMenu() will free them.

### 10.2.2. Status Line

Many applications use a status line to give brief informative help to the user. Examples are feedback on the effect of a menu selection (before the user activates it), or hints about what the selection is and what can be done with it, such as “Double-click to edit this chart object.” When an object is active, it can ask to display information in the container's status line (if there is one), by calling IOleInPlaceFrame::SetStatus-Text.

### 10.2.3. Installing toolbars and other UI Elements

OLE 2 allows objects, when they are active, to supply auxiliary windows to the container. These windows can include toolbars, status lines, palettes, and formula bars. The container application will generally provide space for the windows; the size of the tool area is the result of negotiation between the container, which knows how much space is available, and the object, which knows the contents of the windows. If the container does not provide space for the windows, the object can either do without them, display them as pop-up windows, or refuse to activate for in-place activation.

In general, containers should take down or disable their own toolbars, formula bars, etc., while an embedded object is active since their commands will not be available.

Objects may want to display controls around their borders when they are active, such as row and column headings for tables, etc. Gluing attachments to objects should *not* affect the layout of the surrounding container objects – if necessary the attachments will overlap the surrounding container contents. The model here is that they exist only during the in-place activation process and are not part of the object.

When it is activated, the embedded object may ask windows further up the containment hierarchy to allocate space along their borders in which the object can show toolbars and other editing controls. For this purpose, the object calls IOleInPlaceUIWindow::GetBorder to ask for the outer rectangle, IOleInPlaceUIWindow::RequestBorderSpace to ask for space defined by the inner rectangle, and IOleInPlaceUIWindow::SetBorderSpace() to actually do it.

The container may do nothing, or it may make space as requested. It is up to the container to determine whether the UI object lies over part of the container's contents or whether the layout of the container's contents is shifted to make room. Commonly, the container draws scroll bars inside any border attachments, next to the document content. If the container refuses, the object may elect to show the tool as a pop-up, or it may not display it at all; the choice is up to the embedded object.

Applications that support repositioning of toolbars (e.g. by dragging) may re-invoke the negotiation at any time. Containers may (though it is not recommended) alter the border within which the object's tools are placed. The container should call IOleInPlaceActiveObject::ResizeBorder, and the object will re-negotiate the placement and size of toolbars.

Floating palettes are simply popup windows owned by the object window. The container need not be aware of their existence. The object is responsible for hiding and showing these windows on activation and deactivation.

### 10.2.4. Modeless Dialogs

When an object is in-place active, it is possible for one application to display a modal dialog at the same time that the other application has modeless dialogs visible. To get the right enable/disable behavior, the application that wants to display the modal dialog must first call EnableModeless(FALSE), and after completion of the dialog it must call EnableModeless(TRUE). The object calls this method in the IOleInPlaceFrame interface, and the container calls it in the IOleInPlaceActiveObject interface.

 © Microsoft Corporation 1992-1993. All Rights Reserved.

### 10.2.5. Context-sensitive Help

Many applications use an accelerator key to allow the user to obtain help on some visible control or item (usually shift-F1). On receipt of this key, the application enters a modal state, and the next mouse click determines the help topic. When an object is in-place active, there are two applications that need to enter this modal state. The model is that the object will receive the accelerator and determine that the help mode will be entered. The object will call IOleInPlaceFrame::ContextSensitiveHelp(TRUE), to notify the container to enter the mode. If the user exits the mode (e.g. with the escape key), the object will call IOleInPlaceFrame::ContextSensitiveHelp(FALSE) to tell the container to exit the modal state. If the user clicks on something owned by the object, the object will call IOleInPlaceFrame::Context-SensitiveHelp(FALSE), and provide the help. If the user clicks on something owned by the container, the container will call IOleInplaceActiveObject::ContextSensitiveHelp(FALSE) to tell the object to exit the mode, and the container will provide the help.

### 10.2.6. Undo

When an object is activated in-place, the immediate container should preserve its undo state, and indicate to the object whether it has undo state (through the IOleInPlaceSite::GetWindowContext call). If the first thing the user does is request Undo, the object will deactivate and ask the container to Undo (through the IOleInPlaceSite::OnUIDeactivate call).

When an in-place active object deactivates, it may have undo state. The object indicates this to the container through the IOleInPlaceSite::OnUIDeactivate call. If the next thing the user does is request Undo, the container activates the object to perform the Undo.

Objects which support the partial activation state where they have no menus or tools when they lose focus may still preserve Undo state until the container informs them that they should discard it by calling IOleObject::DoVerb specifying OLEIVERB_DISCARDUNDOSTATE.

### 10.2.7. Clipping

Objects may be clipped, either by the window boundary, frame adornments, or cropping by some containing object or document. The accumulated clipping is communicated to the object by its immediate container through the IOleInPlaceObject::SetObjectRects call. Intermediate objects (that are both contained and a container) are responsible for computing the intersection of the clipping rectangles.

### 10.2.8. Accelerator keys

When an in-place active object gets a keystroke message that is not an accelerator that it recognizes, it must check to see if it is one that the container recognizes. The object should first check its own accelerators, then call OleTranslateAccelerator, which will return S_FALSE if the container does not want the keystroke, in which case the object should continue with the normal TranslateMessage/DispatchMessage code. If the container has indicated that it does want the keystroke (by putting an entry in the accelerator table), OLE will call IOleInPlaceFrame::TranslateAccelerator, and return the result that the container returns. The container may call (Windows functions) TranslateAccelerator and/or TranslateMDISysAccel, or do its own processing. Note however that Windows will no longer have state associated with the message, like key state or extra message info, since the message has been transferred from the server (object) to the client (container). Containers should therefore choose accelerators that can be encoded into the accelerator tables defined here, for use when an object is in-place active.

## 10.3. Activation and Focus

Activation for in-place activation is always initiated by the object, in response to a DoVerb() command; however, this DoVerb() command may have been called by the container in response to a double click on the object.

 © Microsoft Corporation 1992-1993. All Rights Reserved.

### 10.3.1. Negotiation

At the time the activation takes place, the object has been created; it has an associated client site in the container; the object has returned a non-null pointer in response to QueryInterface() for the in-place interface; and the container and the object have gone through some sort of layout negotiation and agree on the size and the position of the object. We assume that the object has its position stored in its structure. The object must call IOleInPlaceSite::CanInPlaceActivate() to ask the container if it is allowed to activate in place. This is because a container application may elect to refuse to allow in-place activation in some circumstances even though the application does in general support it.

### 10.3.2. Object Activation

The following describes the implementation of IOleObject::DoVerb() in the case where the object decides to start in-place activation. The container may call DoVerb() because the user has double-clicked on an embedded object or because the user has selected a verb from the Edit menu.

DoVerb() takes an LPMSG parameter as well as an lindex parameter (to tell which of several extended-layout rectangles containing the object has been activated) which the object can use to activate an object which it contains (thus implementing inside out activation) or to determine how DoVerb() was called (by clicking, menu selection).

The sequence of operations when an object is to be activated for in-place activation is:

- The object uses results from the most recent layout negotiation in order to determine whether it can edit in place or not. For example, a view scale other than 100% might make it impossible to edit in place; the object could open a new top-level editing window in this case, as in OLE 1. Layout negotiation is performed in the coordinate space of the target device, normally a printed page. On activation, the object assumes that the negotiated space is mapped onto the window coordinates supplied in the lprcChildPos parameter of the GetWindowContext call. This enables the object to determine the prevailing view scaling and perform its drawing properly.
- The object gets the pointers to interfaces attached to the window handles of the frame window and document window, as well as the window containing the embedded object. If the object wishes to use menu commands, it creates a combined menu as described above.
- The object should:
    1. Calculate the required size of window, which is the size passed in lprcChildPos, possibly adjusted for adornments around the edge of the object (such as row and column headings), including the border shading and sizing handles, and clipped to the rectangle passed in SetObjectRects. Then the object should create a window of the right size as a child of the parent (obtained from IOleInPlaceSite::GetWindow). The window is created by the object implementation so that the implementation can use window classes it has defined, and will get input messages directly from the system queue.
    2. If the object needs palettes, toolbars, or adornments of any type, it should negotiate with the container to have them positioned, as described above.
    3. If the container refuses to accept any adornments, the object may bail out of in-place activation at this point, clean up, and initiate opening (OLE 1 style editing).
    4. It should display these adornments and display the new menu bar. It is important that the visual changes not take place until after there is any possibility that the object might decide that it is unable to edit in place.
    5. Call SetActiveObject passing its pointer to each of the IOleInPlaceUIWindow interfaces that it was given in the GetWindowContext call. This enables the various windows to be aware of which is the active object.

In-place activation can be deactivated on request from the container (e.g., when the user clicks in the client area of the container and outside the embedded object's window – this is when the container code

 © Microsoft Corporation 1992-1993. All Rights Reserved.

for selection is called) or by the embedded object (e.g., when the user moves the cursor past an end of the object using the keyboard).

When an embedded object is deactivated for in-place embedding it should:

- Remove the shading around the object in the container.
- Remove the combined menu bar.
- Ask the container to reinstall its own UI.
- Destroy or hide its toolbar, palette, and any other adornments that it may have created.
- Call the container's IOleInPlaceSite::OnUIDeactivate() to allow the container to execute any code it may want to run when an object is deactivated.

The object should not at this point discard its undo state, nor need it save changes to storage. The container at some later time will call InPlaceDeactivate, at which time the Undo state may be discarded, and will separately ask the object to save its state to disk. See IPersistStorage::Save for a description of saving. The object may optionally leave its window visible at this point. The container must call IOleObject::DoVerb specifying OLEIVERB_HIDE when the window should be removed. Normally, this will be during OnUIDeactivate, or following UIDeactivate.

### 10.3.3. Interaction with Window Activation and Focus

Since the user can switch between top-level windows, and MDI child windows, as well as between objects and the container, the activation management is more complicated. When the user switches to another frame window, the frame menu and frame and document level toolbars should stay in place, inactive. On switching back, the title bar is shown active again. When the user switches between MDI child windows, the frame-level UI components should disappear (to be replaced by those of the new MDI child) and reappear on switching back. The document level tools should stay in place. Floating palettes should hide or be shown as inactive.

A related complication is if there are nested objects. When an object nested in another deactivates, its container should activate and install its UI. Similar considerations apply to the shading around the active object. The general rule is that window titles, menu bar, border shading, and frame-level tools belong to the active object and are switched as a group.

To make all this work reliably, several pieces of code are required in both the container and the object applications, and these need to be connected in to the existing code for managing window activation. The best description of this is to be found in chapter 6 of the OLE Programmers' Reference manual.

## 10.4. Impact of In-place Activation on the Container Application

The various negotiations (layout, in situ editing adornments, etc.) impose some requirements on the container application that are discussed in the sections on those negotiations. This section discusses some of the global considerations that arise because a container might embed or link to OLE 2 objects.

Objects might be implemented as EXEs or as DLLs; a container must be prepared to embed either kind.

If the object implementation is a DLL, then it does not have its own message queue or message pump and is dependent on those of the container. This means that, at least while an embedded object is being edited in-place, the container should dispatch all messages sent to the in-place activation window. If the container application processes messages based on the message type as they come out of the message queue, it should suspend this special processing while an object is being edited in place. This means that the application should keep a flag to indicate whether some embedded object is activated for in-place activation; the IOleInPlaceSite::OnUIActivate() and OnUIDeactivate() functions are useful for this.

When an embedded object is activated for in-place activation, the accelerator keys for the object take precedence over the accelerator keys for the container. If the container receives any keyboard messages while an embedded object is activated for in-place activation and has the focus (this will happen only if

 © Microsoft Corporation 1992-1993. All Rights Reserved.

the implementation for the object is a DLL) then the container should call IOleInPlaceObject::Translate-Message() and call its own message translation routines only if the returned value is False.

If the object implementation is an EXE, then it will be its own process with its own message queue. In this case the object code is not dependent on the container for message dispatch. There are consequences, however, in having two processes instead of one. There will be two message queues, one for the container and one for the embedded object, and two processes where before there was only one.

Normally a windows application can receive a SendMessage() when it calls GetMessage(), WaitMessage(), Yield(), PeekMessage(), or calls SendMessage() to send a message to a window in another process. If a server is implemented as a separate application (not as a DLL), then the OLE architecture implies that the window created by that application will have a parent in another process. Many windows functions cause SendMessage()s to be sent from parent windows to child windows, and these now will involve SendMessage()s across processes. The result is that the container application will be open to receiving SendMessage()s much more frequently than before; this might lead to re-entrancy problems. The container application will have to defend against this. There is a Windows function returning a Boolean, InSendMessage(), that a window procedure can use in these cases to determine if it is being called via SendMessage() from another process.

The container application also needs to be aware of UI objects that it is displaying on behalf of its embedded objects:

1. The container should not move the embedded object when it is displaying or hiding toolbars, palettes, etc. for an object active in-place since most likely the user has double clicked on the object, and moving it could mean that it is no longer under the mouse cursor.
2. If the container scrolls, causing the object to move, the container must ensure that the object's window gets the correct move and or size messages
3. If the container's window(s) change size, then the container must re-negotiate with the object over the space assigned for the object's tools, in case the object wants to wrap instead of clipping. Floating palettes will not be affected by this, only windows attached as children of the frame and document windows.

## 10.5. Impact of In-Place Activation on the Embedded Object Application

The embedded object implementation must of course implement all those interfaces which are appropriate, such as for in-place activation, layout, etc. There are obligations in addition to those related to implementing any specific interface that must be met.

If the implementation for the embedded object is a separate application (not a DLL) then its message pump should call OleTranslateAccelerator(lpFrame, lpInPlaceFrameInfo, lpmsg) whenever the object is being in-place edited and the object's accelerator translation, if any, has not translated the message, and before calling TranslateMessage() and DispatchMessage(). This is because the rule for translating accelerator keys is that the embedded object's implementation has the first chance of performing translation, and if it does not, then it gives the container a chance to translate. There is a similar requirement on the container when the embedded object is implemented with a DLL; the portion of code that has the message pump also has an obligation to allow the other piece of code to translate messages (in this case through IOleInPlaceActiveObject::TranslateAccelerator).

In general, the implementation of an embedded object has to separate the code that manipulates the top level frame window, the MDI management, and menu bar management from the code that manipulates the contents of the client area of its window or windows. When the code is being used to in-place edit an embedded object, the frame window and the menu bar are not directly available to it.

A reasonable strategy, especially for new applications that support OLE and in-place activation, is to structure the application as two pieces: The first is a relatively small EXE file that contains the code that manages the frame window, the File menu, the Window menu if it is an MDI application, and that contains the message pump. The second piece is a DLL that supports the OLE 2 interfaces. Only the DLL

 © Microsoft Corporation 1992-1993. All Rights Reserved.

is necessary to support embedded objects, and when the application is run as a stand-alone application, the EXE provides the functionality necessary to embed a single instance of the embedded object type in the client area of its top frame window or in its top level MDI windows. This strategy is more feasible in Windows 32 since then most of the annoyances of writing DLLs disappear since per-instance data segments are supported and SS is always == DS.

As noted in the preceding section, the container may move and resize the object's window. The object must take notice of these messages to maintain any necessary relationships, such as aligning a ruler on the document border with the object.

## 10.6. In-place Activation Functions and Interfaces

The following functions and interfaces are used to implement in-place activation support.

### 10.6.1. Accelerator Support

```
typedef struct tagOIFI {
        UINT cb;
        BOOL fMDIApp;
        HWND hwndFrame;
        HACCEL  haccel;
        int    cAccelEntries
        } OLEINPLACEFRAMEINFO;
```

OLEINPLACEFRAMEINFO is a structure that carries data needed by OLE to dispatch keystroke accelerators to a container (frame) while an object is in-place active. On Windows, this is a table of key codes and modifiers. cb is the count of bytes in the structure. fMDIApp is TRUE if the container is an MDI application (so that OLE will dispatch MDI accelerators), hwndFrame is the handle to the container's frame window. haccel is a handle to container-supplied accelerator table. cAccelEntries is the number of entries in the accelerator table.

When an object activates in place, it will call GetWindowContext from its container. The container should allocate a table, and return it through the appropriate argument.

#### 10.6.1.1. OleTranslateAccelerator

HRESULT OleTranslateAccelerator(lpFrame, lpFrameInfo, lpmsg)

This function matches the message against the accelerator table in lpFrameInfo. If found, the message is passed to the container through the IOleInPlaceFrame::TranslateAccelerator (along with the wID), and this function returns whatever the container returns. If no match is found, this function returns S_FALSE without calling the container.

| Argument | Type | Description |
|---|---|---|
| lpFrame | IOleInPlaceFrame* | a pointer to the frame interface where the keystroke might be sent |
| lpFrameInfo | LPOLEINPLACEFRAMEINFO | a pointer to the OLEINPLACEFRAMEINFO struct, which contains among other things the accelerator table obtained from the container. |
| lpmsg | LPMSG | pointer to a MSG struct containing the keystroke |
| return value | HRESULT | S_OK if the keystroke was processed, S_FALSE otherwise, in which case the object should continue processing this message. Other values indicate RPC failures. |

### 10.6.2. Combined Menu Support

```
typedef struct tagOleMenuGroupWidths {
        LONG      width[6];
} OLEMENUGROUPWIDTHS;
```

 © Microsoft Corporation 1992-1993. All Rights Reserved.

OLEMENUGROUPWIDTHS encodes the number of menus in each group, as described at the beginning of this chapter. The container supplies values for elements 0, 2, and 4, and the object supplies 1, 3 and 5.

```
typedef HANDLE HOLEMENU;
HOLEMENU   OleCreateMenuDescriptor (hmenuCombined, lpMenuWidths);
HRESULT    OleSetMenuDescriptor (holemenu, hwndFrame, hwndActiveObject, lpFrame, lpActiveObject);
void       OleDestroyMenuDescriptor (holemenu);
```

HOLEMENU is a handle to a data structure that OLE uses to dispatch menu messages and commands to container and object as appropriate. The object creates the combined menu using IOleInPlaceFrame::InsertMenus, and creates the shared menu descriptor with OleCreateMenuDescriptor. When UI activating, the object calls IOleInPlaceFrame::SetMenu, which in turn should call OleSetMenuDescriptor, so that OLE can install the dispatching code. When deactivating, the object calls IOleInPlaceFrame::SetMenu(NULL) to remove the shared menu, and the container should call OleSetMenuDescriptor(NULL) to unhook the dispatching code. Finally, the object uses OleDestroyMenuDescriptor to free the data structure.

### 10.6.2.1. OleCreateMenuDescriptor

HOLEMENU OleCreateMenuDescriptor(hmenuCombined, lpMenuWidths)

This function can be called by the object to create a descriptor for combined menu, which will be used by OLE to dispatch menu messages and commands.

| Argument | Type | Description |
|---|---|---|
| hmenuCombined | HMENU | the handle to the combined menu created by the object |
| lpMenuWidths | LPOLEMENUGROUPWIDTHS | a pointer to an array of six LONG values giving the number of menus in each group. |
| return value | HOLEMENU | the handle to the descriptor, or NULL if insufficient memory is available. |

### 10.6.2.2. OleSetMenuDescriptor

HRESULT OleSetMenuDescriptor (holemenu, hwndFrame, hwndActiveObject, lpFrame, lpActiveObject);

This function should be called by the container to install the dispatching code on hwndFrame when the object calls IOleInPlaceFrame::SetMenu, or to remove the dispatching code when the object calls the same method with NULL for the HOLEMENU (by passing NULL as the value for holemenu to this call).

| Argument | Type | Description |
|---|---|---|
| holemenu | HOLEMENU | The handle to the shared menu descriptor that was returned by OleCreateMenuDescriptor. If null, the dispatching code is unhooked. |
| hwndFrame | HWND | the handle to the frame window where the menu is installed |
| hwndActiveObject | HWND | the handle to the object's editing window; OLE will dispatch menu messages and commands to this window |
| lpFrame | IOleInPlaceFrame FAR * | a pointer to the frame interface |
| lpActiveObject | IOleInPlaceActiveObject FAR * | a pointer to the active object interface |
| return value | HRESULT | S_OK if the menu was installed correctly; E_FAIL if a Windows function call failed, which indicates improper arguments, or resource allocation failure. |

 © Microsoft Corporation 1992-1993. All Rights Reserved.

#### 10.6.2.3. OleDestroyMenuDescriptor

void OleDestroyMenuDescriptor(holemenu)

This function should be called by the container to free the data structure allocated by OleCreateMenu-Descriptor.

| Argument | Type | Description |
|---|---|---|
| holemenu | HOLEMENU | The handle to the shared menu descriptor that was returned by OleCreateMenuDescriptor. |
| return value | void | This function cannot indicate failure. |

### 10.6.3. Object, Container, and Frame Interfaces.

An in-place active object has interfaces to talk to its immediate container (client site interfaces), and objects higher up the containment hierarchy but attached to significant points in the window hierarchy (document and frame interfaces). A container has interfaces to contained objects (IOleInPlaceObject), and the upper level objects (at document etc. levels) have interfaces to the currently active object. Nesting of objects makes it possible for several levels of object to be believed active by their immediate containers, but one in particular is the current active object which has the editing menus etc.

The arrangement of objects and connections between their interfaces is illustrated in the Figure 95. Calls related to installing menus and UI tools, and window activation and switching, and menu and keystroke dispatching go through the direct path between the active object and the top-level container. Calls related to object activation/de-activation, scrolling, clipping etc. go through the containment hierarchy so that each level can perform the correct actions.

Where appropriate, interfaces include IOleInPlaceUIWindow, or just IOleWindow.

```
interface IOleWindow : IUnknown {
    virtual HRESULT GetWindow (HWND FAR * phwnd) = 0;
    virtual HRESULT ContextSensitivieHelp (fEnterMode) = 0;
};
```

#### 10.6.3.1. IOleWindow::GetWindow

HRESULT IOleWindow::GetWindow(phwnd)

This function returns the window handle associated with the object. The object may be the in-place active object, the in-place active site (its parent window) or one of document or frame object.

| Argument | Type | Description |
|---|---|---|
| phwnd | HWND FAR * | a pointer to an HWND where the window handle will be returned. |
| return value | HRESULT | S_OK if the window handle was successfully return through phwnd;<br>E_FAIL if there is no window handle (currently) attached to this object. This can only occur if the object is misbehaving, or if the caller has held on to a pointer to the interface beyond the proper time. |

#### 10.6.3.2. IOleWindow::ContextSensitiveHelp

HRESULT IOleWindow::ContextSensitiveHelp(fEnterMode)

This method is called by the active applications to manage entering and exiting help mode. When an application wishes to enter help mode, it calls ContextSensitiveHelp(TRUE) to tell the other to enter the mode also. If the user clicks on an item in the container, the container will call ContextSensitive-Help(FALSE) to take the object out of the mode, and then gives help. If the user clicks in the object, the object should call IOleWindow::ContextSensitiveHelp(FALSE) to tell the container to exit the mode, and then give help. The object may also initiate the help mode with IOleInPlaceFrame::ContextSensitiveHelp.

Handling of F1-invoked help on menu items is achieved wither by installing a message hook to filter the F1 key, or by asking OLE to install such a hook by calling OleSetMenuDescriptor with non-NULL values for lpFrame and lpActiveObj.

 © Microsoft Corporation 1992-1993. All Rights Reserved.

| Argument | Type | Description |
|---|---|---|
| fEnterMode | BOOL | TRUE if help mode should be entered, FALSE if it should be exited. |
| return value | HRESULT | S_OK if the call succeeded. Other return values indicate RPC failures. |

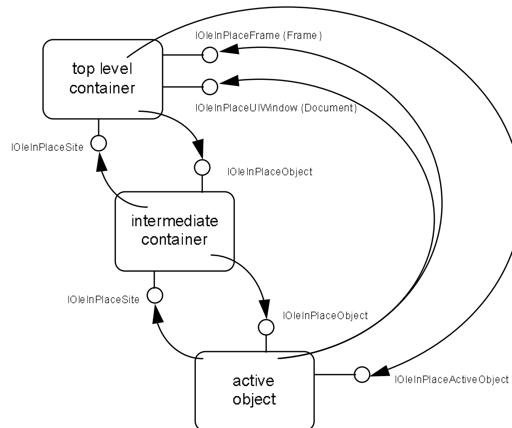



**Figure 95. Relationship of Object, Container, and Frame Interfaces**

## 10.6.4. IOleInPlaceObject Interface

```
interface IOleInPlaceObject : IOleWindow {
      virtual HRESULT InPlaceDeactivate () = 0;
      virtual HRESULT UIDeActivate () = 0;
      virtual HRESULT SetObjectRects (LPRECT lprcPoRect, LPRECT lprcClipRect) = 0;
      virtual HRESULT ReactivateAndUndo() = 0;
      };
// This is obtained through QueryInterface on IOleDataObject or IOleObject or whatever
```

### 10.6.4.1. IOleInPlaceObject::InPlaceDeactivate

HRESULT IOleInPlaceObject::InPlaceDeactivate()

This function can be called by the object's immediate container to deactivate the object and free associated resources. In particular, the object will discard its Undo state on this call. A server should not shut down immediately after this call, but wait for an explicit call to close it.

| Argument | Type | Description |
|---|---|---|
| return value | HRESULT | S_OK if successful. Other values indicate RPC failures. |

### 10.6.4.2. IOleInPlaceObject::UIDeactivate

HRESULT IOleInPlaceObject::UIDeactivate()

This function can be called by the object's immediate container (e.g. if the user has clicked in the client area outside the object). The object should remove its UI, and call the client site's OnUIDeactivate etc. at the appropriate points. The object may clean up resources such as menus and windows, or may keep them around hidden until the full InPlaceDeactivate() is called. Object implementors should note that they must call IOleInPlaceSite::OnUIDeactivate before touching the menus, so that the container will detach it from the frame window first. After a call to UIDeactivate(), the object may be able to respond faster to a later DoVerb() call. If the container has called UIDeactivate(), it should later call InPlaceDeactivate() to clean up resources properly. The container may assume that stopping the object (see OleStop()) or releasing it will clean up resources if necessary, so the object must be prepared to do so if InPlaceDeactivate() has not been called at these points.

 © Microsoft Corporation 1992-1993. All Rights Reserved.

The object need not remove its window until the container calls IOleObject::DoVerb specifying OLEIVERB_HIDE. This implies that containers that assume only a single active object at a time *must* call IOleObject::DoVerb specifying OLEIVERB_HIDE after IOleInPlaceObject::UIDeactivate.

| Argument | Type | Description |
|---|---|---|
| return value | HRESULT | S_OK if successful. Other values indicate RPC failures or failure to free resources. |

#### 10.6.4.3. IOleInPlaceObject::SetObjectRects

HRESULT IOleInPlaceObject::SetObjectRects(lprcChildPos, lprcClipRect)

The immediate container calls this method on the object when the object is activating to communicate the amount of the object that is actually visible. Clipping may have been imposed by container objects, border controls, or window boundary. Additionally, the container calls this method when either rectangle changes, because of border negotiation or scrolling or sizing the window.

| Argument | Type | Description |
|---|---|---|
| lprcChildPos | LPRECT | a pointer to a RECT structure containing the position and size of the object relative to the parent window (i.e. the same coordinate space as the lprcChildPos argument to GetWindowContext) |
| lprcClipRect | LPRECT | a pointer to a RECT structure containing a clipping rectangle that the container imposes (e.g. to allow for frame adornments) relative to the parent window (i.e. the same coordinate space as the lprcClipRect argument to GetWindowContext) |
| return value | HRESULT | S_OK if successful. Other values indicate RPC failures. |

#### 10.6.4.4. IOleInPlaceObject::ReactivateAndUndo

HRESULT IOleInPlaceObject::ReactivateAndUndo()

The object's immediate container calls this method instead of InPlaceDeactivate, if after deactivating the object (by clicking outside) the user invokes Undo before the Undo state of the object is lost. The object should perform the UI activation sequence, and carry out the Undo operation, and remain active.

| Argument | Type | Description |
|---|---|---|
| return value | HRESULT | S_OK if re-activation succeeded, E_NOTUNDOABLE if called when Undo is not available. |

### 10.6.5. IOleInPlaceActiveObject Interface

```
interface IOleInPlaceActiveObject : public IOleWindow { public:
    virtual HRESULT TranslateAccelerator (LPMSG lpMsg) = 0;
    virtual HRESULT OnFrameWindowActivate (BOOL fActivate) = 0; // (un)grey the shading border
    virtual HRESULT OnDocWindowActivate (BOOL fActivate) = 0; // show(hide) the frame level tools,
                        // and (un)grey the shading border
    virtual HRESULT ResizeBorder (LPRECT pBorderRect, IOleInPlaceUIWindow FAR* lpUiWindow) = 0;
    virtual HRESULT EnableModeless (BOOL fEnable) = 0 ;
    };
```

#### 10.6.5.1. IOleInPlaceActiveObject::TranslateAccelerator

HRESULT IOleInPlaceActiveObject::TranslateAccelerator(lpMsg)

This function is called by the container's message pump when an embedded object is active in-place. This function should be called before any other translation, and the container should apply its own translation only if this function returns S_FALSE. This function will only be invoked for an embedded .DLL object since an .EXE object will get keystrokes in its own message pump, and so the container will not get those messages. If perchance the container does call this method for an object that is not implemented in a DLL,

 © Microsoft Corporation 1992-1993. All Rights Reserved.

the default handler will simply return S_FALSE. There will be no overhead from communicating with a server application.

| Argument | Type | Description |
|---|---|---|
| lpMsg | LPMSG | a pointer to the message retrieved from the message queue which might need to be translated. |
| return value | HRESULT | S_OK if message was translated, S_FALSE if it was not translated. Other values indicate RPC failures. |

### 10.6.5.2. IOleInPlaceActiveObject::OnFrameWindowActivate

HRESULT IOleInPlaceActiveObject::OnFrameWindowActivate(fActivate)

This method is called when the top level frame window is activated or deactivated, and the object is the current active object for the frame. If activating, the object should shade take focus.

| Argument | Type | Description |
|---|---|---|
| fActivate | BOOL | TRUE if the window is activating, FALSE if it is deactivating. |
| return value | HRESULT | S_OK if successful. Other values indicate RPC failures. |

### 10.6.5.3. IOleInPlaceActiveObject::OnDocWindowActivate

HRESULT IOleInPlaceActiveObject::OnDocWindowActivate(fActivate)

This method is called when the document (MDI child) window is activated or deactivated, and the object is the current active object for the frame. If activating, the object should install frame-level tools (menu, toolbars, pop-ups etc.), and take focus. If deactivating, the object should remove the frame level tools but not call IOleUIWindow::SetBorderSpace( NULL )

| Argument | Type | Description |
|---|---|---|
| fActivate | BOOL | TRUE if the window is activating, FALSE if it is deactivating. |
| return value | HRESULT | S_OK if successful. Other values indicate RPC failures. |

### 10.6.5.4. IOleInPlaceActiveObject::ResizeBorder

HRESULT IOleInPlaceActiveObject::ResizeBorder(pBorderRect, pUIWindow)

This method is called by the top-level container (document or frame window object) when the border space allocated to the object should change. The object should re-negotiate its border space using RequestBorderSpace and SetBorderSpace.

| Argument | Type | Description |
|---|---|---|
| pBorderRect | LPRECT | a pointer to a RECT structure containing the new outer rectangle within which the object might allocate space. |
| pUIWindow | IOleInPlaceUIWindow FAR * | a pointer to the window object (document or frame) whose border has changed. |
| return value | HRESULT | S_OK if successful. Other values indicate RPC failures. |

### 10.6.5.5. IOleInPlaceActiveObject::EnableModeless

HRESULT IOleInPlaceActiveObject::EnableModeless(fEnable)

This method is called by the top level container to manage enabling and disabling modeless dialogs that the object might be displaying. When the container wishes to display a modal dialog, it first calls EnableModeless(FALSE) to tell the object to disable its modeless dialog windows, and after completion the container calls EnableModeless(TRUE) to re-enable them if necessary. See also IOleInPlaceFrame::-EnableModeless.

 © Microsoft Corporation 1992-1993. All Rights Reserved.

| Argument | Type | Description |
|---|---|---|
| fEnable | BOOL | TRUE if the modeless dialogs should be enabled, FALSE if they should be disabled. |
| return value | HRESULT | S_OK if successful. Other values indicate RPC failures. |

## 10.6.6. IOleInPlaceUIWindow Interface

```
typedef    RECT       BORDERWIDTHS;
typedef LPRECT     LPBORDERWIDTHS
```

BORDERWIDTHS is a structure isomorphic to a RECT structure, except that the members of the structure are interpreted differently. The members specify a width in pixels of a border that should be assigned at the top, left, right, and bottom (respectively) edges of the containing RECT.

```
interface IOleInPlaceUIWindow : public IOleWindow { public:
    virtual HRESULT GetBorder (LPRECT pBorderRect) = 0;
    virtual HRESULT RequestBorderSpace (LPBORDERWIDTHS lpborderwidths) = 0;
    virtual HRESULT SetBorderSpace (LPBORDERWIDTHS lpborderwidths) = 0;
    virtual HRESULT SetActiveObject (IOleInPlaceActiveObject FAR* pActiveObject, LPCSTR lpszObjName) = 0;
    };
```

### 10.6.6.1. IOleInPlaceUIWindow::GetBorder

HRESULT IOleInPlaceUIWindow::GetBorder(pBorderRect)

This function on a document or frame window object returns the rectangle (relative to the window) where the object might put toolbars or similar controls. If the object wants to install such tools it should allocate them within this rectangle, and calculate the resulting rectangle that would be available after such allocation. This latter rectangle should be used in calls to RequestBorderSpace and SetBorderSpace (see below).

| Argument | Type | Description |
|---|---|---|
| pBorderRect | LPRECT | a pointer to a RECT structure where the window-relative outer rectangle will be returned. |
| return value | HRESULT | S_OK if the rectangle was successfully returned through pBorderRect; E_NOTOOLSPACE if the object cannot install toolbars on this window object. |

### 10.6.6.2. IOleInPlaceUIWindow::RequestBorderSpace

HRESULT IOleInPlaceUIWindow::RequestBorderSpace(lpborderwidths)

The object calls RequestBorderSpace to ask if it could install tools around the window frame. These tools would be allocated between the rectangle returned by GetBorder and the rectangle specified by the argument to this call. The space is not actually allocated to the object until it calls SetBorderSpace. This allows the object to negotiate for space (e.g. while dragging toolbars around) but defer moving tools until the action is completed.

| Argument | Type | Description |
|---|---|---|
| lpborderwidths | LPBORDERWIDTHS | a pointer to a BORDERWIDTHS structure containing the requested width for each edge. |
| return value | HRESULT | S_OK if the requested space could be allocated to the object. E_NOTOOLSPACE if the object cannot install toolbars on this window object or there is insufficient space. |

 © Microsoft Corporation 1992-1993. All Rights Reserved.

#### 10.6.6.3. IOleInPlaceUIWindow::SetBorderSpace

HRESULT IOleInPlaceUIWindow::SetBorderSpace(lpborderwidths)

The object calls this function to allocate the space on the border. The rectangle used in this call must have been passed to a previous call to RequestBorderSpace which must have returned S_OK.

| Argument | Type | Description |
|---|---|---|
| lpborderwidths | LPBORDERWIDTHS | a pointer to a BORDERWIDTHS structure containing the requested width for each edge. |
| return value | HRESULT | S_OK if the requested space has been allocated to the object.<br>OLE_E_INVALIDRECT if the rectangle does not lie within that returned by IOleInPlaceUIWindow::GetBorder. |

#### 10.6.6.4. IOleInPlaceUIWindow::SetActiveObject

HRESULT IOleInPlaceUIWindow::SetActiveObject(lpActiveObject, lpszObjName

The object calls this function to inform .the Frame and Document interfaces that the caller is now the active object, which should be notified of window activation etc. In cases of multiple active objects, this call implies that the object has acquired the keyboard focus.

| Argument | Type | Description |
|---|---|---|
| lpActiveObject | IOleInPlaceActiveObject FAR * | a pointer to the object's IOleInPlaceActiveObject interface. |
| lpszObjName | LPCSTR | a pointer to a string containing the name that is to be used as part of the window title while the object is active. |
| return value | HRESULT | S_OK if successful. Other values indicate RPC failures. |

### 10.6.7. IOleInPlaceFrame Interface

```
interface IOleInPlaceFrame : public IOleInPlaceUIWindow { public:
    virtual HRESULT InsertMenus (HMENU hmenu, LPOLEMENUGROUPWIDTHS lpMenuWidths) = 0;
    virtual HRESULT SetMenu (HMENU hmenu. HOLEMENU holemenu, HWND hwndObject) = 0;
    virtual HRESULT RemoveMenus (HMENU hmenu) = 0;
    virtual HRESULT SetStatusText (LPCSTR lpszStatusText) = 0;
    virtual HRESULT EnableModeless (BOOL fEnable) = 0;
    virtual HRESULT ContextSensitiveHelp (BOOL fEnterMode) = 0;
    virtual HRESULT TranslateAccelerator (LPMSG lpmsg, WORD wID) = 0;
    };
```

#### 10.6.7.1. IOleInPlaceFrame::InsertMenus

HRESULT IOleInPlaceFrame::InsertMenus(hmenu, lpMenuWidths)

The object calls this function to ask the frame to add its menus to hmenu, which is empty, and to set the group counts into elements 0, 2 and 4 of the array pointed to by lpMenuWidths. The object will then add its own menus and counts. Objects may call this function several times to build several composite menus if they so desire. The container should reuse the same menu handles for the drop-downs, and indeed may use the same handles as on the normal menu bar.

| Argument | Type | Description |
|---|---|---|
| hmenu | HMENU | a handle to an empty menu. |
| lpMenuWidths | LPOLEMENUGROUPWIDTHS | a pointer to an array of 6 LONG values, where the container will store the number of menus in groups 0, 2 and 4. |
| return value | HRESULT | S_OK if successful. Other values indicate RPC failures or Windows API failures. |

 © Microsoft Corporation 1992-1993. All Rights Reserved.

#### 10.6.7.2. IOleInPlaceFrame::SetMenu

HRESULT IOleInPlaceFrame::SetMenu(hmenu, holemenu, hwndObject)

The object calls this function to ask the frame to install the composite menu. The container should call the Windows SetMenu function, or send the MDI_SETMENU message as appropriate, using hmenu as the menu to install. Then the container should call OleSetMenuDescriptor to install the OLE dispatching code.

| Argument | Type | Description |
|---|---|---|
| hmenu | HMENU | a handle to the composite menu constructed by IOleInPlaceFrame::InsertMenus and Windows InsertMenu calls. |
| holemenu | HOLEMENU | a handle to the menu descriptor returned by OleCreateMenuDescriptor |
| hwndObject | HWND | a handle to a window owned by the object and to which menu messages and commands and accelerators will be sent. |
| return value | HRESULT | S_OK if successful. Other values indicate RPC failures or Windows API failures. |

#### 10.6.7.3. IOleInPlaceFrame::RemoveMenus

HRESULT IOleInPlaceFrame::RemoveMenus(hmenu)

The object calls this function to ask the frame to remove its menus from the composite.

| Argument | Type | Description |
|---|---|---|
| hmenu | HMENU | a handle to the composite menu constructed by IOleInPlaceFrame::InsertMenus and Windows InsertMenu calls. |
| return value | HRESULT | S_OK if successful. Other values indicate RPC failures or Windows API failures. |

#### 10.6.7.4. IOleInPlaceFrame::SetStatusText

HRESULT IOleInPlaceFrame::SetStatusText(lpszStatusText)

The object calls this function to ask the frame to display text in its status line, if it has one.

| Argument | Type | Description |
|---|---|---|
| lpszStatusText | LPCSTR | a pointer to a null terminated character string containing the message to display. |
| return value | HRESULT | S_OK if the text was displayed, S_TRUNCATED if some text was displayed but the message was too long; E_FAIL if status text cannot be displayed. |

#### 10.6.7.5. IOleInPlaceFrame::EnableModeless

HRESULT IOleInPlaceFrame::EnableModeless(fEnable)

The object calls this function to ask the frame to enable or disable its modeless dialogs. See IOleInPlaceActiveObject::EnableModeless for a description.

| Argument | Type | Description |
|---|---|---|
| fEnable | BOOL | TRUE if modeless dialogs should be enabled, FALSE if they should be disabled. |
| return value | HRESULT | S_OK if successful. Other values indicate RPC failures. |

 © Microsoft Corporation 1992-1993. All Rights Reserved.

#### 10.6.7.6. IOleInPlaceFrame::TranslateAccelerator

HRESULT IOleInPlaceFrame::TranslateAccelerator(lpmsg, wID)

This method is called (indirectly by OleTranslateAccelerator) when a keystroke accelerator intended for the container (frame) is received. The container application should perform its normal accelerator processing, or use wID directly, and then return indicating whether or not the keystroke accelerator was processed.

| Argument | Type | Description |
|---|---|---|
| lpmsg | LPMSG | A pointer to a MSG struct containing the keystroke message. |
| wID | WORD | the ID value corresponding to the keystroke in the container-provided accelerator table. Containers may use this value in preference to translating again if they wish. |
| return value | HRESULT | S_OK if the keystroke was consumed, S_FALSE otherwise. Other values indicate RPC failures. |

### 10.6.8. IOleInPlaceSite Interface

```
interface IOleInPlaceSite : public IOleWindow { public:
        virtual HRESULT CanInPlaceActivate () = 0;
        virtual HRESULT OnInPlaceActivate () = 0;
        virtual HRESULT OnUIActivate () = 0;
        virtual HRESULT GetWindowContext (IOleInPlaceFrame ** ppFrame, IOleInPlaceUIWindow ** ppDoc,
                                LPRECT lprcChildPos, LPRECT lprcClipRect,
                                LPOLEINPLACEFRAMEINFO lpFrameInfo) = 0;
        virtual HRESULT Scroll (SIZE scrollExtent) = 0;
        virtual HRESULT OnUIDeactivate (BOOL fIsUndoable) = 0;
        virtual HRESULT OnDeActivate () = 0;
        virtual HRESULT DiscardUndoState() = 0;
        virtual HRESULT DeactivateAndUndo() = 0;
        virtual HRESULT OnPosRectChanged(LPRECT lprcPosRect) = 0;
        };
// Note 1: This interface is not derived from IOleInPlaceUIWindow.
// Note 2: This object is obtained through QueryInterface on the activation site
```

#### 10.6.8.1. IOleInPlaceSite::CanInPlaceActivate

HRESULT IOleInPlaceSite::CanInPlaceActivate()

This function is called when the IOleInPlaceObject wants to activate for in-place activation. It allows the container application to accept or refuse the activation.

| Argument | Type | Description |
|---|---|---|
| return value | HRESULT | S_OK if the container allows the in-place activation, S_FALSE if the container does not want to allow in-place activation. Other values indicate RPC failures. |

#### 10.6.8.2. IOleInPlaceSite::OnInPlaceActivate

HRESULT IOleInPlaceSite::OnInPlaceActivate()

This function is called when the IOleInPlaceObject is activated in-place. The container should note that the object is becoming active.

| Argument | Type | Description |
|---|---|---|
| return value | HRESULT | S_OK if the container allows the in-place activation. S_FALSE if the container does not allow the activation. Other values indicate RPC failures. |

 © Microsoft Corporation 1992-1993. All Rights Reserved.

### 10.6.8.3. IOleInPlaceSite::OnUIActivate

HRESULT IOleInPlaceSite::OnUIActivate()

This function is called when the IOleInPlaceObject is UI activated in-place. The container should remove any UI associated with its own activation. This is significant if the container is itself an embedded object: in that case it should remove its document level UI.

| Argument | Type | Description |
|---|---|---|
| return value | HRESULT | S_OK if the container allows the in-place activation. S_FALSE if the container does not allow the activation. Other values indicate RPC failures. |

### 10.6.8.4. IOleInPlaceSite::GetWindowContext

HRESULT IOleInPlaceSite::GetWindowContext(pFrame, pDoc, lprcChildPos, lprcClipRect, lpFrameInfo)

This allows the object to retrieve the window interfaces which form the window object hierarchy, and the position in the parent where the object's window should be placed.

| Argument | Type | Description |
|---|---|---|
| ppFrame | IOleInPlaceFrame ** | A pointer to the location where the pointer to the Frame interface will be returned |
| ppDoc | IOleInPlaceUIWindow ** | A pointer to the location where the pointer to the document window interface will be placed. NULL is returned through ppDoc if the document window is the same as the frame window; the object should use ppFrame instead of ppDoc for border negotiation. |
| lprcChildPos | LPRECT | A pointer to the location where the client site will store the rectangle of the object within the parent window. |
| lprcClipRect | LPRECT | A pointer to the location where the client site will store the clipping rectangle to be applied to allow for frame adornments etc. |
| lpFrameInfo | LPOLEINPLACEFRAMEINFO | a pointer to an OLEINPLACEFRAMEINFO structure which the container will fill with appropriate data. |
| return value | HRESULT | S_OK if successful. Other values indicate RPC failures. |

### 10.6.8.5. IOleInPlaceSite::Scroll

HRESULT IOleInPlaceSite::Scroll(scrollExtent)

This function is called by an in-place active object to tell the container to scroll the window by the number of pixels specified by scrollExtent, in each direction. As a result of scrolling, the container should call IOleInPlaceObject::SetObjectRects to update the object's size, position and clipping information.

| Argument | Type | Description |
|---|---|---|
| scrollExtent | SIZE | specifies the number of pixels to scroll in X and Y directions. Positive X means scroll to the right. |
| return value | HRESULT | S_OK if successful. Other values indicate RPC failures. |

### 10.6.8.6. IOleInPlaceSite::OnUIDeactivate

HRESULT IOleInPlaceSite::OnUIDeactivate(fIsUndoable)

OnUIDeactivate is called by an IOleInPlaceObject when it is deactivating. It notifies the container that it should reinstall its own UI components and take focus. Following this call, the object has not yet completed de-activation. The container should wait for the call to OnDeactivate() before fully cleaning up (e.g. destroying submenus etc.). The object indicates whether it can Undo through the fIsUndoable flag. If the object indicates it can undo, the container may (if the user invokes Undo) ask it to Undo through the

 © Microsoft Corporation 1992-1993. All Rights Reserved.

IOleInPlaceObject::ReactivateAndUndo call. The container may tell the object to discard its Undo state and reclaim associated resources either when deactivating the object or by calling IOleObject::DoVerb specifying OLEIVERB_DISCARDUNDOSTATE which does not require deactivation of the object.

| Argument | Type | Description |
|---|---|---|
| fIsUndoable | BOOL | TRUE if the object can undo, FALSE if it cannot. |
| return value | HRESULT | S_OK if successful. Other values indicate RPC failures. |

#### 10.6.8.7. IOleInPlaceSite::OnDeactivate

HRESULT IOleInPlaceSite::OnDeactivate()

OnDeactivate is called by an IOleInPlaceObject when it is fully deactivated. It notifies the container that the object has been deactivated, and it gives the container a chance to run code pertinent to the object's deactivation. In particular, OnDeactivate is called as a result of InPlaceDeactivate, and indicates that the object no longer has Undo state.

| Argument | Type | Description |
|---|---|---|
| return value | HRESULT | S_OK if successful. Other values indicate RPC failures. |

#### 10.6.8.8. IOleInPlaceSite::DiscardUndoState

HRESULT IOleInPlaceSite::DiscardUndoState()

DiscardUndoState is called by the active object when it performs some action that would discard Undo state in the object, to tell the container to discard its state instead.

| Argument | Type | Description |
|---|---|---|
| return value | HRESULT | S_OK if successful. Other values indicate RPC failures. |

#### 10.6.8.9. IOleInPlaceSite::DeactivateAndUndo

HRESULT IOleInPlaceSite::DeactivateAndUndo()

DeactivateAndUndo is called by the active object when the user invokes Undo in a state just after activating the object. The container should UIDeactivate the object, activate itself, and Undo.

| Argument | Type | Description |
|---|---|---|
| return value | HRESULT | S_OK if successful. Other values indicate RPC failures or failures in Windows API calls. |

#### 10.6.8.10. IOleInPlaceSite::OnPosRectChange

HRESULT IOleInPlaceSite::OnPosRectChange(lprcPosRect)

This is called by the active object when editing causes the object to change its extent.

| Argument | Type | Description |
|---|---|---|
| lprcPosRect | LPRECT | a pointer to a RECT containing the new position and size. |
| return value | HRESULT | S_OK if successful. Other values indicate RPC failures or failures in Windows API calls.. |