OLE Controls 96

Distribution: Public - Updated 21th February 1996

DRAFT

Introduction

This document presents various enhancements to the OLE model for the OLE Control 96 specification.

The enhancements are designed to offer improved performance and new features whilst maintaining compatibility with the existing OLE Controls architecture.

The enhancements described in this document include optimizations allowing objects and controls to stay inactive most of the time, drawing optimizations and enhancements, windowless OLE objects, support for non-rectangular and transparent objects.

- Each kind of enhancement is documented as a separate entry with a header giving general information such as main objective and overview.

Mouse interaction, drag & drop for inactive objects

Main objective: Performance, by allowing objects to stay inactive most of the time.

Overview: By allowing inactive OLE objects to minimally interact with the mouse and be drag & drop aware, a new interface, *IPointerInactive*, removes the need for most controls to set the OLEMISC_ACTIVATEWHENVISIBLE flag, and consequently to have a window. This significantly boots performance and reduces the working set.

Motivation

OLE 2 introduces two fundamental states for objects: *active* (in-place or UI active states) or *inactive* (loaded or running states). Only active objects can respond to mouse and keyboard input. They do this by creating a window. Inactive objects are a lot less functional; they are merely able to render themselves and provide a representation of their data in a given format. They have no way to interact directly with the user.

Since they consume a window and are more functional, objects are likely to be bigger and slower when active than when not. Consequently, keeping as few objects as possible active at a given time is generally a performance boost and reduces the overall instance size. Ideally, only one object needs to interact with the user at a given time, usually the one with the keyboard focus. In that scenario, all objects except one could stay inactive. Most applications, however, require a certain amount of participation in the user interface for other (inactive) objects, such as controlling the mouse cursor, firing mouse move events, acting as drag & drop targets, etc.

OLE 2 provides no way for an object to be notified when the mouse cursor is over it. Consequently, an object needs to be active all the time to control the mouse cursor, fire mouse events, or act as a drop target. Most current OLE controls set the OLEMISC_ACTIVATEWHENVISIBLE bit for that very reason. The net result is that most controls on a form tend to be always active, which severely hits load and paint performance, and badly increases the overall working set.

By introducing a new interface, *IPointerInactive*, allowing inactive objects to minimally interact with the mouse, we can lower the need for most objects to set the OLEMISC_ACTIVATEWHENVISIBLE flag.

Mouse interaction for inactive objects

Whenever the container receive a WM_SETCURSOR or WM_MOUSEMOVE message with the mouse pointer over an inactive object supporting *IPointerInactive*, it should call *GetActivationPolicy* on this interface. This method returns a combination of flags from the POINTERINACTIVE enumeration. One of the flags, POINTERINACTIVE_ACTIVATEONENTRY, lets the object request to be in-place activated as soon as the mouse enters it. Any object wishing to do more than setting the mouse cursor and/or firing a mouse move event, such as for example giving a special visual feedback, should use that flag and draw the feedback only when active.

If the object returns this flag, the container should activate it in-place immediately and then forward it the same message that triggered the call to *GetActivationPolicy*. The object will then stay active (and therefore get subsequent messages through its own window) until the container gets another WM_SETCURSOR or WM_MOUSEMOVE, at which point, it should deactivate the object. For windowless OLE objects this mechanism is slightly different and is explained in the section entitled *"Drag & Drop Onto Windowless Objects"*.

If both the POINTERINACTIVE_ACTIVATEONENTRY and POINTERINACTIVE_DEACTIVATEONLEAVE flags are set then the object will only be activated when the mouse is over the object. If only the POINTERINACTIVE_ACTIVATEONENTRY flag is set then the object will only be activated once when the mouse first enters the object.

The information communicated by *GetActivationPolicy* should not be cached. Instead, this method should be called every time the mouse enters an inactive object.

If an inactive object does not request to be in-place activated when the mouse enters it, its container should dispatch subsequent WM_SETCURSOR messages to this object by calling *OnInactiveSetCursor* as long as the mouse pointer stays over the object.

To avoid rounding errors and to make the job easier on the object implementor, this method takes window client coordinates (of the window the object is displayed in, usually the container window) for the mouse position , as opposed to himetrics, as it would normally do for an inactive object. The same coordinates and code path can therefore be used when the object is active and not.

However, window client coordinates would be meaningless for an inactive object unless it was also told its actual position and size on the screen. This is the role of the *lprcBounds* parameter. The container should compute the actual position of the object in window client coordinates and pass it to the object using this parameter. Knowing its actual rectangle on the screen, the object can accurately interpret the mouse coordinates. In essence, this is very similar to what happen for *IViewObject::Draw*, with only a small difference: since no hdc is available, the mouse position and the object rectangle should always be in client coordinates of the containing window, as opposed to the logical coordinates of the hdc.

OnInactiveSetCursor takes an additional parameter (*fSetAlways*) indicating whether the object is obligated to set the cursor or not. Containers should first call this method with this parameter FALSE. The object may return S_FALSE to indicate that it did not set the cursor. In that case, the container should either set the cursor itself, or, if it does not wish to do this, call the *OnInactiveSetCursor* method again with *fSetAlways* being TRUE.

Similarly, the container should dispatch WM_MOUSEMOVE messages to the inactive object under the mouse pointer using *OnInactiveMouseMove.* OLE controls can use this to fire mouse move events.

Drag & drop onto inactive objects

Currently, inactive OLE objects have no simple way to act as potential drop targets. Since they do not have a window they cannot register an *IDropTarget* interface. Even if the container registers a *IDropTarget* for itself, it has no way of knowing whether the inactive object under the mouse pointer wishes to participate in drag & drop or not. Since the cost of activating an object may be high (especially for an out of process servers), most containers simply ignore inactive objects as potential drop targets.

In a situation where most objects stay inactive most of the time, this, of course, would make drag & drop pretty much useless. We need a way for inactive OLE objects to notify their container that they want participate in drag & drop.

Acting as a drop target means for an object to be capable of reacting to a “drag over” notification by performing two different actions:

- indicates whether it would accept a drop at the mouse pointer location, based on the available data formats,
- show the drop location with some kind of visual feedback (outline rectangle, insertion point, etc.)

The first action is quite similar to what happens in *OnInactiveSetCursor* and could be performed by an inactive object. The second action however usually requires the object to synchronously redraw parts of itself, which is not easy for an inactive object.

Given these considerations, trying to get inactive objects to participate in drag & drop while they are inactive seems difficult. Instead, it would be more practical for most objects to tell the container that they wish to be in-place activated as soon as the mouse pointer is dragged over them.

Here is what should happen: to let its inactive embeddings act as potential drop targets the container needs to register a *IDropTarget* interface for its own window. When, inside a call to *IDropTarget::DragOver* (or *DragEnter*), the mouse pointer is over an inactive object, the container QIs that object for *IPointerInactive*. If the object does not support this interface, it is assumed not to be a

potential drop target. If this interface is supported, the container calls *GetActivationPolicy* on it. If the object returns the POINTERINACTIVE_ACTIVATEONDRAG flag, the container in-place activates the object immediately. At that point the object registers an *IDropTarget* interface for its own window (for a windowless object this is a bit different, see the section entitled "Drag & drop onto windowless objects*"*) and the drag & drop operation loop continues using that interface.

When the *GetActivationPolicy* method was called, the container was in the middle of a *DragOver* call on its own *IDropTarget* interface. In order to comply to the contract for this method, it must return a value for the *pdwEffect* parameter. Since it cannot query to object for whether it would accept the drop or not, the container can only safely return DROPEFFECT_NONE here. Note that with windowless OLE objects, however, because it is controlling the drag & drop loop, the container may find out whether the object would accept the drop and return that value instead (see the section entitled *"Drag & drop onto windowless objects"*).

Then, the drag & drop loop continues as if the object had been active in the first place. OLE figures that the object under the mouse pointer changed. The container receives a call to *IDropTarget::DragLeave* and the object gets a call to *IDropTarget::DragEnter*.

If the drop occurs onto the object, the object will probably be UI activated and will get UI-deactivated normally whenever the focus changes again. If the drop did not occur on the object, the container should deactivate the object the next time it gets a call to *IDropTarget::DragEnter*. In some cases the drop may occur on another object that was active originally without the container getting a call to *DragEnter* in the mean-time. In that case the container should try to deactivate the object as soon as it get a chance, for example the next time it UI activates another object.

Container support negotiation

In order for '96 objects to work correctly in down-level containers, they may need to have the OLEMISC_ACTIVATEWHENVISIBLE flag set. However, they also need to let a uplevel container (one that is supporting the mechanism described above) know that there is no need to activate them when visible. This is provided by a new MiscStatus flag:

```
#define OLEMISC_IGNOREACTIVATEWHENVISIBLE ...
```

If the object has set this flag and the container understands and uses *IPointerInactive*, then the container should ignore the OLEMISC_ACTIVATEWHENVISIBLE flag and not in-place activate the object when it becomes visible.

IPointerInactive interface

The *IPointerInactive* interface can be used by the container to let inactive object participate in the interaction with the mouse pointer, including drag & drop.

```
typedef enum
{
      POINTERINACTIVE_ACTIVATEONENTRY  = 1,
      POINTERINACTIVE_DEACTIVATEONLEAVE = 2,
      POINTERINACTIVE_ACTIVATEONDRAG  = 4
}
POINTERINACTIVE;

interface IPointerInactive : public IUnknown
{
    HRESULT GetPointerActivationPolicy([out] DWORD* pPolicy);
    HRESULT OnInactiveSetCursor([in] LPCRECT lprcBounds, [in] LONG x, [in] LONG y,
                        [in] DWORD dwMouseMsg, [in] BOOL fSetAlways);
    HRESULT OnInactiveMouseMove([in] LPCRECT lprcBounds, [in] LONG x, [in] LONG y,
                        [in] DWORD grfKeyState);
};
```

IPointerInactive::GetActivationPolicy

```
HRESULT GetPointerActivationPolicy([out] DWORD* pPolicy);
```

Returns the present activation policy for the object. Objects can request to be in-place activated when the mouse enters them (and in-place deactivated when the mouse leaves them). Any object that want to give more complex visual feedback than simply setting the mouse cursor should request to be in-place activated on entry in order to draw and/or set the mouse capture. Objects can also use this method to request activation when the mouse is dragged over them during a drag & drop operation.

Argument	Type	Description
pPolicy	DWORD*	Activation policy
Return value	S_OK	Success
	E_FAIL	Some unexpected error occurred.

The *pPolicy* member is a combination of the following bits:

POINTERINACTIVE_ACTIVATEONENTRY	The object should be in-place activated in place when the mouse enters it during a mouse move operation.
POINTERINACTIVE_DEACTIVATEONLEAVE	The object should be deactivated when the mouse leaves the object during a mouse move operation.
POINTERINACTIVE_ACTIVATEONDRAG	The object should be in-place activated in place when the mouse is dragged over it during a drag & drop operation.

IPointerInactive::OnInactiveSetCursor

```
HRESULT OnInactiveSetCursor([in] LPCRECT lprcBounds, [in] LONG x, [in] LONG y,
                                                [in] DWORD dwMouseMsg, [in] BOOL fSetAlways);
```

Called by the container for the inactive object under the mouse pointer on receipt of a WM_SETCURSOR message.

Note that window client coordinates (pixels) are used to pass the mouse cursor position as opposed to himetrics. This is made possible by also passing the bounding rectangle of the object in the same coordinate system.

Argument	Type	Description
lprcBounds	LPCRECT	The object bounding rectangle, in client coordinates of the containing window. Let the object know its exact position and size on the screen when the WM_SETCURSOR message was received.
x, y	LONG	Mouse location in client coordinates of the containing window.
dwMouseMsg	DWORD	Specifies the identifier of the mouse message for which a WM_SETCURSOR occurred.
fSetAlways	BOOL	Specifies whether or not the object must set the cursor; if this value is TRUE, the object must set the cursor; if this value is FALSE, the cursor is not obligated to set the cursor, and should return S_FALSE in that case.
Return value	S_OK	Success
	E_FAIL	Some unexpected error occurred.
	S_FALSE	The object didn't set the cursor; the container should either: set the cursor, or call the object again with *fSetAlways* set to TRUE.

IPointerInactive::OnInactiveMouseMove

```
HRESULT OnInactiveMouseMove([in] LPCRECT lprcBounds, [in] LONG x, [in] LONG y,
                        [in] DWORD grfKeyState);
```

Indicates to an inactive object that the mouse pointer has moved over the object. Called by the container for the object under the mouse pointer on receipt of a WM_MOUSEMOVE message.

Note that window client coordinates (pixels) are used to pass the mouse position as opposed to himetrics. This is made possible by also passing the bounding rectangle of the object in the same coordinate system.

Argument	Type	Description
lprc	LPCRECT	The object bounding rectangle, in client coordinates of the containing window. Let the object know its exact position and size on the screen when the WM_MOUSEMOVE message was received.
x, y	LONG	Mouse location in client coordinates of the containing window.
grfKeyState	DWORD	Identifies the current state of the keyboard modifier keys on the keyboard. Valid values can be a combination of any of the flags MK_CONTROL, MK_SHIFT, MK_ALT, MK_BUTTON, MK_LBUTTON, MK_MBUTTON, and MK_RBUTTON.
Return value	S_OK	Success
	E_FAIL	Some unexpected error occurred.

Drawing optimizations

Main objective: Drawing performance.

Overview: A few enhancements to *IViewObject::Draw* that allow to speed up rendering of inactive OLE objects by making a more efficient use of GDI.

Description

A major source of optimization when rendering multiple objects comes from not unnecessarily deselecting and re-selecting font, brush and pen. Because most objects on a form usually tend to share the same font, background color and border types, leaving the font, brush and pen selected on return to *Draw* would often allow the next object to not have to re-select them and would therefore speed up the display. Unfortunately, OLE 2 requires that all GDI objects selected into the hdc passed to *IViewObject::Draw* be unselected before returning.

To enable these optimizations the container can now use the so far reserved *pvApsect* parameter to *Draw* to pass information on:

- whether it is OK to leave font, brush and pen selected on return to *Draw*.

The *pvAspect* parameter (which was supposed to be always NULL in OLE2) can point to a DVASPECTINFO structure passing this information. If this parameter is NULL, objects should assume that the OLE 2 rules are in effect.

Containers that allow font, brush and pen to be left selected on return to *Draw* should always unselect them from the hdc at the end of the drawing process. It is particularly important that they get deselected before returning control to the message loop so that their creator has no chance to delete them while they are still selected into the hdc. Objects should also refrain from deleting font, brush or pen that they left selected in a separate thread for the same reason.

Containers or other objects may not delete any font, brush or pen that an object selected into the rendering hdc. It is the responsibility of the object that created them to delete them. A simple way to implement this is for the object to hold on to the handle and delete it each time it needs a new one. A more sophisticated scheme (usually the case for fonts) would involve a cache of handles using a LRU mechanism to free them.

Note that only the selected font, brush, pen may be changed by *Draw*. The clipping region and selected bitmap must be left unchanged.

DVASPECTINFO structure

The *pvAspect* parameter of the *IViewObject::Draw* method can now point to a DVASPECTINFO structure which contains information needed for various drawing optimizations.

```
typedef enum tagAspectInfoFlag
{
    DVASPECTINFOFLAG_CANOPTIMIZE      = 1,
} DVASPECTINFOFLAG;

typedef struct tagAspectInfoFlag
{   UINT          cb;   // size
    DWORD         dwFlags;
} DVASPECTINFO;
```

Member	Type	Description
cb	INT	Count of bytes in the structure, including this member.
dwFlags	DWORD	See below.

The *dwFlags* parameter can be the following value:

`DVASPECTINFOFLAG_CANOPTIMIZE = 1`

By specifying the can optimize flag, a control may leave objects selected in the draw DC (such as the pen, brush, font, extpen), as well as leaving other drawing state changes in the DC (such as back color, text color, ROP code, current point, line drawing, poly fill mode, etc.). The control may not change other state which another control is not capable of restoring, like mode or anything which changes transformation. Also, the selected bitmap, clip region and metafile may not be changed.

Flicker-free activation/deactivation

Main objective: Suppress flicker when a windowed OLE object in-place activates.

Overview: OLE2 objects have no way of knowing whether their bits are already correct on the screen when they in-place activate. Therefore they must repaint themselves, creating unnecessary flicker. A new interface, *IOleInPlaceSiteEx,* fixes this problem.

Description

When an OLE2 object is in-place activated, it cannot know whether its bits are already correct on the screen and therefore must always repaint itself. Similarly, when an in-place object deactivates, it has no way to let its container know whether its bits are correct on screen at that very moment. Therefore, the container must repaint the object entirely. All this creates unnecessary and undesirable flicker.

To fix this problem, a new site interface, *IOleInPlaceSiteEx*, is derived from *IOleInPlaceSite*. It contains three new methods, *OnInPlaceActivateEx and OnInPlaceDeActivateEx* which replace the already existing method of same name less the "Ex", and the RequestUIActivate method.

```
HRESULT OnInPlaceActivateEx(BOOL* pfNoRedraw, DWORD dwFlags);
HRESULT OnInPlaceDeactivateEx(BOOL fNoRedraw);
HRESULT RequestUIActivate();
```

When *OnInPlaceActivateEx* is called, the container should let the object know whether it needs to redraw by returning the appropriate value in the flag pointed to by *pfNoRedraw.* The container should carefully check the invalidation status of the object, the z-order, clipping and any other relevant parameters to determine the appropriate value for that flag.

Similarly, an object can let the container know whether its bit are currently correct on screen when it deactivates by using the *fNoRedraw* parameter of *OnInPlaceDeActivateEx*.

If the site does not support the *IOleInPlaceSiteEx* interface, objects should use the existing methods on *IOleInPlaceSite* instead. If the site supports *IOleInPlaceSiteEx* but the *IOleInPlaceSite* methods are called, the *fNoRedraw* flag is considered FALSE.

The *dwFlags* parameter is used for windowless objects (see the section entitled *"Window vs. windowless negotiation"*).

A control calls the RequestUIActivate method to determine if UI activation is allowed and to announce the control's intent to transition to the UI active state. A container might return S_FALSE because the currently active control will not validate or the user canceled the focus change from an event handler.

If the control determines that it cannot complete the UI activation sequence after calling RequestUIActivate, the control should call OnUIDeactivate. This gives the container a chance to cleanup.

If a control does not call RequestUIActivate, the container handles data validation and fires Enter and Exit events from IOleInPlaceSite::OnUIActivate.

IOleInPlaceSiteEx interface

This interface provides alternate activation / deactivation notification methods to avoid unnecessary flashing on screen.

```
interface IOleInPlaceSiteEx : public IOleInPlaceSite
{
    HRESULT OnInPlaceActivateEx( [out] BOOL* pfNoRedraw, [in] DWORD dwFlags);
    HRESULT OnInPlaceDeactivateEx( [in] BOOL fNoRedraw);
```

```
    HRESULT RequestUIActivate();
}

//Container implements; called by control
//All of the functions are required if interface is implemented (interface is optional).
```

IOleInPlaceSiteEx::OnInPlaceActivateEx

HRESULT OnInPlaceActivateEx([out] BOOL* pfNoRedraw, [in] DWORD dwFlags);
This method replaces *OnInPlaceActivate.* Windowed objects can use this method to know whether it is necessary to redraw after activating. They may still use the old method, but should always redraw in that case.

Argument	Type	Description
pfNoRedraw	BOOL*	Returns FALSE if the object should redraw after completing the activation, TRUE if it needs not. This parameter can be NULL, in which case there is no need for the container to compute the value of that flag.
dwFlags	DWORD	Communicates extra information to the container, including whether the object is activating windowless or not.
Returned value	HRESULT	As defined for *IOleInPlaceActivate.*

The *dwFlags* parameter can be a combination of the following values:

ACTIVATE_WINDOWLESS = 1	The object is in-place activating windowless. Within this method, the container should use this information as opposed to calling *GetWindow* on the object to determine whether the object is windowless or not.

IOleInPlaceSiteEx::OnInPlaceDeactivateEx

HRESULT OnInPlaceDeactivateEx([in] BOOL fNoRedraw);
This method replaces *OnInPlaceDeactivate.* Windowed objects can use this method to let their container know whether it is necessary to redraw them after deactivating. If an object use the old method, the container should always redraw it in that case.

Argument	Type	Description
fNoRedraw	BOOL	FALSE if the container should redraw the object after completing the deactivation, TRUE if it needs not.
Returned value	HRESULT	As defined for *OnInPlaceDeactivate.*

IOleInPlaceSiteEx::RequestUIActivate

HRESULT RequestUIActivate();
A control should use this method to warn the container it is about to UIActivate. The container may refuse this request by returning S_FALSE, in which case the control must call OnUIDeactivate to allow the container to tidy up.

Argument	Type	Description
Returned value	S_OK	The control may continue the activation process and call OnUIActivate.
	S_FALSE	The control may not UIActivate. The control must call OnUIDeactivate to allow the container to tidy up.

Flicker-free drawing

Main objective: Efficient flicker-free drawing, non-rectangular and transparent objects.

Overview: An extension of the *IViewObject2* interface, *IViewObjectEx*, packages a number of additions to the OLE drawing mechanism to support flicker-free drawing and non rectangular and transparent inactive objects. Containers can now choose between a variety of drawing algorithms, depending on their sophistication and the situation.

Note. Although documented here two pass drawing is not currently utilized by any containers.

Introduction

Flicker is created by redrawing the background before letting an object redraw its foreground. This happens when following the "Painter's Algorithm" (back to front drawing). There are essentially two ways to avoid flickering:

- Draw into an offscreen bitmap and then copy the resulting image to the screen in one chunk. This might require significant additional resources to store the offscreen image, depending on the size of the region to drawn, the resolution and the number of colors.
- Draw front to back, excluding each rectangular area from the clipping region as soon as its has been painted. One benefit of this method is that each pixel is painted only once. Speed depends essentially on the performance of the clipping support.

The major drawback of the first method - consuming memory resources - can be somewhat offset by splitting the region to draw in several areas (bands for example) painted one after another. An advantage of this method is that it works well for non rectangular shapes, since drawing can occur back to front without concern about flickering.

The second algorithm is as efficient as clipping is. In the case of complex forms with hundreds of objects, the clipping region can end up fairly complex and slow down the drawing excessively. Algorithms can be devised to keep the clipping region as simple as possible, but they tend to perform in n^2.

Another problem is that clipping gets very inefficient or even impossible for non rectangular or complex shapes, such as text for example. With the current clipping support in Windows, it is impossible to exclude the text from the clipping region. Support of pixel perfect regions in future versions of Windows may allow such a solution.

Finally, clipping of non-rectangular objects only works for objects prepared to support it, while back-to-front drawing works for all objects. Real applications are likely to contain a mixture of uplevel and downlevel objects, and should be able to give reasonable, fast and flickerless images of all objects.

Consequently, no one method is perfect. Depending on the situation and their sophistication, containers may choose to use one or another, or a mix of both. The objective of the design presented here is to allow the use of both methods with various degrees of sophistication. Simple containers may implement a simplistic "back to front" painting algorithm directly to the screen. The speed is likely to be high but so will flicker. If flicker is to be reduced to a minimum, painting to an off-screen device context is the solution of choice. If memory consumption is a problem, containers can use clipping to reduce the use of off screen bitmaps.

In short, the goal of this design is to allow algorithms and implementation to evolve as the requirements on the application change and computers and Windows evolve.

Two pass drawing

To draw as flicker-free as possible without using an offscreen bitmap, the container will have to paint in two passes. First pass is done front to back. During that pass, each object draws regions of itself that are cheap enough to clip out efficiently and that it can entirely obscure. These regions will be referred to as "opaque" in the rest of this document. After each object is done, the container clips out the regions just painted to ensure that subsequent objects will not modify the bits on the screen.

During the second pass, which occurs back to front, each object draws its remaining parts - irregular, oblique or in general difficult to clip out, such as text on transparent background. Such parts will be referred to as "transparent" in the rest of this document. At this point, the container is responsible for clipping out any opaque, already painted regions in front of the object currently drawing. The less painting during this second pass, the less flicker on the screen.

Clipping during the second pass may be very inefficient, since the clipping region needs to be recreated for every object that has something to draw. This might be acceptable if not too many overlapping objects have irregular or transparent parts. An object can tell its container ahead of time whether it wants to be called during this second pass or not.

If the container provides an offscreen bitmap to paint into, then it can skip the first pass and ask every object to render itself entirely during the second pass. In certain cases, the container may also decide than flicker is not a problem and use that same technique while painting directly on screen. For example, flicker might be acceptable when painting a form for the first time, but not when repainting.

Note. Although documented here two pass drawing is not currently utilized by any containers.

New drawing aspects

Most of the drawing enhancements described in the previous section are packaged into a new interface, derived from *IViewObject2,* and called *IViewObjectEx*.

The discussion above shows that each object needs to be able to draw and return information about three separate aspects of itself:

- its entire image,
- its opaque, easy to clip regions only,
- its irregular or transparent parts only.

The following drawing aspects are needed:

DVASPECT_CONTENT	Same as before: the entire content of an object. All objects should support this aspect.
DVASPECT_OPAQUE	Represents the opaque, easy to clip parts of an object. Objects may or may not support this aspect.
DVASPECT_TRANSPARENT	Represents the transparent or irregular parts of on object, typically parts that are expensive or impossible to clip out. Objects may or may not support this aspect.

The container can determine which of these drawing aspects an object supports by calling the new method *IViewObjectEx::GetViewStatus*. Individual bits return information about which aspects are supported. If an object does not support the *IViewObjectEx* interface, it is assumed to support only DVASPECT_CONTENT.

Depending on which aspects are supported the container may ask the object to draw itself during the front to back pass only, the back to front pass only or both. The various possible cases are:

- Objects supporting only DVASPECT_CONTENT should be drawn during the back to front pass, with all opaque parts of any overlapping object clipped out. Since all objects should support this aspect, a

container not concerned about flickering - maybe because it is drawing in an offscreen bitmap - can opt to draw all objects that way and skip the front to back pass.

- Objects supporting DVASPECT_OPAQUE may be asked to draw this aspect during the front to back pass. The container is responsible for clipping out the object's opaque regions (returned by *IViewObjectEx::GetRegion*) before painting any further object behind it.
- Objects supporting DVASPECT_TRANSPARENT may be asked to draw this aspect during the back to front pass. The container is responsible for clipping out opaque parts of overlapping objects before letting an object draw this aspect.

Even when DVASPECT_OPAQUE and DVASPECT_TRANSPARENT are supported, the container is free to use these aspects or not. In particular, if it is painting in an offscreen bitmap and consequently is unconcerned about flicker, the container may use DVASPECT_CONTENT and a one-pass drawing only. However, in a two-pass drawing, if the container uses DVASPECT_OPAQUE during the front to back pass, then it must use DVASPECT_TRANSPARENT during the back to front pass to complete the rendering of the object.

Object origin and extent

In order not to break the OLE model too much, all objects, rectangular or not, are required to maintain an origin and a rectangular extent. This allows the container to still consider all its embedded objects as rectangles and to pass them appropriate rendering rectangles in *IViewObjectEx::Draw*.

An object's extent depends on the drawing aspect. For non rectangular object, the extent should be the size of a rectangle covering the entire aspect. By convention the origin of an object is the top-left corner of the rectangle of the DVASPECT_CONTENT aspect. In other words, the origin always coincides with the top-left corner of the rectangle maintained by the object's site, even for a non-rectangular object.

All *IViewObjectEx* methods described in this document that take or return a position assume it to be expressed in himetric, relative to the origin of the object.

Getting extents of the drawing aspects

IViewObject2::GetExtent is extended to support the new drawing aspects, but must return the same size as DVASPECT_CONTENT for all the new aspect. (IOleObject::GetExtent must do the same thing.)

In order to let the container easily get the bounding rectangle of the new drawing aspects, a new method *IViewObjectEx::GetRect* is added. This method returns a rectangle (in himetric relative to the origin of the object) defined as follow for the various drawing aspect:

DVASPECT_CONTENT: the returned rectangle corresponds to the object's extent.

DVASPECT_OPAQUE: only objects with a rectangular opaque region should return a rectangle. Others should fail and return error code DV_E_ASPECT. If a rectangle is returned, it is guaranteed to be completely obscured by calling *IViewObjectEx::Draw* for that aspect. The container should use that rectangle to clip out the object's opaque parts before drawing any object behind it during the back to front pass. For objects with a non-rectangular opaque region (when this method fails) the container should use draw the entire object in the back to front part using the DVASPECT_CONTENT aspect (In '97 containers will be able to also use *GetRegion*).

DVASPECT_TRANSPARENT: the returned rectangle should completely cover the transparent region of the object. A container may use this rectangle to determine whether there are other objects overlapping the transparent parts of a given object.

View status

Checking for opaque objects

In order to optimize the drawing process, the container needs to be able to know whether an object is opaque and/or has a solid background. Indeed, it is not necessary to redraw objects that are entirely covered by a completely opaque object. Other operations, such as scrolling for example, can also be highly optimized if an object is opaque and has a solid background.

The *IViewObjectEx::GetViewStatus* method returns whether the object is entirely opaque or not (VIEWSTATUS_OPAQUE bit) and whether its background is solid (VIEWSTATUS_SOLID bit). This information may change in time. An object may be opaque at a given time and become totally or partially transparent later on (for example because of a change of the BackStyle property). Objects should notify their sites when it changes using *IAdviseSinkEx::OnViewStatusChange.* This ensures that the site can cache this information for high speed access.

Objects not supporting *IViewObjectEx* are considered to be always transparent.

Checking for supported drawing aspects

The *IViewObjectEx::GetViewStatus* method also returns a combination of bits indicating which aspects are supported.

If a given drawing aspect is not supported, all *IViewObjectEx* methods taking a drawing aspect as an input parameter should fail and return E_INVALIDARG. The *GetViewStatus* method allows the container to get back information about all drawing aspects in one quick call. Normally the set of supported drawing aspects should not change with time. However if this was not the case, an object should notify its container using *IAviseSinkEx::OnViewStatusChange.*

Which drawing aspects are supported is independent of whether the object is opaque, partially or totally transparent. In particular, a transparent object that does not support DVASPECT_TRANSPARENT should be drawn correctly during the back to front pass using DVASPECT_CONTENT. However, this is likely to result in more flicker.

View status change notification

In order to let objects notify their container when their view status information changes, the *IAdviseSink* interface is extended (derived) into *IAdviseSinkEx*, which contains one extra method, *OnViewStatusChange*.

In order to determine which interface the sink supports, an object must call the *QueryInterface* function using the pointer that was passed to *IViewObject::SetAdvise.*

View change notification

It is important that controls call the IAdviseSink:OnViewChange method whenever the object's view changes even when the control is in place active. Containers' rely on this notification to keep a control's view up-to-date.

IViewObjectEx interface

This interface is an extension to *IViewObject2* providing support for enhanced, flicker-free drawing for irregular and transparent objects. It also support non-rectangular hit-testing, and control sizing (see the sections entitled *"Hit Detection for Non-Rectangular Objects"* and *"Control Sizing"* for a description of these features).

```
enum _DVASPECT2
```

```
{
    DVASPECT_OPAQUE         = 16,
    DVASPECT_TRANSPARENT    = 32
}
DVASPECT2;
// The above is an extension to DVASPECT

interface IViewObjectEx : public IViewObject2
{
    // from IViewObject
    HRESULT Draw( [in] DWORD dwAspect, [in] LONG lindex, [in] void* pvAspect,
                        [in] RGETDEVICE* ptd, [in] HDC hicTargetDev, [in] HDC hdcDraw
                        [in] LPCRECTL lprcBounds, [in] LPCRECTL lprcWBounds,
                        [in] BOOL (CALLBACK* pfnContinue) (DWORD), [in] DWORD dwContinue);
                ...

    // from IViewObject2
    GetExtent( [in] DWORD dwAspect, [in] DWORD lindex, [out] DVTARGETDEVICE* ptd
                        [out] LPSIZEL lpsizel);

    // IViewObjectEx methods
    HRESULT GetRect( [in] DWORD dwAspect, [out] LPRECTL pRect);
    HRESULT GetViewStatus( [out] DWORD* pdwStatus);

    // More methods (see the section entitled Hit Detection For Non-Rectangular Objects)
    ...
};
```

IViewObjectEx::Draw

```
HRESULT Draw( [in] DWORD dwAspect, [in] LONG lindex, [in] void* pvAspect,
                        [in] RGETDEVICE* ptd, [in] HDC hicTargetDev, [in] HDC hdcDraw
                        [in] LPCRECTL lprcBounds, [in] LPCRECTL lprcWBounds,
                        [in] BOOL (CALLBACK* pfnContinue) (DWORD), [in] DWORD dwContinue);
```

Inherited from *IViewObject*. The contract associated with that method is changed as follow:

- new drawing aspects are supported (see the section entitled *"New Drawing Aspects"*),
- may be called to redraw a windowless in-place active object (see the section entitled *"IViewObject::Draw and In-Place Windowless Objects"*),
- the *pvAspect* parameter may be used to pass additional information allowing drawing optimizations (see the section entitled *"Drawing Optimizations"*).

Arguments have the same meaning as defined by OLE2, with the exception of "lprcBounds" that should be NULL when this method is called to redraw an in-place active windowless object.

Argument	Type	Description
dwAspect	DWORD	Aspect to be drawn.
lindex	LONG	As defined by OLE2.
pvAspect	VOID*	Can point to a DVASPECTINFO structure (see the section entitled *"Drawing Optimizations"*) to allow drawing optimizations. If this parameter is NULL, OLE2 rules are in effect.
ptd	RGETDEVICE*	As defined by OLE2.
hicTargetDev	HDC	As defined by OLE2.
hdcDraw	HDC	As defined by OLE2.
lprcBounds	LPCRECTL	Rectangle (in logical coordinates of hdcDraw) where to draw the object. Should be NULL to draw a windowless in-place active object. hdcDraw should be in MM_TEXT with the origin at the client area origin of the containing window in that case.
lprcWBounds	LPCRECTL	As defined by OLE2.

pfnContinue	BOOL (CALLBACK*) (DWORD)	As defined by OLE2.
dwContinue	DWORD	As defined by OLE2.

IViewObjectEx::GetExtent

```
GetExtent( [in] DWORD dwAspect, [in] DWORD lindex, [out] DVTARGETDEVICE* ptd
                    [out] LPSIZEL lpsizel);
```

Inherited from *IViewObject2*. Extents are returned in himetric. This method may fail for the new aspects or return the same rectangle as for the DVASPECT_CONTENT aspect.

IViewObjectEx::GetRect

```
HRESULT GetRect( [in] DWORD dwAspect, [out] LPRECTL pRect);
```

This method returns a rectangle describing a given drawing aspect. The following rectangles should be returned for depending on the drawing aspect :

DVASPECT_CONTENT	Bounding rectangle of the whole object. Top-left corner at the object's origin and size equal to the extent returned by *IViewObjectEx::GetExtent.*
DVASPECT_OPAQUE	Objects with a rectangular opaque region should return that rectangle. Others should fail and return error code DV_E_DVASPECT.
DVASPECT_TRANSPARENT	Rectangle covering all transparent or irregular parts. If the object does not have any transparent or irregular parts, it may return DV_E_ASPECT.

The returned rectangle is in himetric, relative to the object's origin.

Argument	Type	Description
dwAspect	DWORD	Drawing aspect of interest (see above).
lindex	LONG	As defined by OLE2.
lpRectl	LPRECTL	Pointer to returned region rectangle.
Returned value	S_OK	A valid region has been returned.
	DV_E_DVASPECT	This method does not support this aspect. This can be either because the object does not support the aspect or, for DVASPECT_OPAQUE, because this aspect is not rectangular.

IViewObjectEx::GetViewStatus

```
HRESULT GetViewStatus( [out] DWORD* pdwStatus);
```

Returns information about the opacity of the object, and what drawing aspects are supported. This information is returned as a combination of the following bits:

VIEWSTATUS_OPAQUE	Object is completely opaque, i.e., for any aspect, it promises to draw the entire rectangle passed to Draw. If this bit is not set, the object contains transparent parts. If it also support DVASPECT_TRANSPARENT, then this aspect may be used to draw the transparent parts only. This bit applies only to CONTENT related aspects and *not* to DVASPECT_ICON or DVASPECT_DOCPRINT.
VIEWSTATUS_SOLIDBKGND	Object has a solid background (consisting in a solid color, not a brush pattern). This bit is meaningful only if VIEWSTATUS_OPAQUE is set. This bit applies only to CONTENT related aspects and *not* to DVASPECT_ICON or DVASPECT_DOCPRINT.
VIEWSTATUS_DVASPECTOPAQUE	Object supports DVASPECT_OPAQUE. All *IViewObjectEx* methods taking a drawing aspect as a parameter can be called with this aspect.

VIEWSTATUS_DVASPECTTRANSPARENT	Object supports DVASPECT_TRANSPARENT. All *IViewObjectEx* methods taking a drawing aspect as a parameter can be called with this aspect.

Argument	Type	Description
pdwStatus	DWORD	Returns view status (see above).
Returned value	S_OK	This method should never fail.

IAdviseSinkEx interface

Extension of *IAdviseSink* to provide synchronous and partial display invalidation.

```
interface IAdviseSinkEx : public IAdviseSink
{
    HRESULT OnViewStatusChange( [in] DWORD dwViewStatus);
};
```

IAdviseSinkEx::OnViewStatusChange

```
HRESULT OnViewStatusChange( [in] DWORD dwViewStatus);
```

Notifies the sink that a view status of an object has changed.

Argument	Type	Description
dwViewStatus	DWORD	New view status (see *GetViewStatus* method).
Returned value	S_OK	Success.

Windowless OLE objects

Main objective: Allow an OLE object to never consume a window.

Overview: OLE2 requires object to have a window when they are active. This is often an unnecessary burden on small OLE objects such as control and prevent objects to be transparent or non-rectangular. A new kind of OLE objects, called windowless object, can now in-place active without requiring a window. They get user input and services from their container instead.

Why windowless objects ?

Windows have two major drawbacks for OLE objects: they prevent an object to be transparent or non-rectangular when active and they add unwanted and unnecessary instance size to the object. These limitations are usually not insurmountable for big, out of process OLE servers. However, for small objects such as controls, often found in great numbers on a container, this is an issue.

Modern forms often make use of non-rectangular controls such as lines, arrows, shapes with editable labels, round “volume” buttons, fancy looking gauges and switches, “real world” or 3-D objects, etc. These objects cannot be described in term of a rectangle (at best, they need several of them) but still need go in-place or UI activate to interact with the user. Windows, rectangular by nature, are not well suited for them.

Transparent objects, used intensively in slide presentation packages for example, are a variation of non rectangular objects. The most common example is a text control with transparent background. Typically, all editable text over a bitmap such as a watermark belongs to that category. Other examples include fancy shadows, postal stamp like borders, etc. Again, with a window, controls have no hope to stay transparent when active.

Another problem caused by using windows is the unnecessary extra instance size that they carry. Controls are generally small objects, most often in process. Their typical instance size is in the order of 200 bytes. Because typical forms can embed hundreds of controls, keeping the instance size small is critical and so is keeping the loading/instantiation time down. This basically prevents controls from having a window at all times since the instance size and creation speed of hundreds of window would be unacceptable.

An alternate strategy is to create a window only when an object becomes active - when it needs to get user input. But as a consequence, the amount of work needed for the inactive-active transition goes up and the speed of the transition goes down. There are cases when this is a problem: as an example, consider a grid of text boxes. When cursoring up and down through the column, each control must be in-place activated and then deactivated. The speed of the inactive/active transition will directly affect the scrolling speed.

Another case where the burden of creating a window during the inactive-active transition might be a problem is with nested objects. When the user clicks on a object that is 5 levels down the hierarchy of objects, as many windows need to be created.

Finally, the bottom line is that controls actually do not need a window. Services that a window offers can easily be provided via a single shared window (usually the container’s) and a bit of dispatching code. Having a window is mostly an unnecessary complication on the object.

Windowless object model

General

Windowless objects are an extension of normal Compound Document objects. They follow the same in-place activation model and share the same definitions for the various OLE states, with the difference that

they do not consume a window in the in-place and UI active states. They are required to comply with the OLE2 Compound Document specification (including in-place and UI activation).

Windowless objects require special support from their container. In other words, the container has to be specifically writing to support this new kind of OLE objects. However, windowless objects are backward compatible with down level containers. In such containers, they simply create a window when active and behave as normal Compound Document objects.

As other Compound Document objects, windowless objects need to be in-place active to get mouse and keyboard messages. In fact, since an object needs to have the keyboard focus to receive keyboard messages, and having focus implies being UI active for an OLE object, only UI active objects will actually get keyboard messages. Non active controls can still process keyboard mnemonics.

Since windowless objects do not have a window to get input and keyboard message from, they rely on their container for this. The container is responsible for routing user input messages sent to its own window to the appropriate windowless object, via a new interface. Similarly, windowless objects can obtain services from their container such as capture the mouse, setting the focus, getting a hdc to paint in, etc. In addition, the container is responsible for drawing any border hatching as well as the grab handles.

Support for windowless objects is provided by extensions of the *IOleInPlaceObject* and *IOleInPlaceSiteEx* interfaces, *IOleInPlaceObjectWindowless* and *IOleInPlaceSiteWindowless*. By extending (deriving from) existing interfaces rather than creating new ones, no new vtable pointer is added to the object instance. This help keeping the instance size small, an important requirement for controls.

Window vs. windowless negotiation

When a windowless object gets in-place activated, it should query its site for the *IOleInPlaceSiteWindowless* interface. If this interface is supported, the object should call *CanWindowlessActivate* on it to determine if it can proceed and in-place activate without a window.

If the container does not support *IOleInPlaceSiteWindowless* or *CanWindowlessActivate* returns S_FALSE, the windowless object should behave like a normal Compound Document object and create a window.

This negotiation implies that the container needs to be "windowless aware", i.e., to have been written specifically to support windowless objects. Indeed, the rest of this document details many new behaviors expected for a windowless aware container. However, these new requirements are not breaking the overall OLE model and should be implementable on top of existing code without too much difficulty.

The container knows whether or not a given in-place active object has a window by calling *IOleInPlaceObject::GetWindow*. This method should fail (return E_FAIL) for a windowless object. As a convenience for the developer, the CDK for OLE Controls 96 could provide standard support for creating a window on top of a windowless object.

However, nothing in the OLE2 specification states that an object must have created its window by the time it calls *OnInPlaceActivate* on its site. Consequently, many existing OLE objects only create their window after calling this method. This is a problem for windowless objects since, within the *OnInPlaceActivate* call, the container cannot reliably know whether an object is windowless by calling *GetWindow* on the object.

This problem is solved by requiring windowless objects to call *OnInPlaceActivateEx* (as opposed to *OnInPlaceActivate*):

```
HRESULT OnInPlaceActivateEx(BOOL* pfNoRedraw, DWORD dwFlags);
```

The *dwFlags* parameter contains additional information about the activation as a combination of bits. The ACTIVATE_WINDOWLESS bit indicates that the activation happens windowless.

Note: the *pfNoRedraw* parameter serves a complete different purpose and is explained in the section entitled "*Flicker Free Activation and Deactivation"*. Windowless objects usually will not need this parameter and can therefore pass NULL.

Containers may elect to cache the status of the ACTIVATE_WINDOWLESS flag instead of querying the object using *GetWindow* every time they need to know whether an object is windowless or not.

Message dispatching

Since windowless objects do not have a window, we need a mechanism to let the container dispatch mouse and keyboard messages to them. We also need a way to let them capture the mouse and get the keyboard focus.

A windowless OLE object gets messages from its container, via the *OnWindowMessage* method on the new *IOleInPlaceObjectWindowless* interface. The signature of this method is similar to the SendMessage API function, except that

1. It does not take an HWND in parameter,
2. It returns both a HRESULT and a LRESULT code.

Containers should dispatch messages to the appropriate windowless OLE object as described in the following sections.

Objects should refrain from calling DefWindowProc directly. Objects should instead call *IOleInPlaceSiteWindowless::OnDefWindowMessage* to perform the default action. This can be used for messages such as WM_SETCURSOR or WM_HELP, for which the default action is to propagate the message up to the container. This gives the object a way to let the container handle the message before processing it itself. See the *IOleInPlaceSiteWindowless* interface description for details.

Mouse messages and capture

A windowless OLE object can capture the mouse input, by calling *IOleInPlaceSiteWindowless::SetCapture(TRUE)*. Mouse capture can be denied, in which case this method should fail. If the capture is granted, the container must set the Windows mouse capture to its own window and pass any subsequent mouse message on to the object, regardless of whether the mouse cursor position is over this object or not.
The object can later on release the capture by calling *IOleInPlaceSiteWindowless::SetCapture(FALSE)*. The capture can also be released because of an external event (such as the ESC key being depressed). This will normally be notified by a WM_CANCELMODE message (that the container should forward to the object with the keyboard focus).

Containers should dispatch all mouse messages (including WM_SETCURSOR) to the windowless OLE object that has captured the mouse, or, if no object has captured the mouse, to the object under the mouse cursor. Refer to the section entitled "*Summary of message dispatching rules"* for more details.

An object can indicate that it did not process a mouse message by returning S_FALSE. The container should then perform the default behavior for the message. For all mouse message except WM_SETCURSOR, it should just call DefWindowProc. For WM_SETCURSOR the container can either set the cursor itself or do nothing. Objects can also use *IOleInPlaceWindowlessSite::OnDefWindowMessage* to obtain the default message processing from the container. In the case of the WM_SETCURSOR message, this allows an object to take action if the container did not set the cursor.

Keyboard messages and focus

In order to get keyboard focus, a windowless object needs the participation of its container. Not only Windows focus should be set to the container's window, but also the container should dispatch all keyboard messages to the windowless object that actually has the focus.

Therefore, windowless objects should call *IOleInPlaceSiteWindowless::SetFocus* to get the keyboard focus. This method should be called wherever a windowed object would have called the Windows API function SetFocus. Normally this happens during the UI activation process and within the notification methods *IOleInPlaceActiveObject::OnDocWindowActivate* and *OnFrameWindowActivate*.

Containers should implement *IOleInPlaceSiteWindowless::SetFocus* by setting the Windows focus to the window they are using to get keyboard messages, and redirecting any subsequent keyboard message to the object requesting the focus. Note that *SetFocus* can also be called to release the focus (by passing FALSE) without assigning it to any particular other object. In that case, the container should call the Windows® API function with a NULL parameter so that no window has the focus.

Objects can use the *IOleInPlaceSiteWindowless::GetFocus* method to find whether they currently have the focus or not.

Containers should dispatch all keyboard messages to the windowless OLE object with the keyboard focus. Other messages intended for the object with the focus such as IME messages, WM_CANCELMODE or WM_HELP should also be sent to that object. Refer to the following list for a description of the message dispatching rules.

A windowless object can indicate that it did not process a keyboard message by returning S_FALSE. In that case, the container should perform the default process of the message on behalf of the object, i.e., pass the message to DefWindowProc.

Summary of message dispatching rules

The following table summarizes how containers should dispatch messages:

Message	Action
WM_MOUSEMOVE WM_SETCURSOR WM_MOUSEMOVE WM_xBUTTONDOWN WM_xBUTTONUP WM_xBUTTONDBLCLK	Dispatch to windowless OLE object that has captured the mouse, if any. Otherwise, dispatch to the windowless OLE object under the mouse cursor. If there is no such object, the container is free to process the message for itself.
WM_KEYDOWN WM_KEYUP WM_CHAR WM_DEADCHAR WM_SYSKEYDOWN WM_SYSKEYUP WM_SYSDEADCHAR WM_IME_xxx WM_HELP WM_CANCELMODE	Dispatch to the windowless OLE object with the keyboard focus
All other messages	Process in the container

Accelerators

The UI active object checks for its own accelerators in *IOleInPlaceActiveObject::TranslateAccelerator*. This also goes for a windowless object. However, a windowless object cannot send a WM_COMMAND message to itself, as would a windowed OLE object. Therefore, instead of “translating” the key to a command, a windowless object should simply process the key right away. Except for that difference, windowless objects should implement this method as defined by the OLE2 specifications. In particular they should pass the message up to their site if they do not which to handle it, and return S_OK if the

message got “translated” (or more exactly, in the case of a windowless object, “processed”) and S_FALSE if not. Non translated messages will come back to the object via *IOleInPlaceObjectWindowless::OnWindowMessage*.

Note that because the container’s window gets all keyboard input, a UI active object should look for messages sent to that window to find those it needs to process in *TranslateAccelerator.* An object can get its container’s window by calling *IOleWindow::GetWindow*.

Mnemonics

OLE Control mnemonics are handled the same way whether the control is windowless or not. The container gets the control mnemonic table with *IOleControl::GetControlInfo* and calls *IOleControl::OnMnemonic* when it receives a key combination that matches a control mnemonic.

IViewObject::Draw and in-place windowless objects

With OLE2 a container was only responsible for drawing inactive objects, since active objects have their own window and therefore drew themselves independently. This, however, changes with windowless objects. Because they do not have a window of their own, they depend on their container to be drawn even when in-place active.

With OLE 2.0, all inactive drawing was performed using *IViewObject::Draw*:

```
HRESULT Draw(DWORD dwAspect, LONG  lindex, void* pvAspect,
                           RGETDEVICE* ptd, HDC hicTargetDev, HDC hdcDraw
                           LPCRECTL lprcBounds, LPCRECTL lprcWBounds,
                           BOOL (CALLBACK* pfnContinue) (DWORD), DWORD dwContinue);
```

In order to stay consistent with OLE2, the container still uses *IViewObject::Draw* to redraw an in-place active windowless object. When use to draw an in-place active windowless object, this method should be used as follow:

- The “lprcBounds” parameter (the rectangle where to draw the object) should be NULL. The object should use its in-place rectangle - passed by the activation verb or by *IOleInPlaceSite::SetObjectRects* - instead.
- Only the following drawing aspects are allowed in that case: DVASPECT_CONTENT, DVASPECT_OPAQUE and DVASPECT_TRANSPARENT.
- The hdc should be in MM_TEXT mapping mode, with its logical coordinates matching the client coordinates of the containing window. In other words, the hdc should be in the same state as the one normally passed by a WM_PAINT message.

Note that *IViewObject::Draw* can be called with the *lprcBounds* parameter NULL only to draw an in-place active windowless object. In every other situation, this is illegal and should result in a E_INVALIDARG error code.

It is legal, however, to call *IViewObject::Draw* with *lprcBounds* not NULL for a in-place active object. This has the same effect as for an in-place active windowed object: the object should render the requested aspect into the passed hdc and rectangle. A container can use this to render a second, non active “view” of an in-place active object, to print an object, etc.

Drag & drop onto windowless objects

In OLE2, the registration of a *IDropTarget* interface was tied to a window. Because they do not have a window when active, windowless objects cannot register an *IDropTarget* interface. Therefore, they cannot directly participate in the OLE drag & drop loop, but instead, need support from their container.

Here is how it works: windowless objects wishing to be drop targets should still implement the *IDropTarget* interface, but not register it and not make it part of their object identity. The container registers its own *IDropTarget* interface. When, inside a call to *DragEnter* or *DragOver*, the container

figures that the mouse pointer just entered an in-place active windowless object, it calls *GetDropTarget* on its *IOleInPlaceObjectWindowless* interface. The object should return a pointer to its own *IDropTarget* interface if it wants to participate in drag & drop. The interface returned from *GetDropTarget* may be cached for use later.

This is preferred to making the IDropTarget interface part of the object identity (and QI-ing for it) since OLE2 specifically states that it should not be.

The container then calls *DragEnter* on this interface and passes the returned value for **pdwEffect* up when returning from its own *DragOver* or *DragEnter*. From there on, the container forwards all subsequent *DragOver* calls to the windowless object. When the mouse leaves the object (if it does) the container should call *DragLeave* on the object's *IDropTarget* and then release this interface.

Inactive windowless objects that wish to participate in drag & drop should do as explained in the section entitled *"Drag & Drop Onto Inactive Objects"* for windowed objects, i.e., return the POINTERINACTIVE_ACTIVATEONDRAG flag in *GetActivationPolicy*. However, there are some differences due to the absence of the window and the fact that the container can directly get to the object's *IDropTarget* interface. Here is the exact sequence of events that should happen when the mouse is dragged over an inactive windowless object:

- Within a call to its own *DragEnter* or *DragOver*, the container calls *GetActivationPolicy* on the object's *IPointerInactive* interface.
- The object answers by returning the POINTERINACTIVE_ACTIVATEONDRAG flag.
- The container in-place activates the object and then calls IOleInPlaceObjectWindowless::GetDropTarget on the object to get its *IDropTarget* interface or uses the *IDropTarget* interface for the object that was cached from a previous call.
- The container calls *DragEnter* on the object, and then returns from the call to its own to *DragEnter* or *DragOver*, forwarding up the value passed by the object for **pdwEffect*.
- The container forwards all subsequent *DragOver* calls to the object's same method until the mouse leaves the object or a drop occurs.
- If a drop happens the container forwards the *DragDrop* call down to the object.
- If the mouse leaves the object, the container calls *DragLeave* on the object and release its *IDropTarget* interface
- Finally, the container in-place deactivates the object.

Objects can return S_FALSE from *IDropTarget::DragEnter* to indicate that they do not accept any of the data formats in the data object. In that case the container can decide to accept the data for itself and return an appropriate dwEffect from its own DragEnter or DragOver method. Note that objects returning S_FALSE from DragEnter should be prepared to receive subsequent calls to DragEnter without any DragLeave in between. Indeed, if the mouse is still over the same object during the next call to the container's DragOver, the container may decide to try and call DragEnter again on the object.

IOleInPlaceObjectWindowless interface

An extension of *IOleInPlaceObject* allowing a container to dispatch user input to an in-place active windowless object.

```
interface IOleInPlaceObjectWindowless : public IOleInPlaceObject
{
    HRESULT GetWindow([out] HWND* phwnd);
    // Other IOleInPlaceObject methods
    ...


    // Message dispatching
    HRESULT OnWindowMessage([in] UINT msg, [in] WPARAM wParam,
                                    [in] LPARAM lParam, [in] LRESULT* plResult);
```

```
    // Drag & drop
    HRESULT GetDropTarget([out] IDropTarget** ppDropTarget);
};
```

IOleInPlaceObjectWindowless::GetWindow

```
HRESULT GetWindow([out] HWND* phwnd);
```

This method is inherited from *IOleInPlaceObject* but has a slightly extended contract for a windowless object in the sense that it could fail (return E_FAIL). This is how the container knows an object in-place activated without creating a window. Note that returning E_FAIL is defined by the OLE2 specification as "No window handle attached to this object", so no compatibility problem should arise.

IOleInPlaceObjectWindowless::OnWindowMessage

```
HRESULT OnWindowMessage([in] UINT msg, [in] WPARAM wParam,
                                    [in] LPARAM lParam, [in] LRESULT* plResult);
```

This method let the container dispatch a message to a windowless OLE object. Containers should send mouse messages to the object with the mouse capture or under the mouse, keyboard, IME and help messages to the object with the keyboard focus. Refer to the following section for a summary of the message dispatching rules.

Argument	Type	Description
msg	UINT	Message identifier, as passed by Windows
wParam	WPARAM	As passed by Windows
lParam	LPARAM	As passed by Windows
plResult	LRESULT*	Returned Windows result code (as defined by Windows API spec)
Returned value	S_OK	Success.

A windowless control should call *IOleInPlaceSiteWindowless::OnDefWindowMessage* to obtain the default windows processing of the container.

Window Message	**Container Responsibilities**
WM_MOUSEMOVE WM_XBUTTONxxx WM_KEYDOWN WM_KEYUP WM_CHAR WM_DEADCHAR WM_SYSKEYUP WM_SYSCHAR WM_SYSDEADCHAR WM_IME_xxx	The container must pass the message to the default window procedure.
WM_SETCURSOR	The container must process the message as its own.

WM_CONTEXTMENU WM_HELP	

All coordinates passed to a control are specified as client coordinates of the containing window.

IOleInPlaceObjectWindowless::GetDropTarget

```
HRESULT GetDropTarget([out] IDropTarget** ppDropTarget);
```

This method returns a pointer to the object's *IDropTarget* interface. Since they do not have a window, windowless objects cannot register an *IDropTarget* interface. However, if they wish to participate in drag & drop, they should still implement this interface and return it here. They should not make this interface part of their object identity since this is explicitly forbidden by the OLE2 specifications. A container may cache the pointer to the returned interface for use later.

Argument	Type	Description
ppDropTarget	IDropTarget**	Returns a pointer to the object's *IDropTarget* interface.
Returned value	S_OK	Success.
	E_NOTIMPL	The windowless object does not support drag & drop.

IOleInPlaceActiveObject Interface

1 This interface is not changed, but the GetWindow method behaves differently for a windowless control.

IOleInPlaceActiveObject::GetWindow

Always fails and returns E_FAIL for a windowless control. Since this is documented in the current OLE2 specification, there should not be any problem with existing containers and frames.

IOleInPlaceSiteWindowless interface

This site interface allows a windowless object to get services from its container in order to carry drawing related operations.

```
interface IOleInPlaceSiteWindowless : public IOleInPlaceSiteEx
{
    //IOleInPlaceSiteEx methods
    HRESULT OnInPlaceActivateEx( [out] BOOL* pfNoRedraw, [in] DWORD dwFlags);
    HRESULT OnInPlaceDeActivateEx( [in] BOOL fNoRedraw);

    // windowless vs. window negotiation
    HRESULT CanWindowlessActivate();
    HRESULT SetCapture([in] BOOL fCapture);
    HRESULT GetCapture();
    HRESULT SetFocus([in] BOOL fFocus);
    HRESULT GetFocus();
    HRESULT OnDefWindowMessage([in] MSG msg, [in] WPARAM wparam,
                        [in] LPARAM lparam, [out] LRESULT* plResult);

    // … more methods for in-place painting…
};
```

IOleInPlaceSiteWindowless::OnInPlaceActivateEx

HRESULT OnInPlaceActivateEx([out] BOOL* pfNoRedraw, [in] DWORD dwFlags);
This method is inherited from *IOleInPlaceSiteEx*. It replaces *OnInPlaceActivate*. Windowless object are required to use this method instead of *OnInPlaceActivate* in order to let their container know whether they are activating windowless or not.

Argument	Type	Description
pfNoRedraw	BOOL*	See explanation the section entitled *"Flicker-free Activation/Deactivation"*. Windowless objects usually do not need the value returned by this parameter and may pass NULL, therefore saving the container the burden of computing this value.
dwFlags	DWORD	Communicates extra information to the container, including whether the object is activating windowless or not.
Returned value	HRESULT	As defined for *IOleInPlaceActivate.*

The *dwFlags* parameter can be a combination of the following values:

ACTIVATE_WINDOWLESS = 1	The object is in-place activating windowless. Within this method, the container should use this information as opposed to calling *GetWindow* on the object to determine whether the object is windowless or not.

IOleInPlaceSiteWindowless::CanWindowlessActivate

```
HRESULT CanWindowlessActivate();
```

Lets an object know whether it can in-place activate without creating a window. If this method returns S_FALSE, the object should create a window and behave as a normal Compound Document object.

Argument	Type	Description
Returned value	S_OK	Object can in-place activate without creating a window. The container will dispatch events to it using *IOleInPlaceObjectWindowless.*
	S_FALSE	Object can't in-place activate without a window.

IOleInPlaceSiteWindowless::SetCapture

```
HRESULT SetCapture([in] BOOL fCapture);
```

Provides a means for a in-place active windowless object to capture all mouse messages. The container should respond by setting the mouse capture to its own window and dispatching any subsequent mouse message to this object. Also used to release the mouse capture.

Argument	Type	Description
fCapture	BOOL	TRUE to capture the mouse, FALSE to release it.
Returned value	S_OK	Mouse capture has been granted to the object. If called to release the capture, this method should never fail.
	S_FALSE	Mouse capture has been denied to the object.

IOleInPlaceSiteWindowless::GetCapture

```
HRESULT GetCapture();
```

An in-place active windowless object can use this method to figure out whether it still has the mouse capture or not. As an alternative a control may wish to cache whether it has the capture or not.

Argument	Type	Description
Returned value	S_OK	The object currently has mouse capture.

	S_FALSE	The object does not currently have mouse capture.

IOleInPlaceSiteWindowless::SetFocus

```
HRESULT SetFocus([in] BOOL fFocus);
```

A UI active windowless object may call this method to take the keyboard focus, for example in response to a WM_xBUTTONDOWN message being sent via IOleInPlaceSiteWindowless::OnDefWindowMessage. It should call this method in response to standard OLE2 notifications *IOleInPlaceActiveObject::OnFrameWindowActivate(TRUE)* and *OnDocWindowActivate(TRUE)* to ensure that the keyboard focus is set back to its containing window when its Document or Frame gain activation.

The container should implement this method so that it sets the focus to its own window and dispatch all subsequent keyboard message to the calling object.

Argument	Type	Description
fFocus	BOOL	TRUE sets focus to the calling server, FALSE removes focus from the calling server (provided it has the focus).
Returned value	S_OK	The object has been given the keyboard focus. If called to release the focus this method should never fail.
	S_FALSE	Keyboard focus has been denied to the object.

IOleInPlaceSiteWindowless::GetFocus

```
HRESULT GetFocus();
```

An in-place active windowless object can use this method to find out whether it has the keyboard focus or not. As an alternative a control may wish to cache whether it has the focus or not.

Argument	Type	Description
Returned value	S_OK	The object currently has keyboard focus.
	S_FALSE	The object does not currently have keyboard focus.

IOleInPlaceSiteWindowless::OnDefWindowMessage

```
HRESULT OnDefWindowMessage([in] MSG msg, [in] WPARAM wparam,
                    [in] LPARAM lparam, [out] LRESULT* plResult);
```

This method implements the default processing for all messages passed to an object. An object can invoke the default processing for a message by calling this method explicitly.

Argument	Type	Description
msg	UINT	Message identifier, as passed by Windows
wParam	WPARAM	As passed by Windows
lParam	LPARAM	As passed by Windows
plResult	LRESULT*	Returned Windows result code (as defined by Win32® API specification for the DefWindowProc function).
Returned value	S_OK	Success.
	S_FALSE	Depends on the message, see below.

Containers should implement this method as follow:

Window Message	**Container Responsibilities**
WM_MOUSEMOVE WM_XBUTTONxxx	The container must pass the message to the default window procedure (DefWindowProc) and return S_OK. Note that *plResult should contain the value

WM_KEYDOWN WM_KEYUP WM_CHAR WM_DEADCHAR WM_SYSKEYUP WM_SYSCHAR WM_SYSDEADCHAR WM_IME_xxx	returned by DefWindowProc.
WM_SETCURSOR WM_CONTEXTMENU WM_HELP	The container can either process the message as its own and return S_OK or not do anything and return S_FALSE.

In-place drawing for windowless objects

Main objective: Provide windowless OLE objects with a way to redraw themselves when active.

Overview: Windowless OLE object need services to redraw themselves, scroll or show a caret when active. A new interface on the site (*IOleInPlaceSiteWindowless*) provides these services.

Introduction

When active, a normal OLE object can use its own window to redraw itself. With the model described so far, a windowless object relies on its container for being redrawn and can only be entirely redrawn. This may not always be appropriate.

We need to define a way for an object to get a hdc to draw in. However, a particular object has no knowledge of its surrounding environment and therefore cannot safely draw itself without the participation of the container. The clipping region needs to be set properly, the background may need to be repainted, objects in front have to be given a chance to redraw themselves, etc. The same goes for other drawing related services such as scrolling, drawing a caret, etc. The site interface *IOleInPlaceSiteWindowless* is extended to provide these services.

One important characteristic of the *IOleInPlaceSiteWindowless* interface is that all methods take positional information in client coordinates of the containing window, i.e., the window the object is drawn into.

Obtaining / Releasing a device context

To draw on its own when in-place active, an object should:

- call *IOleInPlaceSiteWindowless::GetDC* to get a device context (HDC) to draw into,
- draw into this HDC,
- hand back the HDC to the container by calling *IOleInPlaceSiteWindowless::ReleaseDC*.

When calling *GetDC*, objects pass the rectangle they wish to draw into (in client coordinates of the containing window). The container is expected to intersect this rectangle with the object's site rectangle and clip out everything outside the resulting rectangle. This prevent objects from inadvertently drawing where they are not supposed to.

Containers are also expected to map the device context origin in such a way that the object can draw in client coordinates of the containing window - normally the container's window. If the container is merely passing its window DC, this will be the case automatically. If it is returning another DC, such as an offscreen memory DC for example, then the viewport origin should be set appropriately.

The *GetDC* method also takes various flags indicating what the device context will be used for:

OLEDC_NODRAW — When set, indicates that the object won't use the device context to perform any drawing but merely to get information about the display device. The container should simply pass the window's DC without further processing.

OLEDC_PAINTBKGND — When set, requests that the container paint the background before returning the DC. Objects should set that flag if they are requesting a DC for redrawing an area with transparent background.

OLEDC_OFFSCREEN — When set, informs the container that the object wishes to render in an off-screen bitmap that should then be copied to the screen. An object

should use this flags when the drawing operation it is about to perform generates a lot of flicker. The container is free to honor this request or not. However, if this bit is not set, the container must hand back an on-screen DC. This allows objects to perform direct screen operations such as showing a selection (via an xor operation).

Depending whether it is returning a on-screen or off-screen device context, and how sophisticated it is, container may use one of the following algorithms

On-screen, one pass drawing: in *GetDC* the container should:

- get the window DC,
- if OLEDC_PAINTBKGND is set, draw the DVASPECT_CONTENT aspect of every object behind the object requesting the device context,
- return the DC.

In *ReleaseDC*, the container should:

- draw the DVASPECT_CONTENT of every overlapping object,
- release the DC.

On-screen, two pass drawing: in *GetDC* the container should:

- get the window DC,
- clip out the opaque regions of any overlapping object (these regions do not need to be redrawn since they are already correct on the screen),
- if OLEDC_PAINTBKGND is not set, return the DC,
- otherwise, clip out the opaque parts of the object requesting the DC and draw the opaque parts of every object behind it (going front to back),
- draw the transparent aspects of every object behind (going back to front), setting the clipping region appropriately each time,
- finally return the DC.

In *ReleaseDC*, the container should:

- draw the transparent parts of every overlapping object,
- release the DC.

Off-screen drawing: in *GetDC* the container should:

- create a screen compatible memory device context, containing a compatible bitmap of appropriate size,
- map the viewport origin of the device context to ensure that the calling object can draw using client area coordinates of the containing window,
- if OLEDC_PAINTBKGND is set, draw the DVASPECT_CONTENT of every object behind the calling object,
- return the DC.

In *ReleaseDC* the container should:

- draw the DVASPECT_CONTENT aspect of every overlapping object,

- copy the off-screen bitmap to the screen at the location the calling object originally requested in *GetDC*,
- delete and release the memory DC.

Display Invalidation

In-place windowless objects may need to invalidate regions of their on-screen image. Even though the notification methods in *IAdviseSinkEx* can be used for that purpose, they are not ideal for in-place active objects because they take himetric coordinates and force to hold on to a different interface. In order to simplify and speed up in-place drawing, *InvalidateRect and InvalidateRgn* provide the same functionality on *IOleInPlaceSiteWindowless*.

Objects may not call the Windows API function InvalidateRect and InvalidateRgn directly on the HWND they get from calling *GetWindow* on their site. Indeed, containers may need to perform special processing (such as maintaining a "dirty" state for each site) that would not work correctly in that case.

Scrolling

In-place active windowless objects may need to scroll a given rectangle of their on screen image. This would be typically the case of a multi-line text control. Because of transparent and overlapping objects, the Windows API ScrollWindow and ScrollDC are not appropriate. The *IOleInPlaceSiteWindowless::ScrollRect* let objects perform scrolling.

Containers may implement this method in a variety of ways. However, all of them should account for the possibility that the object requesting scrolling may be transparent or not have a solid background, and that there may be overlapping objects.

The simplest way to implement this method consists in simply redrawing the rectangle to scroll. A refinement would be to use ScrollDC when the object requesting the scroll is opaque and has a solid background and there are not overlapping objects. More sophisticated containers may use the following procedure:

- check whether the object is opaque and has a solid background, using *GetViewStatus.* If not, simply invalidate the rectangle to scroll (a refinement would be to check whether the scrolling rectangle is entirely in the opaque region of a partially transparent object),
- get the window DC,
- clip out the opaque parts (returned by *IViewObjectEx::GetRegion*) of any overlapping object,
- clip out and invalidate the transparent parts of any overlapping object,
- finally call ScrollDC (Window's API),
- redraw the previously invalidated transparent parts of any overlapping object.

All redraw generated by the *Scroll* method should happen synchronously (before this method returns)

Caret support

A windowless object cannot safely show a caret without first checking whether the caret is partially or totally hidden by overlapping objects. In order to make that possible, an object can submit a rectangle to its site using *IOleInPlaceSiteWindowless::AdjustRect* to get it adjusted (reduced) to ensure it fits in the clipping region.

Objects willing to create a caret should submit the caret rectangle to their site that way and use the adjusted rectangle for the caret. If the caret is entirely hidden, this method will return S_FALSE and the caret should not be shown at all in this case.

In a situation where objects are overlapping AdjustRect should return the largest rectangle that is fully visible.

This method can also be used to figure whether a point or a rectangular area is visible or hidden by overlapping objects.

IOleInPlaceSiteWindowless interface

This site interface allows a windowless object to get services from its container in order to carry drawing related operations.

```
interface IOleInPlaceSiteWindowless : public IOleInPlaceSiteEx
{
    //IOleInPlaceSiteEx Methods
    …
    // windowless control redrawing functions
    HRESULT GetDC( [in] LPCRECT lpRect, [in] DWORD dwflags, [out] HDC* phdc);
    HRESULT RereleaseDC( [in] HDC hdc);
    HRESULT InvalidateRect( [in] LPCRECT lprc, [in] BOOL fErase);
    HRESULT InvalidateRgn( [in] HRGN hrgn, [in] BOOL fErase);
    HRESULT ScrollRect( [in] int dx, [in] int dy,
                        [in] LPCRECT lprcScroll, [in] LPCRECT lprcClip);
    HRESULT AdjustRect( [in,out] LPRECT lprc);

};
```

IOleInPlaceSiteWindowless::GetDC

```
HRESULT GetDC( [in] LPCRECT lpRect, [in] DWORD dwflags, [out] HDC* phdc);
```

Provides a means for an object to get a screen (or compatible) HDC from its container. See the section entitled *"Obtaining/Releasing a Device Context"* for an explanation of how containers should implement this method. A device context obtained by *GetDC* should be released by calling *ReleaseDC*.

Argument	Type	Description
lpRect	LPCRECT	The rectangle the object wants to redraw, in client coordinates of the containing window. NULL means the full object's extent.
dwflags	DWORD	A combination of the following bits:
		OLEDC_NODRAW Informs the container that the HDC will not be used for drawing but merely to get device information.
		OLEDC_PAINTBKGND Requests that the container paint the background behind the object before returning the HDC. Objects should use this flag when requesting a DC to paint a transparent area.
		OLEDC_OFFSCREEN Indicates that an offscreen HDC is preferred. The container may honor the request or not. If this bit is cleared, the container must return an on-screen DC.
phdc	HDC*	Pointer to returned HDC. The clipping region should be set so that the object can't paint in any area it is not supposed to. If the object is not opaque, the background should have been painted. If this is a screen HDC, any overlapping opaque areas should be clipped out.
Returned value	S_OK	A valid HDC has been returned
	OLE_E_NESTEDPAINT	Already in the middle of a paint session. i.e., *GetDC* has been called again without *ReleaseDC* being called yet.

IOleInPlaceSiteWindowless::ReleaseDC

```
HRESULT ReleaseDC( [in] HDC hdc);
```

Release a HDC obtained by *GetDC*. Notifies the container that an object is done with this device context. If this device context was used for drawing, the container should ensure that all overlapping objects are repainted correctly. If this was a offscreen device context, its content should also be copied to the screen in the rectangle originally passed to *GetDC*.

Argument	Type	Description
hdc	HDC	Device context to release.

IOleInPlaceSiteWindowless::InvalidateRect

```
HRESULT InvalidateRect( [in] LPCRECT lpRect, [in] BOOL fErase);
```

This method let an object invalidate a given rectangle of its in-place image on the screen.

Argument	Type	Description
lpRect	LPCRECTL	The rectangle to invalidate, in client coordinates of the containing window. NULL means the full object's extent.
fErase	BOOL	Specifies whether the background within the update region is to be erased when the update region is processed. If this parameter is TRUE, the background is erased. If this parameter is FALSE, the background remains unchanged.
Returned value	S_OK	Success
	E_INVALIDARG	Invalid combination of dwFlags

Note that an object is only allowed to invalidate pixels contained in its own site rectangle. Any attempt to invalidate an area outside of that rectangle should result in a no-op.

IOleInPlaceSiteWindowless::InvalidateRgn

```
HRESULT InvalidateRgn( [in] HRGN hrgn, [in] BOOL fErase);
```

This method lets an object invalidate a given region of its in-place active image on the screen.

Argument	Type	Description
hrgn	HRGN	The region to invalidate, in client coordinates of the containing window. NULL means the full object's extent.
fErase	BOOL	Specifies whether the background within the update region is to be erased when the update region is processed. If this parameter is TRUE, the background is erased. If this parameter is FALSE, the background remains unchanged.
Returned value	S_OK	Success
	E_INVALIDARG	Invalid combination of dwFlags

Note that an object is only allowed to invalidate pixels contained in its own site rectangle. Any attempt to invalidate an area outside of that rectangle should result in a no-op.

IOleInPlaceSiteWindowless::ScrollRect

```
HRESULT ScrollRect( [in] int dx, [in] int dy,
                    [in] LPCRECT lprcScroll, [in] LPCRECT lprcClip);
```

Allows an object to scroll an area within its in-place active image on the screen. This method should take in account the fact that the caller may be transparent and that there may be opaque or transparent overlapping objects. See Scrolling for suggestions on algorithms this method may use.

Argument	Type	Description
lprcScroll	LPCRECTL	The rectangle to scroll, in client coordinates of the containing window. NULL means the full object.
lprcClip	LPCRECTL	The rectangle to clip to (same definition as for the Windows API function). Only pixels scrolling into this rectangle are drawn. Pixels scrolling out are not. NULL means no clipping.
dx, dy	int	The amount to scroll by on both axis.
Returned value	S_OK	Success

Regardless of the scrolling and clipping rectangle, only pixels contained in the object's site rectangle will be painted. The area uncovered by the scrolling operation is invalidated and redrawn immediately (before this method returns).

This method should automatically hide the caret during the scrolling operation and move the caret by the scrolling amounts if it is inside the clip rectangle.

IOleInPlaceSiteWindowless::AdjustRect

```
HRESULT AdjustRect( [in,out] LPRECTL lprc);
```

Adjusts a rectangle if it is entirely or partially covered by overlapping, opaque objects. The main usage of this method is to adjust the size of the caret.

Argument	Type	Description
lprc	LPRECTL	The rectangle to adjust.
Returned value	S_OK	The rectangle was not entirely covered. It was adjusted successfully.
	S_FALSE	The rectangle was entirely covered. It was adjusted successfully (width and height are now null).

This method should check overlapping object for transparency in order not to reduce a rectangle that is behind a transparent object.

Hit detection for non-rectangular objects

Main objective: Support non rectangular objects.

Overview: Provide entry point allowing the container to have the OLE object participate in the hit-test logic.

Hit test for points

In order to support hit detection on non-rectangular objects, the container needs a reliable way to ask an object whether or not a given location is inside one of its drawing aspects. This function is provided by *IViewObjectEx::QueryHitPoint*.

Note that since this method is part of the *IViewObjectEx* interface, the container can figure whether an mouse hit is over an object without having to necessarily launch the server. If the hit happens to be inside the object, then it is likely that the object will be in-place activated and the server started.

Normal usage of *IViewObjectEx::QueryHitPoint* is the following: the container quickly determines whether a given location is within the rectangular extent of an object and, if yes, calls *QueryHitPoint* to get confirmation that the location is actually inside the object. The hit location is passed in client coordinates of the container window. Since the object may be inactive when this method is called, the bounding rectangle of the object in the same coordinate system is also passed to this method, similarly to what happens in *IPointerInactive::OnSetCursor*.

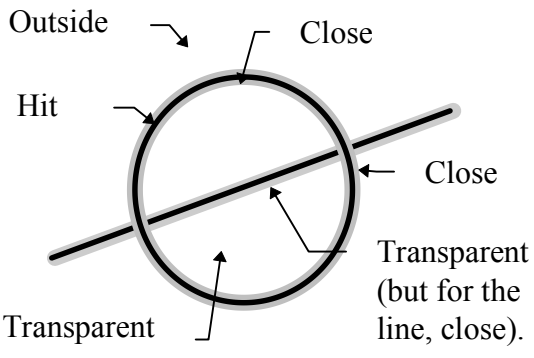
Possible returned values include: outside, on a transparent region, close enough to be considered a hit (may be used by small or thin objects) and hit.

QueryHitPoint is not concerned by the sub-objects of the object it is called for. It merely indicates whether the mouse hit was within an object or not.

QueryHitPoint can be called for any of the drawing aspects an object supports. It should return whether there is a hit with this particular aspect, or fail.

Transparent objects may wish to implement a complex hit-detection mechanism where the user can select either the transparent object or an object behind it depending where exactly the click happens inside the object. For example a transparent TextBox showing big enough text may let the user select the object behind (possibly a bitmap) when he clicks between the characters. For this reason, the information returned by *QueryHitPoint* includes indication about whether the hit happens on an opaque or transparent region.

An example of non-rectangular and transparent hit detection is a transparent circle control with an object behind it (a line in the example below):



The values shown are for hit tests against the circle; gray regions are not part of the control, but are shown here to indicate an area around the image considered “close.” Each object implements its own

definition of “close” but is assisted by a hint provided by the container so that closeness can be adjusted as images zoom larger or smaller.

In the picture above, the points marked “Hit”, “Close” and “Transparent” would all be hits of varying strength on the circle, with the exception of the one marked “Transparent, (but for the line, close).” This illustrates the effect of the different strength of hits. Since the circle responds “transparent” while the line claims “close,” and transparent is weaker than close, the line takes the hit.

Hit test for rectangles

We saw earlier in this document that containers may need to test whether an object overlaps a given drawing aspect of another object. This can be achieved by requesting a region or at least a bounding rectangle of the aspect in question. However, a quicker way to do this would be to simply ask the object whether a given rectangle intersects one of its drawing aspects. The *IViewObjectEx::QueryHitRect* routine serves that purpose.

IViewObjectEx interface (Hit test support)

This interface is an extension of *IViewObject2*. This section details support for non-rectangular hit-testing provided by this interface.

```
typedef enum
{
    HITRESULT_OUTSIDE         = 0,
    HITRESULT_TRANSPARENT     = 1
    HITRESULT_CLOSE           = 2,
    HITRESULT_HIT             = 3,
} HITRESULT;

interface IViewObjectEx : public IViewObject2
{
    // Methods from IViewObject, IViewObject2
    // and methods described in the section entitled
    // “Flicker Free Drawing”.
                ...

    // Hit test support
    HRESULT QueryHitPoint( [in] DWORD dwAspect, [in] LPRECT pRectBounds,
                        [in] POINT ptlLoc, [in] LONG lCloseHint,
                                           [out] DWORD* pHitResult);
    HRESULT QueryHitRect([in] DWORD dwAspect, [in] LPRECT pRectBounds,
                        [in] LPCRECT prclLoc, [in] LONG lCloseHint,
                                           [out] DWORD* pHitResult);
                ...
};
```

IViewObjectEx::QueryHitPoint

```
HRESULT QueryHitPoint( [in] DWORD dwAspect, [in] LPRECT pRectBounds,
                       [in] POINTL ptlLoc, [in] LONG lCloseHint,
                       [out] DWORD* pHitResult);
```

Indicates whether a point is within a given aspect of an object. An object supporting *IViewObjectEx* is required to implement this method at least for the DVASPECT_CONTENT aspect. The object should not take any other action in response to this method other than to return the information, i.e. no side-effects.

Argument	Type	Description
dwAspect	DWORD	Drawing aspect of interest.
prcBounds	LPCRECT	Object bounding rectangle in client coordinates of the containing window. This rectangle is computed and passed by the container so that the object can meaningfully interpret the hit location.
ptLoc	POINTL	Hit location client coordinates of the containing window.

lCloseHint	LONG	Suggested distance in himetric the container considers “close.” This is a hint, and objects can interpret idiosyncratically. Objects can also use this to roughly infer output resolution to choose expansiveness of hit test implementation.
pHitInfo	HITRESULT*	Pointer to returned information about the hit.
Return value	S_OK	Success. pHitInfo contains accurate returned information
	E_FAIL	Method not implemented for this particular aspect. Use DVASPECT_CONTENT instead.

The value returned by pHitInfo is one of the following:

HITRESULT_OUTSIDELocation is outside the object and not close.

HITRESULT_TRANSPARENT	Location is within the bounds of the object, but not close to the image. For example, a point in the middle of a transparent circle could be HITRESULT_TRANSPARENT.
HITRESULT_CLOSE	Location is inside (or perhaps outside) the object but close enough to the object to be considered inside. Small, thin or detailed objects may use this value. Even if a point is outside the bounding rectangle of an object it may still be close (this is needed for hitting small objects).
HITRESULT_HIT	Location is within the image of the object.

IViewObjectEx::QueryHitRect

```
HRESULT QueryHitRect([in] DWORD dwAspect, [in] LPRECT pRectBounds,
                     [in] LPCRECT prclLoc, [in] LONG lCloseHint,
                     [out] DWORD* pHitResult);
```

Indicates whether any point in a rectangle is within a given drawing aspect of an object. An object supporting *IViewObjectEx* is required to implement this method at least for the DVASPECT_CONTENT aspect. The object should not take any other action in response to this method other than to return the information, i.e. no side-effects. If there is any ambiguity about whether a point is a hit, for instance due to coordinates not converting exactly, the object should return HITRESULT_HIT whenever any point in the rectangle *might* be a hit on the object. That is, it is permissible to claim a hit for a point that is not actually rendered, but never correct to claim a miss for any point that is in the rendered image of the object.

Note that unlike QueryHitPoint, this method does not return HITRESULT_TRANSPARENT or HITRESULT_CLOSE. It is strictly boolean: Hit or Miss.

Argument	Type	Description
dwAspect	DWORD	Drawing aspect of interest.
prcBounds	LPCRECT	Object bounding rectangle in client coordinates of the containing window. This rectangle is computed and passed by the container so that the object can meaningfully interpret the hit location.
prcLoc	LPCRECT	Hit test rectangle in himetric, relative to top-left corner of the object.
pHitInfo	HITRESULT*	Pointer to returned information about the hit.
Return value	S_OK	pHitInfo contains accurate returned information
	E_FAIL	Method not implemented for this particular aspect. Use DVASPECT_CONTENT instead.

The value returned by pHitInfo is one of the following:

HITRESULT_OUTSIDENo point in the rectangle is hit.

HITRESULT_HIT	At least one point in the rectangle would be a hit on the object. (See dwHitReason).

Quick Activation

Main objective: Allow controls and containers to avoid performance bottlenecks on loading controls.

Abstract: A new interface is added that combines the load-time or init-time handshaking between the control and its container into a single call.

Overview

The performance interface *IQuickActivate* was added to prevent the performance hit that occurred when a container loaded a control.

In the original model, when a container loaded a control it performed a handshake dance: the container would call certain interfaces on a control and the control, in turn, would call back to certain interfaces on the container's client site. First, the container would create the control and call *QueryInterface* to query for interfaces that it needed; then, the container would call IOleObjectSetClientSite on the control, passing a pointer to its client site. Next, the control would call *QueryInterface* on this site, retrieving a pointer to additional necessary interfaces.

Using the new *IQuickActivate::QuickActivate* function, the container passes a pointer to a QACONTAINER structure, the structure contains pointers to interfaces which are needed by the control and the values of some ambient properties that the control may need. Upon return, the control passes a pointer to a QACONTROL structure that contains pointers to its own interfaces that the container requires, and additional status information.

The *IPersist*::Load* and *IPersist*::InitNew* functions should be called *after* quick activation occurs.

The control should establish its connections to the containers sinks during quick activation. However, these connections are not "live" until *IPersist*::Load* or *IPersist*::InitNew* has been called.

The following ambient flags passed from the container in the dwAmbientFlags member of QACONTAINER for the equivalent standard ambients of type Boolean:

Name	**Value**	**Ambient Dispid**
QACONTAINER_SHOWHATCHING	1	-712
QACONTAINER_SHOWGRABHANDLES	2	-711
QACONTAINER_USERMODE	4	-709
QACONTAINER_DISPLAYASDEFAULT	8	-713
QACONTAINER_UIDEAD	16	-710
QACONTAINER_AUTOCLIP	32	-715
QACONTAINER_MESSAGEREFLECT	64	-706
QACONTAINER_SUPPORTSMNEMONICS	128	-714

```
struct QACONTAINER // from container
{
        ULONG                   cbSize;
        IOleClientSite*         pClientSite;
        IAdviseSinkEx*          pAdviseSink;
        IPropertyNotifySink*    pPropertyNotifySink;
        IUnknown*               pUnkEventSink;
        DWORD                   dwAmbientFlags;             // See QACONTAINER_xxxxx
        flags above
        OLE_COLOR               colorFore;          // ForeColor Ambient, Dispid = -704
        OLE_COLOR                       colorBack;              // BackColor Ambient, Dispid = -
        701
        IFont*                  pFont;              // Font Ambient, Dispid = -703
```

```
        IOleUndoManager*    pUndoMgr;
        DWORD               dwAppearance;       // Appearance ambient, Dispid = -716
        LONG                lcid;               // LocaleID ambient, Dispid = -705
        HPALETTE            hpal;               // Palette ambient, Dispid = -726
        IBindHost*          pBindHost;
};
```

If an interface pointer in the QACONTAINER structure is NULL it does not indicate that the interface is not supported, in this situation the control should use Query Interface to obtain the interface pointer in the standard manner.

```
struct QACONTROL // from control
{
        ULONG           cbSize;
        DWORD           dwMiscStatus;   //IOleObject::GetMiscStatus
        DWORD           dwViewStatus;   //IViewObjectEx::GetViewStatus
        DWORD           dwEventCookie;
        DWORD           dwPropNotifyCookie;
        DWORD           dwPointerActivationPolicy;      //IPointerInactive::GetActivationPolicy
};
```

If all the bits of dwPointerActivationPolicy are set then this indicates that the IPointerInactive interface may not be supported, and QueryInterface should be used to obtain the interface in the standard manner.

IQuickActivate Interface

If this interface is supported then all methods of this interface must be implemented.

```
interface IQuickActivate : public IUnknown
{
        HRESULT QuickActivate(  [in] QACONTAINER* pqacontainer,
                                [out]  QACONTROL* pqacontrol);
        HRESULT SetContentExtent( LPSIZEL lpsizel );
        HRESULT GetContentExtent( LPSIZEL lpsizel );
};
```

IQuickActivate::QuickActivate

HRESULT QuickActivate([in] QACONTAINER* pqacontainer, [out] QACONTROL* pqacontrol);
Allows the container to activate the control.

Argument	Type	Description
pqacontainer	QACONTAINER*	The structure containing information about the container.
pqacontrol	QACONTROL*	The structure the control fills in to return information concerning the control. The container calling QuickActivate must reserve memory for this structure.
Returned value	S_OK	The quick activation is proceeding and the QACONTROL structure has been completed.
	E_FAIL	An unexpected error has occurred. Quick activation will not proceed.

IQuickActivate::SetContentExtent

HRESULT SetContentExtent([in] LPSIZEL lpsizel);
The container calls this to set the content extent of the control.

Argument	Type	Description
lpsizel	LPSIZEL	The size of the content extent.
Returned value	S_OK	The extent has been successfully set.
	E_FAIL	The extent has not been set, the object's size is fixed.

IQuickActivate::GetContentExtent

HRESULT GetContentExtent([out] LPSIZEL lpsizel);
The container calls this to get the content extent of the control.

Argument	Type	Description
lpsizel	LPSIZEL	The size of the content extent.
Returned value	S_OK	The extent has been returned.

Undo

Main objective: Allow containers to implement multi-level undo and redo, incorporating undo for actions performed within contained controls.

Overview: New interfaces are exposed to describe an undo stack, and individual undo objects. The undo design allows compound undo objects (that is, undo objects which nest others).

Motivation

The component-software strategy pursued places demands on Undo/Redo support which cannot be satisfied using the mechanisms defined in OLE 2.0. OLE 2.0 presents a model in which each object manages its own undo state, with the undo context threading through objects with activation and deactivation. However, these assumptions lead to limitations that cripple components:

- Only the UI-active object can contribute undo information for any given action. In the component world, a single action can affect several objects—for example, a user can select several controls on a form and change their color as a single action.
- There is no way to gracefully discard the oldest information when the stack grows too large; the UI-active object can only request the container to discard all undo information generated before this object was UI-activated.
- Undo/Redo UI is managed by the UI-active object. Our strategy breaks this assumption, because the host application owns menus and toolbars, and bears the responsibility for displaying the undo stack; the OLE 2.0 interfaces do not give the host application a way to generate a list of all undoable actions.

To meet these needs, we use a centralized Undo/Redo service, with which objects (whether UI-active or not) can deposit undo information. The centralized undo manager then has the data necessary to support the Undo/Redo UI and to discard undo information gradually as the stack becomes full.

Objects get this central undo manager from their sites through the IServiceProvider interface.

Terminology

Due to the number of objects, controls, and applications which interact as part of this undo architecture, as well as the complexity of their interaction, the documentation of the undo architecture easily becomes cluttered with words that have multiple meanings. For this reason, the term *undo unit* is used in this specification as synonymous with “undo object” and “undo action”. You may discover cases apart from this specification where the terms “undo action” or “undo object” are used to refer to the object that encapsulates a state change that can be undone. However, the use of these terms here would cause confusion, since an undo action is created as a result of a user action, and an OLE object is what creates an undo object.

Overview

The charter of an undo architecture is to allow the user to undo changes they’ve made to objects if they decide they don’t want those changes. From the object’s perspective this means ‘un-modifying’ the state that was modified. In the context of OLE Controls, however, there are two ways the state of an object can be modified: by the user, and by automation code. For example, the text of a TextBox can be changed by the user typing into that TextBox, or by some piece of code that changes the Text property on the TextBox. An example of such a piece of code is an event handler in Visual Basic.

In a simple undo architecture, the two ways of changing an object are treated the same - an *undo unit* (some way of describing what changed) is added to the undo stack in either case. Any time an object's state changes in a way that is meaningful to the user, the object should create an undo unit that is capable of undoing that change. The challenge in incorporating undo into OLE Controls is integrating the side effects of user-written event handlers into the undo stack. That is, when an undoable action is taken on an object, the object will often fire an event as well as adding an undo unit to the undo stack. If the user has written an event handler for this event, how are the actions performed by the event handler integrated with the object's undo unit?

A solution is to distinguish the *cause* of an undo unit. If it is possible to determine what caused an undo unit to be created (whether it was a direct user action or if it was something programmatic in an event handler) then it is possible to determine what should be done with that undo unit. If the user types into a TextBox, then the change in the Text property of that TextBox was caused by a direct user action. If the Text property is changed via a property browser, then the act of applying the changes in the property browser is also a direct user action. Calling SetText programmatically, however, would be a change not caused by a direct user action.. Essentially, whoever is interacting directly with the user (the TextBox when the user types or the property browser) indicates to any object creating an undo unit that the cause of the change is a direct user action.

If an undo unit was not caused by a direct user action, then it is logical that it should not appear on the undo stack, since the existence of the undo stack is solely for the benefit of the user. This means that anytime an object creating an undo unit determines that the undo unit was not caused by a direct user action, it should discard that unit. Not only that, but the *entire* undo stack should be discarded. To explain the necessity of this, consider an object which is manipulated programmatically (e.g. changing the Text property of a TextBox) such that its state has changed outside of the context of a direct user action. It isn't safe for the object to merely throw away the undo unit that encapsulates that change; any undo units already on the stack assume that the object is in a particular state, and discarding the undo unit would violate these assumptions. The safest thing for the object to do in this case is to empty the undo stack. Essentially, to ensure the integrity of the undo stack, *objects **must** either create an undo unit or clear the undo stack when their state changes*.

This architecture is also *hierarchical*, meaning that one undo unit can contain other undo units. This allows complex actions, such as a Wizard or changing a property on a multiple selection, to be presented to the user as a single undoable action. This conforms closely to user expectations. After all, if the user changes the Text property of a selection of three TextBox controls, then the user expects to see a single "Undo Property Changes" action on the stack, not three separate actions.

Symmetric and Non-Symmetric Events

The problem of undo gets particularly tricky when you have to consider user-written event handlers for controls. Consider the following event handler written in Visual Basic:

```
Sub CommandButton1_Click()
       GlobalCounter = GlobalCounter + 1
       Text1.Text = Str(GlobalCounter)
End Sub
```

It is difficult to undo the changes made by this event handler because it affects global state as well as other objects. In order to undo the changes made by this event handler, Text1 would have to create an undo unit for the text change, and the language interpreter would have to create an undo unit to undo the global state change. Requiring language interpreters such as Visual Basic to participate in this fashion is impossible for both technical and pragmatic reasons. An alternative would be to allow the Visual Basic programmer to add his or her own undo units to the undo stack, but such a full-blown undo object model is beyond the scope of this architecture at this time.

The simplest solution is to empty the undo stack anytime an event handler does something that would change the state of an object. This ensures that if there are any undo units on the stack, there is never any "missing" information that might be important to an object. This is, in fact, the basic premise of this architecture; actions taken by user-written event handlers are considered programmatic actions, and as a result will clear the undo stack if they change the state of an object.

However, this simple solution will often result in an undo stack that has nothing on it.

What we do to partially overcome this disadvantage is take advantage of parallels between the user undoing an operation by selecting "Undo" and the user *manually* reversing the operation using the control's normal UI. For example, after a user types text into a text box, there are two main avenues for getting the text back to its original state. The user can undo the Typing operation using the undo stack, or the user can select the text added and delete it. Both of these operations get the TextBox back to its original state. In all but the most rare cases, both avenues will be open to the user. This architecture defines such operations as **symmetric** operations. They are changes that can be undone both by using the undo stack and by the user reversing himself manually. Operations that cannot be manually reversed are **non-symmetric**.

Often an OLE Control will fire an event when the user does something to change it. For example, the TextBox fires a Change event when the user types into it, and the button fires a Click event when the user clicks on it. Events fired by a control are defined as **symmetric** or **non-symmetric** by whether the operation that caused the event to be fired is symmetric or not. For example, the TextBox's Change event is symmetric because the user typing into it is symmetric, while the button's Click event is non-symmetric because there is no way for the user to manually reverse the button click.

Why is defining events as symmetric or non-symmetric necessary? What this categorization does is allow us to define cases where undo units created programmatically can be safely discarded. We've said that any change made by code in a user-written event handler is a programmatic change. As a result, any changes done by a user-written event handler (like setting properties on other controls) would cause the undo stack to be cleared. However, defining symmetric events allows us to say that any undo units created by programmatic changes in a symmetric event handler *can be safely discarded*, as long as the undo unit created by the control fires *that same event* again when performing the undo. To illustrate, consider the following event handler:

```
Sub Text1_Change()
       Text2.Text = Text1.Text
End Sub
```

We've already established that the Change event for a TextBox is symmetric. (If we hadn't, then the programmatic change of setting the Text property on Text2 would clear the undo stack). Since it is symmetric, the undo unit created by Text2 when its Text property changes is merely discarded. Also, the undo unit created by Text1 is designed to fire the Change event again if the user selects "Undo". When the user selects "Undo", Text1's undo unit will change the text in Text1 back to its original value and fire the Change event, which will cause Text2's text to be updated.

Note that the event handler for a symmetric event can do "non-symmetric" things. For example, the Text1_Change() event could increment a global variable. There is no way we can prevent this. The goal, then, is not to produce the "expected" result when the user undoes a symmetric change, at least not if the expected result is that the world returns to the exact state it was in previously. Rather, the goal is that the system will end up in the same state it would be in if the user had reversed the change manually.

Parent Units

In the architecture described above, there is some information that must be communicated to objects, such as whether the current operation is the result of a direct user action and whether an event is symmetric or not. This is done by creating *parent units* that enable certain behaviors. Since this architecture is hierarchical, one undo unit can contain an arbitrary number of other units, which can contain others, etc. These parent units can be put in different states that correspond to the information that needs to be communicated to other objects. When an object needs to create an undo unit, it queries the state of any existing parent units and takes appropriate action depending on the information it obtained.

One type of parent unit indicates that the current operation is the result of a direct user action. Another type indicates that a symmetric event is being fired. In certain cases, a third type of parent is used to indicate that a *non*-symmetric event is being fired. An object that wishes to create an undo unit queries the undo manager for information about what type of parent unit is open, and proceeds either to create

the parent unit or not depending on that data. In many cases, the object will also need to flush the undo stack.

Parent units are simply normal undo units that support containment of other undo units. They are added to the undo stack by "opening" them on the stack, adding their children, and then "closing" them. When a parent unit is open, the central undo manager gives it any new undo units that are added to the stack. Then, when the parent unit is closed, it is put on the top of the undo stack. It is possible to have nested parent units, in which case the outermost open unit is given any new undo units, as well as being the one whose state is returned when an object asks for it. There are two categories of parent units: *standard parent units* and *special parent units*.

Standard parent units are merely regular undo units that add containment behavior so they can contain other undo units. They can encapsulate state changes themselves as well as containing other undo units. These are most often used when an object has to make a complex series of calls to enact a state change, and would prefer to create several undo units to represent that state change instead of just one. Note that error handling requirements limit the dependencies that such units can have on each other (described later). Objects creating standard parent units follow the same procedure as when creating non-parent undo units, which is to query for any special parent units and take the appropriate action.

There are three types of special parent units: *enabling parents, blocking parents,* and *disabling parents.* An enabling parent is used to communicate that the current operation is the result of a direct user operation, and in effect *enables* adding new undo units to the stack. A blocking parent is used to communicate that the current event being fired is a symmetric event and new undo units should be discarded without clearing the undo stack. A disabling parent is used only in certain cases where a parent already exists on the stack and an object wishes to fire a non-symmetric event.

An enabling parent unit is technically no different from a standard parent unit, except in the way it is opened. In fact, once an enabling parent unit is opened, there is no way to distinguish it from a standard parent unit. The difference between an enabling parent and a standard parent is that a standard parent requires an enabling parent to exist before it can be opened, while an enabling parent does not.

A blocking parent unit is a parent unit that discards all undo units handed to it. This type of parent is opened when a symmetric event is fired to prevent new undo units from being added to the stack, and does not require an enabling parent.

A disabling parent unit is used to tell objects that there is no enabling parent. This will cause objects to clear the undo stack if their state changes while the disabling parent is open. This is only used when an enabling parent has been opened, but an object (which did not open the enabling parent) needs to fire a non-symmetric event. The object then adds a disabling parent, which overrides any enabling parents, and fires the event. Then, if the event handler changes the state of any objects, the undo stack will be cleared as required for consistency in the undo stack.

It is possible for special parent units to contain each other. For example, an enabling parent can contain a blocked parent. A blocked parent, however, can never have an open parent inside itself, by definition. It is also possible for an open parent to change state - for example, changing from an enabling parent to a blocked parent and back again. This type of change can only be performed by the object that created the parent, since it has to be done through privately defined interfaces.

Implementation Requirements

Depending on the complexity of the object, the amount of work needed to add undo support to that object can vary. Most OLE Controls need only to implement simple undo units, and do not have to worry about implementing any parent units or the undo manager. This is true if the control fires no events or only fires non-symmetric events. For each state change (e.g. each property change), the object makes the appropriate checks for special parent units (described later), and creates a simple undo unit that it hands to the undo manager. The undo manager is provided by the container of the object. When firing non-symmetric events, the object just needs to make sure there is no open enabling parent. If the control fires non-symmetric events as a result of programmatic changes, then it may need to implement a disabling

parent since an enabling parent may have already been opened by someone else before calling the control.

If the object fires symmetric events, then it will need to implement a blocking parent unit. Implementing a blocking parent that is always blocking is not difficult, since it can never contain children if it remains in the blocked state.

Finally, the control will need to implement an undo manager only if it requires undo in down-level containers. This is typically true if the control is capable of containing other controls.

The Undo Manager

The host application provides the undo manager service, which manages undoable and redoable state changes. This service manages two stacks, the undo and redo stacks, each of which serves as a repository for undo units generated by components or by the host application itself. When an object's state changes, it creates an undo unit encapsulating all the information necessary to undo that change and passes it to the undo manager, which places it on either the undo or the redo stack, as appropriate. When the user selects Undo (or an object programmatically invokes Undo), the undo manager takes the top undo unit off the undo stack, instructs it to carry out its encapsulated actions, and releases it. Similarly, when the user selects Redo, the undo manager takes the top undo unit off the redo stack, instructs it to carry out its encapsulated actions, and releases it.

The undo manager has three states: the base state, the undo state, and the redo state. It begins in the base state. To perform an action from the undo stack it puts itself into the undo state, calls Do() on the undo unit, and goes back to the base state. To perform an action from the redo stack it puts itself into the redo state, calls Do() on the undo unit, and goes back to the base state.

If the undo manager receives a new undo unit while in the base state, it places the unit on the undo stack and discards the entire redo stack; while it is in the undo state, it puts incoming units on the redo stack; and while it is in the redo state, it places them on the undo stack without flushing the redo stack.

The undo manager also allows components to discard the undo or redo stack starting from any object in either stack.

The host application determines the scope of an undo manager. In one application, for example, the scope might be at the document level: a separate undo manager is maintained for each document, and undo is managed independently for each document. However, another application could equally well decide to maintain one undo manager, and therefore one undo scope, for the entire application.

Undo Units

An undo unit encapsulates the information necessary to undo (or redo) a single action. Its principal methods are Do() and GetDescription(). The Do() method implements the actual undo (or redo) operation and shuttles the unit to the other stack—calling Do() on an undo unit in the undo stack creates a corresponding object on the redo stack, and vice-versa. The GetDescription() method returns a user-friendly description of the unit, which is used in the Word-style multi-level undo UI and Edit Menu text.

There are two types of undo units: simple and parent. A simple undo unit encapsulates a single undoable operation, while a parent undo unit can also contain other undo units.

Simple Undo Units

A simple undo unit merely contains the information to undo a single operation; for example, changing the background color of a checkbox or deleting an object. Simple undo units implement IOleUndoUnit, and usually require an *enabling parent*.

A simple undo unit would not require an enabling parent only if it can never be created by a programmatic action. In other words, an undo unit does not need an enabling parent if the only way it can be created is by direct user manipulation of the object creating it.

Parent Undo Units and Nesting

In our component software model, actions which the user perceives as single, atomic, undoable actions may actually involve several components, including components provided by ISV's. For instance, running a wizard which manipulates the form and its controls may involve components provided by the development environment (the window frame), the forms team (the form itself), and ISV's (individual controls on the form). In addition, the wizard manipulates the objects using the primary interfaces they provide; the objects are not directly involved in the semantics of the wizard.

Nevertheless, the user perceives running the wizard as a single action, and expects to see a single "Undo Format Wizard" item in the Undo drop down. To support this behavior, the undo manager must have a single undo unit (provided by the wizard) which encapsulates all the actions of all these objects.

To contain other undo units, an undo unit implements IOleParentUndoUnit, which derives from IOleUndoUnit. The containment behavior is provided by the Open() and Close() methods, which are supported by both the undo manager itself and all parent undo units. Simple undo units are added through Add(); parent units are added through Open(), which leaves the unit open. If the undo manager or a parent undo unit has an open undo unit, it delegates all Add(), Open() and Close() calls to that unit. The open unit composes all the units it receives (via Open() or Add()) into itself; this may be as simple a matter as keeping an ordered list of child units.

The undo manager only concerns itself with top-level undo units, i.e. objects which it did not hand on to an open undo unit. Each parent undo unit is responsible for managing the child units it receives through Open() or Add().

Handling Errors

Having an undo operation fail and leaving the document in an unstable state is something the undo manager, undo units, and the application itself all have to work together to avoid. As a result, there are certain requirements that undo units, the undo manager, and the application or component using undo must conform to.

Error Handling Summary

To perform an undo the undo manager calls Do() on one or more undo units which can, in turn, contain more units. If a unit somewhere in the hierarchy fails, the error will eventually reach the undo manager, who is responsible for making an attempt to roll back the state of the document to what it was before the call to the last top-level unit. It does this by calling Do() on the unit that was added to the redo stack during the undo attempt. If the roll-back also fails, then the undo manager is forced to abandon everything and return to the application. The undo manager indicates whether or not the roll-back succeeded, and the application can take different actions based on this, such as reinitializing components so they're in a known state.

Component or Application Requirements

Typically the component or application using undo puts an undo unit on the stack using the following steps:

1. Create an undo unit (simple or parent) and initialize it with the current state of the document.
2. Pass the created undo unit to the undo manager.
3. Change the state of the object (which will be undone when the undo unit is called).

These steps may be repeated several times to create a parent undo unit that contains other undo units. The requirement for the application is that all three steps must occur atomically; that is, all three steps must succeed or none of them should succeed. Also, undo units cannot depend on each other to the extent that the document is unstable if only some of the units are executed, even when they are siblings in a parent unit. "Unstable" here means having invalid internal state, causing the application to crash, lose data, or otherwise cause the user to not be able to recover from the undo failure. For example, say the user deletes three controls using a multiple selection. If the undo fails leaving only one of the three controls on the

form, that is OK. Having bad internal state such that the application GPFs the next time the user adds a control is not.

Note that if steps (1) or (2) above fail, the application or component has two choices: it can fail the entire operation, or it can clear the undo and redo stacks and continue with the state change. If step (3) fails, then the object must ensure the integrity of the undo stack. This may be done by flushing the undo stack or by creating the undo unit so it doesn't depend on the success of step (3).

The host application that owns the undo manager decides what action(s) to take when undo fails. At the very least it should inform the user of the failure. The host application will be told by the undo manager whether or not the undo succeeded, and if it failed whether or not the attempted rollback succeeded. In the case both the undo and rollback failed, the host application may choose to present the user with several options, including immediately shutting down the application.

Simple Undo Unit Requirements

Simple undo units must not change the state of any object if they return failure. This includes the state of the redo stack (or undo stack if performing a redo). They are also required to put a corresponding unit on the redo (or undo) stack if they succeed. The application should be stable before and after the unit is called.

Parent Undo Unit Requirements

Parent undo units have the same requirements as simple units, with one exception. If one or more children succeeded prior to another child's failure, the parent unit must commit its corresponding unit on the redo stack and return the failure to its parent. If no children succeeded, the parent unit should commit its redo unit only if it has made a state change that needs to be rolled back. For example, say a parent unit contains three simple units. The first two succeed (and add units to the redo stack), but the third one fails. At this point the parent unit commits its redo unit and returns the failure.

A side effect of this is that parent units should never make state changes that depend on the success of their children. Doing this will cause the roll-back behavior to break. If a parent unit makes state changes it should do them before calling any children; if the state change fails, it should not commit its redo unit, not call any children, and return the failure to its parent.

Undo Manager Requirements

The undo manager has one primary requirement for error handling: to attempt roll-back when an undo or redo fails. This discussion applies for both undo and redo, but only undo is mentioned since redo is symmetric.

When the undo manager calls Do() on a top-level undo unit, it puts itself in the undo state. When it is in the undo state it should keep track of whether or not any units have been added to the redo stack as a result of the current undo operation. If the undo fails and nothing has been added to the redo stack, the undo manager does nothing except return the error code (mapping E_ABORT to E_FAIL). If something has been added to the redo stack, the undo manager preserves the error code returned from the undo and calls Do() on the top-level unit on the redo stack. It also passes NULL for the pUndoManager pointer, indicating that the redo units should not put anything back on the undo stack. If the call to the redo unit succeeds, the manager clears both stacks and returns the error code saved from the undo call (mapping E_ABORT to E_FAIL). If the redo unit fails, the manager clears both stacks and returns E_ABORT. (If IOleUndoManager eventually supports rich error reporting then it will be able to give more useful information in this case.)

If the undo manager is calling Do() on more than one undo unit it should only roll-back the one which fails. Note that both stacks are always cleared anytime the undo attempt fails.

Undo Unit Creation

There is a certain sequence to follow when an object's state changes and it needs to create an undo unit. It should also take certain steps anytime it fires a symmetric event. The first piece of information an object needs is what special parent units are open.

Determining the State of Open Parents

Objects need to be able to quickly detect whether there is an open parent undo unit on the stack, and what its state is. This allows the object to take appropriate action depending on the existence and state of an open parent unit.

To handle this, a method on the IOleUndoManager interface is used:

```
#define UAS_NORMAL            0       // Always mask out unused bits with UAS_MASK
#define UAS_BLOCKED           1
#define UAS_NOPARENTENABLE    2
#define UAS_MASK              0x03

interface IOleUndoManager : public IUnknown
{
                  ...
    GetOpenParentState(DWORD* pdwState);
                  ...
};
```

The GetOpenParentState method returns S_OK if there is currently an open parent unit, and S_FALSE if not. The flags indicate if the open unit is a special parent unit, and what type. Objects should mask out unused bits when checking for values returned in pdwState, to allow for future expansion. If there is an open unit, the undo manager delegates through to a corresponding method on the open parent unit:

```
interface IOleParentUndoUnit : public IUnknown
{
                  ...
    GetParentState(DWORD* pdwState);
                  ...
};
```

GetParentState always returns S_OK, but fills in the pdwState member to indicate the open unit's current state. If the open parent contains another open parent, then it delegates to that parent's GetParentState method.

Creating an Undo Unit

Objects which participate in undo are expected to retain a pointer to the undo manager. Before an object creates a simple undo unit, it calls the GetOpenParentState method on the undo manager. If the call returns S_FALSE then there is no enabling parent. If the call returns S_OK but the UAS_NOPARENTENABLE flag is set, then the open parent is a disabling parent. In either of these cases, the object calls DiscardFrom(NULL) on the undo manager and skips creating the undo unit. If the method returns S_OK, but the UAS_BLOCKED flag is set, then the open parent is a blocking parent, and the object does not need to create an undo unit since it would be immediately discarded. If the return value is S_OK and neither of the bit flags are set, then the object creates the undo unit and calls Add() on the undo manager.

To create an enabling parent unit, the object calls GetOpenParentState on the undo manager and checks the return value. If the value is S_FALSE, then the object creates the enabling parent and opens it. If the return value is S_OK, then there is a special parent already open. If the open parent is blocked (UAS_BLOCKED bit set), or an enabling parent (UAS_BLOCKED and UAS_NOPARENTENABLE bits not set), then there is no need to create the enabling parent. If the currently open parent is a disabling parent (UAS_NOPARENTENABLE bit set), then the enabling parent should be created and opened to re-enable adding undo units. Note that UAS_NORMAL has a value of zero, which means it is the absence of all other bits and is not a bit flag that can be set. If comparing *pdwState against UAS_NORMAL, mask out unused bits from pdwState with UAS_MASK to allow for future expansion.

To create a blocked parent, the object calls GetOpenParentState and checks for an open parent that is already blocked. If one exists, then there is no need to create the new blocking parent. Otherwise, the object creates it and opens it on the stack.

To create a disabling parent, the object calls GetOpenParentState and checks for an open parent that is blocked or disabling. If either one exists it is unnecessary to create the new parent. Otherwise, the object creates the parent and opens it on the stack.

In the event that both the UAS_NOPARENTENABLE and UAS_BLOCKED flags are set, the flag that is most relevant to the caller should be used, with UAS_NOPARENTENABLE taking precedence. For example, if an object is creating a simple undo unit, it should pay attention to the UAS_NOPARENTENABLE flag and clear the undo stack. If it is creating an enabling parent unit, then it should pay attention to the UAS_BLOCKED flag and skip the creation.

Firing Events

Here's an example of how a TextBox control might implement its state change when the user types something into the TextBox:

```
CTextBox::MessageHandler()
{
    case WM_CHAR:   // User pressed a key
          Check Undo stack for open parent.
          If necessary, create enabling parent and open on undo stack.
          Call CTextBox::SetText()
          Close enabling parent
}

CTextBox::SetText( )
{
    Check for open enabling parent.
    If one exists
           Create TextChangeUndo undo unit
           Add TextChangeUndo unit to stack
    else
           Clear undo and redo stacks.

    Record the new text value. On failure clear the undo stack.

    Mark parent unit as "blocked" (or open a new blocked unit)
    Fire Change event
    Remove "blocked" status from parent (or close the blocked unit)
}
```

When the parent undo unit is marked "blocked", it discards any undo units it receives.

Note that event sequences are not guaranteed to be the same during an undo as when the user makes the same change. For example, a text box would normally fire a GotFocus or KeyPress event before a Change event. If the user changes the text via undo, then these events might not be fired.

To summarize, if an undoable operation directly corresponds to a user action, then the object should see if there is a blocked parent unit currently open. If so, it can skip creating a unit; otherwise, it should pass the undo unit to the undo manager. That unit may end up being a top level unit or part of a more complex undo unit. If an undoable operation does *not* directly correspond to a user action, then it should make sure there is an unblocked, open unit on the stack before creating the undo unit. If there is no open parent unit then it should not add a new unit to the undo stack and should tell the undo manager to discard the current undo and redo stacks. If there is an open, but blocked, unit, then it should do nothing.

Non-Compliant Objects

Objects which do not support multi-level undo can cause serious havoc. Effectively, they poison their entire context. Since the object cannot be relied on to properly update the undo manager, any units submitted by other objects are also suspect, since the units may rely on the state of the non-compliant object. Attempting to undo a compliant object's units may not be successful, since the state in the non-

compliant object will not match. Worse, the failure of this undo operation may not be detectable—by anyone but the user, that is.

The immediate problem is detecting objects which do not support multi-level undo. This is done by requiring objects which support multi-level undo to mark themselves with a new MiscStatus bit:

#define OLEMISC_SUPPORTSMULTILEVELUNDO ...

When an object without this bit is added to a user-visible undo context, then the safest thing to do is disable the undo UI for this context. Alternatively, a dialog could be presented to the user, asking them whether to attempt to provide partial undo support, working around the non-compliance of the new object.

In addition, non-compliant objects may be added to nested containers. In this case, the nested container needs to notify the undo manager that undo can no longer be safely supported by calling IOleUndoManager::Enable(FALSE).

Repeating Actions

This undo/redo architecture does not support repeating the last action, as Word's "Repeat <action>" menu item does. An application can provide it's own method of implementing this behavior if it likes, perhaps even using the interfaces defined here as a starting point. This architecture may be expanded in the future to include repeat behavior.

Interfaces

```
interface IOleUndoUnit : public IUnknown {
    HRESULT  Do( [in] IOleUndoManager* pUndoManager );
    HRESULT  GetDescription( [out] BSTR* pbstr );
    HRESULT  GetUnitType( [out] CLSID* pclsid, [out] LONG* pnID );
    HRESULT  OnNextAdd( void );
};

interface IOleParentUndoUnit : public IOleUndoUnit {
    HRESULT  Open( [in] IOleParentUndoUnit* pPUU );
    HRESULT  Close( [in] IOleParentUndoUnit* pPUU, [in] BOOL fCommit );
    HRESULT  Add( [in] IOleUndoUnit* pUU );
    HRESULT  GetParentState( [out] DWORD* pdwState );
    HRESULT  FindUnit( [in] IOleUndoUnit* pUU );
};

interface IOleUndoManager : public IUnknown {
    HRESULT  Open( [in] IOleParentUndoUnit* pUU );
    HRESULT  Close( [in] IOleParentUndoUnit* pPUU, [in] BOOL fCommit );
    HRESULT  Add( [in] IOleUndoUnit* pUU );
    HRESULT  GetOpenParentState( [out] DWORD* pdwState );
    HRESULT  DiscardFrom( [in] IOleUndoUnit* pUU );
    HRESULT  UndoTo( [in] IOleUndoUnit* pUU );
    HRESULT  RedoTo( [in] IOleUndoUnit* pUU );
    HRESULT  EnumUndoable( [out] IEnumOleUndoUnits** ppEnum );
    HRESULT  EnumRedoable( [out] IEnumOleUndoUnits** ppEnum );
    HRESULT  GetLastUndoDescription( [out] BSTR* pbstr );
    HRESULT  GetLastRedoDescription( [out] BSTR* pbstr );
    HRESULT  Enable( [in] BOOL fEnable );
};
```

IOleUndoUnit

IOleUndoUnit::Do

HRESULT Do(IOleUndoManager* pUndoManager);

This method instructs the undo unit to carry out the action it encapsulates. Note that if it contains child undo units it must call their Do() methods as well.

The undo unit is responsible for moving itself to the redo (or undo) stack or creating a new undo unit and adding it to the appropriate stack. This is done by creating the undo unit and calling IOleUndoManager::Open or IOleUndoManager::Add; the undo manager will put the undo unit on the undo or redo stack depending on its current state. Parent units should put themselves on the redo (or undo) stack before calling Do() on their children.

If pUndoManager is NULL, the undo unit should perform the undo but not attempt to put anything on the redo (or undo) stack. If pUndoManager is not NULL, then the unit is required to put a corresponding unit on the redo (or undo) stack. Parent units must pass to their children the same undo manager (possibly NULL) that was given to them. It is permissible (but not necessary) when pUndoManager is NULL to open a parent unit on the redo (or undo) stack as long as it is not committed. This allows the use of a blocked parent unit to ensure nothing is added to the stack.

After calling this method, the undo manager will release the undo unit.

See "Handling Errors" for the undo error handling strategy which affects the implementation of this method, particularly for parent units.

Argument	Type	Description
pUndoManager	IOLEUNDOMANAGER*	Supplies the undo manager.
Return Value	HRESULT	S_OK upon success.

IOleUndoUnit::GetDescription

HRESULT GetDescription (BSTR* pbstr)

This method fetches a string which describes the undo unit; this is the string which is displayed in the undo/redo UI. *pbstr is a string allocated with the standard string allocator; the caller is responsible for freeing this string. All units are required to provide a user-readable description of themselves.

Argument	Type	Description
pbstr	BSTR*	Location to put string.
Return Value	HRESULT	S_OK upon success.

IOleUndoUnit::GetUnitType

HRESULT GetUnitType (CLSID* pclsid, LONG* pnID)

This method identifies the type of a unit. A parent undo unit can call this method on its child units to determine whether it can apply special handling to them. The Class ID returned may be the Class ID of the undo unit itself, or of the creating object, or an arbitrary GUID; the only requirement is that the CLSID and LONG together uniquely identify this type of undo unit. The undo unit has the option of returning CLSID_NULL, in which case the caller can make no assumptions about the type of this unit.

Note that the undo manager and parent undo units do not have the option of accepting or rejecting child units based on their type.

Argument	Type	Description
pclsid	CLSID*	Receives a Class ID.
pnID	LONG*	Receives a type ID relative to the returned Class ID.
Return Value	HRESULT	S_OK upon success.

IOleUndoUnit::OnNextAdd

HRESULT OnNextAdd (void)

This method notifies the last undo unit in the collection that a new unit has been added. A parent undo unit should merely call OnNextAdd on its most recently added child undo unit.

This method is useful for supporting fuzzy actions, like typing, which do not have a clear point of termination but instead are terminated when something else happens. An object can create an undo unit for an action and add it to the undo manager but continue inserting data into it through private interfaces. When the undo unit receives an OnNextAdd notification, it communicates back to the creating object that the context has changed, and the creating object stops inserting data into the undo unit.

Note that parent units merely delegate this method to their most recently added child unit. A parent unit should terminate communication through any private interfaces when it is closed. A parent unit knows it is being closed when it returns S_FALSE from IOleParentUndoUnit::Close().

OnNextAdd may not always be called if the undo manager or an open parent unit chooses to discard the unit by calling Release(). This implies that any connection which feeds data to an undo unit "behind the scenes" (as described above and in "Open-Ended Undoable Actions: Typing*"*) should not AddRef the undo unit.

Implementations of OnNextAdd should always return S_OK. The HRESULT return type is provided only for remotability.

IOleParentUndoUnit

This interface is supported by undo units which are capable of containing other units. It supports all of the IOleUndoUnit methods, and supports the following methods in addition:

IOleParentUndoUnit::Open

HRESULT Open(IOleParentUndoUnit* pPUU)

This method opens a new parent undo unit, which will become part of the containing unit's undo stack. The given unit is left open and is passed any additional units (via Add or Open) until the Close method is called. The given unit is not added to the undo stack until the Close method is called with fCommit TRUE.

The parent undo unit (or undo manager) must contain any undo unit given to it unless it is blocked. If it is blocked, it must return S_OK but should do nothing else.

Argument	Type	Description
pPUU	IOleParentUndoUnit*	Supplies the undo unit to open
Return Value	HRESULT	S_OK upon success or if blocked.

IOleParentUndoUnit::Close

HRESULT Close(IOleParentUndoUnit* pPUU, BOOL fCommit)

This method closes the most recently opened undo unit. The fCommit parameter indicates whether the undo unit should be kept or discarded. This allows the client to discard a complex undo unit under construction if an error or cancellation occurs: the client can use the undo manager to build up this complex undo unit, rather than building it itself and only adding it on completion.

The pPUU parameter should point to the currently open undo unit. If this parameter does not match the currently open undo unit then implementations of this method should return E_INVALIDARG without changing any internal state. The only exception to this is if the unit is blocked (see below).

To process a Close, a parent undo unit first checks to see if it has an open child unit. If it does not, it returns S_FALSE; if it does, it calls Close on the child. If the child returns S_FALSE, then the parent undo unit verifies that *pPUU* points to the child unit, and closes that child undo unit. If the child returns S_OK then it handled the Close internally and its parent should do nothing. An error return indicates a fatal error condition. The unit (or undo manager) must accept the undo unit if fCommit is TRUE.

If the parent unit is blocked, it should check the *pPUU* parameter to determine the appropriate return code. If *pPUU* is pointing to itself, then it should return S_FALSE. Otherwise it should return S_OK. The fCommit parameter is ignored, and no action is taken.

Note that a parent undo unit knows it is being closed when it returns S_FALSE from this method. At that time it should terminate any private communication with other objects which may be giving data to it.

Argument	Type	Description
pPUU	IOleParentUndoUnit*	Pointer to the currently open undo unit.
fCommit	BOOL	Indicates whether to keep the undo unit. TRUE if the undo unit should be kept in the collection, FALSE if it should be discarded.
Return Value	HRESULT	S_OK for success if an open child existed; S_FALSE if the undo unit does not have an open child; if the unit is blocked, see the text above for the appropriate return code; E_INVALIDARG if *pPUU* does not point to the currently open undo unit.

IOleParentUndoUnit::Add

HRESULT Add(IOleUndoUnit* pUU)

This method adds a simple undo unit to the collection. The parent undo unit (or undo manager) must accept any undo unit given to it unless it is blocked. If it is blocked, it should do nothing but return S_OK.

Argument	Type	Description
pUU	IOleUndoUnit*	Simple unit to add.
Return Value	HRESULT	S_OK on success or if blocked.

IOleParentUndoUnit::GetParentState

HRESULT GetParentState(DWORD* pdwState)

This method returns state information about the inner-most open parent undo unit. The pdwState parameter should be filled in with UAS_NORMAL to indicate a normal, unblocked state where new undo units will be accepted.

If checking for a normal state, unused bits in pdwState should be masked out to allow for future expansion. The UAS_MASK value is provided for this:

```
fNormal = ((pdwState & UAS_MASK) == UAS_NORMAL).
```

The bit values that can be set in pdwState are:

UAS_BLOCKED: The currently open undo unit will reject any undo units added via Open() or Add(). The caller need not create any new units since they will just be rejected.

UAS_NOPARENTENABLE: The currently open undo unit will accept new units, but the caller should act like there is no currently open unit. This means if the new unit being created requires a parent, then this parent does not satisfy that requirement and the undo stack should be cleared.

If the unit has an open child it should delegate this method to that child. If not, it should fill in *pdwState appropriately and return. Note that a parent unit must never be blocked while it has an open child. If this happened it could prevent the child unit from being closed, which would cause serious problems.

Argument	Type	Description
pdwState	DWORD*	Place to put state information.
Return Value	HRESULT	S_OK

IOleParentUndoUnit::FindUnit

HRESULT FindUnit(IOleUndoUnit* pUU)

Indicates if the specified unit is a child of this undo unit or one of its children. This is normally called by the undo manager in its implementation of DiscardFrom in the (probably rare) event the unit being discarded is not a top-level unit. The parent unit should look in its own list first, then delegate to each child that is a parent unit (determined by doing a QueryInterface for IOleParentUndoUnit).

Argument	Type	Description
pUU	IOleUndoUnit*	Unit to find.
Return Value	HRESULT	S_OK if the given unit is in this undo unit's list of children or is a descendant of this undo unit, S_FALSE otherwise. An error indicates an RPC failure condition.

IOleUndoManager

This interface is implemented by the undo manager service, and is the interface objects use to manipulate the undo and redo stacks.

IOleUndoManager::Open

HRESULT Open(IOleParentUndoUnit* pPUU)

This method is implemented the same as IOleParentUndoUnit::Open. If the undo manager is disabled it should return S_OK and do nothing else.

Argument	Type	Description
pPUU	IOleParentUndoUnit*	Supplies the undo unit to open
Return Value	HRESULT	S_OK on success, if disabled, or if an open unit is blocked.

IOleUndoManager::Close

HRESULT Close(IOleParentUndoUnit* pPUU, BOOL fCommit)

This method is implemented the same as IOleParentUndoUnit::Close. Which stack the unit is added to is determined the same way as explained in IOleUndoManager::Add. If the undo manager is disabled it should immediately return S_OK.

Argument	Type	Description
pPUU	IOleParentUndoUnit*	Pointer to the currently open unit.
fCommit	BOOL	Indicates whether to keep the undo unit. TRUE if the undo unit should be kept in the collection, FALSE if it should be discarded.
Return Value	HRESULT	S_OK for success or if disabled, S_FALSE if there is no open undo unit. E_INVALIDARG if *pPUU* is not the currently open undo unit.

IOleUndoManager::Add

HRESULT Add(IOleUndoUnit pUU)

This method is implemented the same as IOleParentUndoUnit::Add.

If the undo manager is in the base state, it should put the new unit on the undo stack and discard the entire redo stack. In the undo state it should put new units on the redo stack, and in the redo state it should put units on the undo stack without affecting the redo stack. (See "The Undo Manager")

Argument	Type	Description
pUU	IOleUndoUnit*	Simple unit to add.
Return Value	HRESULT	S_OK on success, if disabled, or if an open unit is blocked.

IOleUndoManager::GetOpenParentState

HRESULT GetOpenParentState(DWORD* pdwState)

If there is no open parent unit then this method returns S_FALSE. Otherwise it delegates to the open unit's GetParentState method and returns S_OK.

If the undo manager is disabled, it should fill pdwState with UAS_BLOCKED and return S_OK.

Argument	Type	Description
pdwState	DWORD*	State of the currently open parent unit is returned here.
Return Value	HRESULT	S_OK if there is an open unit, S_FALSE if not.

IOleUndoManager::DiscardFrom

HRESULT DiscardFrom(IOleUndoUnit* pUU)

This method instructs the undo manager to discard the specified undo unit and all undo units below it on the undo or redo stack. The undo manager first searches the undo stack for the given unit, and if not found there searches the redo stack. Once found, the given unit and all below it on the same stack are discarded. The undo unit may be a child of a parent unit contained by the undo manager (determined by calling IOleParentUndoUnit::FindUnit); if it is, then the root unit containing the given unit and all units below it on the appropriate stack are discarded.

If there is an open parent unit and DiscardFrom(NULL) is called, the undo manager should immediately release and discard the open parent unit (do not call Close first). When the object that opened the parent unit attempts to close it, IOleUndoManager::Close will return S_FALSE. If pUU is not NULL, then any open parent units should be left open.

Argument	Type	Description
pUU	IOleUndoUnit*	pointer to the undo unit to discard; NULL for both the entire undo and redo stacks.
Return Value	HRESULT	S_OK upon success; E_INVALIDARG if the specified undo unit is not found in the undo stack. E_UNEXPECTED if disabled.

IOleUndoManager::UndoTo

HRESULT UndoTo (IOleUndoUnit* pUU)

This method instructs the undo manager to perform actions back through the undo stack, down to and including the specified undo unit. Note that the specified undo unit must be top-level (typically retrieved through EnumUndoable). The undo manager simply invokes the Do method on each top-level undo unit and releases it.

In the case an error is returned from the undo unit, the undo manager needs to attempt to roll back the state of the document by performing actions on the redo stack, and no matter what the success of the rollback the undo manager should always clear both stacks before returning the error. If the undo manager has called Do() on more than one top-level unit it should only roll back the unit that returned the error; the top-level units that succeeded should not be rolled back. The undo manager must also keep track of whether or not units were added to the opposite stack so it won't attempt rollback if nothing was added. "Handling Errors" describes in detail the behavior the undo manager must provide with regard to handling errors.

E_ABORT is a special error code used to indicate that both the undo attempt and the roll-back attempt failed. The undo manager should never propagate E_ABORT obtained from a contained undo unit; it should map E_ABORT returned from other undo units to E_FAIL.

Argument	Type	Description
pUU	IOleUndoUnit*	Pointer to the top-level undo unit to undo, or NULL to indicate the most recently added top-level undo unit.
Return Value	HRESULT	S_OK; E_INVALIDARG if the specified undo unit is not in the undo stack. E_ABORT if both the undo and rollback attempt failed. E_UNEXPECTED if disabled.

IOleUndoManager::RedoTo

HRESULT RedoTo (IOleUndoUnit* pUU)

This method instructs the undo manager to perform actions back through the redo stack, down to and including the specified undo unit. Note that the specified unit must be top-level (typically retrieved through EnumRedoable). The undo manager simply invokes the Do method on each top-level undo unit and releases it.

Error handling is performed the same way as for UndoTo, with the undo stack used for rollback.

Argument	Type	Description
pUU	IOleUndoUnit*	Pointer to the top-level undo unit to redo, or NULL to indicate the most recently added top-level undo unit.
Return Value	HRESULT	S_OK; E_INVALIDARG if the specified undo unit is not in the redo stack. E_ABORT if both the undo and rollback attempt failed. E_UNEXPECTED if disabled.

IOleUndoManager::EnumUndoable

HRESULT EnumUndoable (IEnumOleUndoUnits** ppEnum)

This method enumerates the top-level units in the undo stack.

Argument	Type	Description
pEnum	IEnumOleUndoUnits*	Enumeration of the top-level undoable units in the undo stack.
Return Value	HRESULT	S_OK upon success. E_UNEXPECTED if disabled.

IOleUndoManager::EnumRedoable

HRESULT EnumRedoable (IEnumOleUndoUnits** ppEnum)

This method enumerates the top-level units in the redo stack.

Argument	Type	Description
pEnum	IEnumOleUndoUnits*	Enumeration of the top-level redoable units in the redo stack.
Return Value	HRESULT	S_OK upon success. E_UNEXPECTED if disabled.

IOleUndoManager::GetLastUndoDescription

HRESULT GetLastUndoDescription (BSTR* pstr)

This method fetches the description for the top-level undo unit that is on top of the undo stack. It provides a convenient shortcut for the host application to add a description to the Edit Undo menu item. *pbstr is a string allocated with the standard string allocator. The caller is responsible for freeing this string. If the return value is S_OK then the string will always contain a valid description.

Argument	Type	Description
pbstr	BSTR*	Description of transaction on top of stack.

Return Value	HRESULT	S_OK upon success; E_FAIL if the stack is empty. E_UNEXPECTED if disabled.

IOleUndoManager::GetLastRedoDescription

HRESULT GetLastRedoDescription (BSTR* pstr)

This method fetches the description for the top-level undo unit that is on top of the redo stack. It provides a convenient shortcut for the host application to add a description to the Edit Redo menu item. *pbstr is a string allocated with the standard string allocator. The caller is responsible for freeing this string. If the return value is S_OK then the string will always contain a valid description.

Argument	Type	Description
pstr	BSTR*	Description of transaction on top of stack.
Return Value	HRESULT	S_OK upon success; E_FAIL if the stack is empty. E_UNEXPECTED if disabled.

IOleUndoManager::Enable

HRESULT Enable (BOOL fEnable)

This method enables or disables the undo manager. It is particularly useful when a container creates an object which does not support multi-level undo (i.e. does not have the OLEMISC_SUPPORTSMULTILEVELUNDO bit set in its miscellaneous flags). See each method of IOleUndoManager for the appropriate action to take when the manager is disabled. The manager cannot be disabled if there are any open undo units on the stack or if it is the process of performing an undo or redo.

The undo manager should clear both stacks when making the transition from enabled to disabled.

Argument	Type	Description
fEnable	BOOL	TRUE if the undo manager should be enabled; FALSE if it should be disabled.
Return Value	HRESULT	S_OK upon success. E_UNEXPECTED if there is an open undo unit on the stack or the undo manager is performing an undo or redo.

Examples and Walk-Throughs

Simple Undo and Redo

The user changes the color of a control on a form from blue to red, undoes that change, and finally decides to redo it and save the form, committing all changes (which discards the undo state).

When the user changes the color of the control to red, the control constructs an undo unit describing this property change (Property = color, OldValue = blue) and gives it to the undo manager by calling Add. The undo manager AddRef's the undo unit and puts it on top of the undo stack. The undo manager also flushes the redo stack, since it's in the base state. There is no need for the control to check for an enabling parent since the only way the color of the control can be changed is directly by the user (this control has no programmatic access to its color, even through a property browser).

When the user selects Undo, the host application calls IOleUndoManager::UndoTo(NULL) to undo the last top-level unit. The undo manager takes the control's undo unit off the top of the undo stack, puts itself into the undo state, and invokes the undo unit's Do method.

The undo unit's Do method changes the control's color back to blue, adjusts its internal state (OldColor = red), and gives itself back to the undo manager, through Add. Since the undo manager is in the undo state, it AddRef's the undo unit (which now has a reference count of two) and puts it on the redo stack.

After the undo unit's Do method returns, the undo manager releases the undo unit (reducing its reference count to one) and puts itself back into the base state.

When the user selects Redo, the host application calls IOleUndoManager::RedoTo(NULL) to redo the last top-level unit. The undo manager takes the control's undo unit off the top of the redo stack, puts itself into the redo state, and invokes the undo unit's Do method.

The undo unit's Do method changes the control's color back to red, adjusts the its internal state (OldColor = blue), and gives itself back to the undo manager, through Add. Since the undo manager is in the redo state, it AddRef's the undo unit (which now has a reference count of two) and puts it on the undo stack.

After the undo unit's Do method returns, the undo manager releases the undo unit (reducing its reference count to one) and puts itself back into the base state.

When the user saves the form, the form calls IUndoManager::DiscardFrom(NULL) to discard all undo state information. The undo manager takes the undo unit off the undo stack and releases it; this reduces the undo unit's reference count to zero, so it destroys itself.

Compound Undo

The user selects five controls on a form and changes their color to red via a property browser, and then decides to undo that action and restore the controls to their original colors.

When the user presses the Apply button to change the controls' color, the property browser constructs an enabling parent undo unit, sets its description to "Change color of controls", and hands it to the undo manager through the IOleUndoManager::Open method. This method leaves the undo unit open. Then the browser calls the SetColor method on each of the controls.

Each control checks for an enabling parent unit (which exists), constructs a simple undo unit which represents the color change, and adds it to the undo manager through the IOleUndoManager::Add method. Since the undo manager has an open undo unit (constructed by the property browser), it passes these undo units on to that parent unit through IOleParentUndoUnit::Add, which stores them inside itself.

After setting the color of the last control, the browser closes its enabling parent unit by calling Close on the undo manager. Since the undo manager has an open child, it passes the Close on to that undo unit. The undo unit does not have an open child, so it just returns S_FALSE; this tells the undo manager that the open unit itself should be closed. The undo manager closes the undo unit.

When the user pulls the Undo list down, the host application fetches the descriptions from the units in the undo manager, using IOleUndoManager::EnumUndoable. In this example, there is only one top-level undo unit, the parent undo unit constructed by the property browser; its description is "Change color of controls".

When the user selects that item from the Undo list, the host application instructs the undo manager to undo it, calling IOleUndoManager::UndoTo. The undo manager invokes the Do method on the parent undo unit; it in turn invokes the Do method on each of its child undo units, which restore the controls to their old colors.

Open-Ended Undoable Actions: Typing

The user types 'abcd' into an edit control on a form, moves the control, and then types in 'wxyz'. Three items appear in the Undo drop-down: 'typing "abcd"', 'move control', and 'typing "wxyz"'.

When the first character is typed, the edit control constructs an undo unit and adds it to the undo manager by calling Add. As more characters are typed, it adds them (using privately defined interfaces) to that undo unit.

When the user moves the control, the form constructs an undo unit describing the move and adds it to the undo manager by calling Add. At this point, the undo manager calls OnNextAdd on the undo unit created by the control; that undo unit notifies the edit control (again, through privately defined interfaces) that a non-typing action has occurred, and that the edit control should stop feeding characters to this undo

unit. If this were a parent unit which was accepting open-ended data like this, it would terminate the connection at the time it returned S_FALSE from its Close() method and merely delegate any call to OnNextAdd to its most recently added unit.

When the next character is typed, the edit control constructs a new undo unit and adds it to the undo manager by calling Add. Again, as more characters are typed, it adds them to this new undo unit.

The undo manager now has three top-level undo units: 'typing "abcd"', 'move control', and 'typing "wxyz"'.

Event Handler Walk-Through

1. The user types text into a TextBox, which has no event handlers written for it. However, the TextBox is bound to a database column.

The TextBox creates an undo unit for the change, and adds it to the stack. A Change event is fired, but no event handlers catch it. The TextBox also fires a property change notification.

When the user undoes this action, the old text is restored, a Change event is fired, and a new property change notification is fired. The data binding code catches the property change notification and marks the control as dirty.

2. The user clicks on a PushButton. The Click handler for the button clears the text in a TextBox on the button's form.

The PushButton fires a Click event, but does not create an enabling parent, since there is no way to undo a click. That is, this event is not symmetric. When the Text property of the TextBox is changed, the TextBox will attempt to create an undo unit and fail, since there is no enabling parent unit. The TextBox reacts to this by flushing the undo stack.

3. The user clicks on a PushButton. The Click handler increments a global variable, and sets the text in a TextBox on the same form to be the value of the global variable.

Same as above; the net effect is that the undo stack is flushed.

4. The user clicks a PushButton. The event handler clears a TextBox on another form, with a different undo stack.

Since no objects on the same form as the PushButton are affected, the undo stack of the PushButton form is unaffected. However, when the Text property of the TextBox is changed, there is no enabling parent open on its form, and as a result the undo stack of the TextBox's form is flushed.

5. The user types text into a TextBox. The Change event handler enables or disables a PushButton based on whether the text forms a legal identifier.

The TextBox fires a Change event while it has a blocking parent unit open. When the PushButton's Enabled property is changed, it examines the undo stack to see if a special parent unit is present. Since a blocked unit is open, the PushButton changes its Enabled property without logging an undo unit.

When the user undoes the typing, the TextBox will fire another Change event. The user's event handler will be called, and will update the state of the PushButton appropriately.

6. The user types text into a non-compliant TextBox built with the OLE Controls 95 CDK.

This has no effect on the form's undo stack. If this creates a dangerous situation, then the user should change the AllowUndo property of the form to False. Alternatively, the form could have disabled the undo manager at the time the TextBox was inserted since the TextBox did not have the OLEMISC_SUPPORTSMULTILEVELUNDO flag set. Either method will cause the Form to disallow any access to the undo stack manager, effectively disabling undo support for all controls on the Form.

7. The user clicks a PushButton on a DataFrame detail. The event handler causes the current record to be committed.

The DataFrame flushes the undo stack whenever it commits a record, since the user can't back up past the commit.

8. The user edits the Caption property of several PushButtons using the property browser.

The property browser creates and opens an enabling parent undo unit, then applies the changes to the Caption property to all selected controls. When the Caption property of each PushButton is changed, the PushButton examines the undo stack, and finds that there is an open and unblocked parent undo unit. As a result, the PushButton goes ahead and creates an undo unit for the property change and adds it to the stack. When the property browser is through applying changes, it closes its open parent.

When the user undoes the change, the parent unit calls the PushButtons' undo units, which restores the Caption property to its original value and sends a property change notification. The property browser catches this notification, and updates the property browser (presuming it still is viewing the PushButtons).

9. The user runs a wizard which resizes all the controls on the form to reasonable heights and widths.

Before the wizard code starts manipulating objects, it opens an enabling parent undo unit. When the X Object for each control receives a Move request, it examines the undo stack, finds an open unblocked parent unit, and creates an undo unit for the Move.

When the user undoes the wizard's action, all the X Object undo units are fired, restoring the original positions of all the controls.

Control Sizing

Main objective: Allow controls to provide sizing hints as the user resizes the control

Abstract: A new interfaces is defined through which controls can specify minimum and maximum sizes, and can specify the nearest “good” size to a size specified by the user.

Overview

There are two general approaches to sizing a control: the first approach gives the control responsibility for sizing itself; the second approach gives the container responsibility for sizing the control. The first approach is called autosizing; the second approach consists of two alternatives: content and integral sizing.

Autosizing

Autosizing typically occurs with controls such as the Label control which resizes if the autosize property was enabled *and* the associated text changed. Autosizing is handled differently depending on the state of the control. If the control is inactive, the following occurs:

1. The control calls IOleClientSite::RequestNewObjectLayout
2. The container calls IOleObject::GetExtent and retrieves the new extents
3. The container calls IOleObject::SetExtent and adjusts the new extents

If the control is active, the following occurs:

1. The control calls IOleInPlaceSite::OnPosRectChange to specify that it requires resizing
2. The container calls IOleInPlaceObject::SetObjectRects and specifies the new size

Content and Integral Sizing

In content sizing, the container passes a structure to the control into which the control returns a suggested size. In integral sizing, the container passes a preferred size to the control and the control modifies the requested height. Integral sizing is used when the user rubberbands a new size in design mode.

IViewObjectEx interface

In addition to supporting direct manipulation, this interface supports enhanced, flicker-free drawing for irregular and transparent objects. It also support non-rectangular hit-testing, custom grab handles, and control sizing.

```
typedef struct tagExtentInfo
{
        UINT cb;        //struct size
        DWORD dwExtentMode;         //
        SIZEL sizelProposed;
}DVEXTENTINFO;

typedef enum tagExtentMode
```

{

DVEXTENT_CONTENT; //ask control how big it wants to be to exactly fit content (snap-to-size)

DVEXTENT_INTEGRAL; // while resizing, pass proposed size to control

}DVEXTENTMODE;

```
interface IViewObjectEx : public IViewObject2
{
    // IViewObject methods
                ...

    // IViewObject2 methods
                ...

    // IViewObjectEx methods
    ...

    HRESULT GetNaturalExtent(DWORD dwAspect, LONG lindex, DVTARGETDEVICE* ptd,
                        HDC hicTargetDev, DVEXTENTINFO* pExtentInfo, LPSIZEL* psizel);
};
```

IViewObjectEx::GetNaturalExtent

The *IViewObjectEx::GetNaturalExtent* function supports two types of control sizing: content and integral. For content sizing, the control simply returns the suggested size; for integral sizing, the control actually adjusts its height.

Argument list follows:

Argument	Type	Description	
dwAspect	DWORD	Specifies how the object is to be represented. It can be one of the following values:	
		DVASPECT_CONTENT	Provide a representation of the control so it can be displayed as an embedded object inside of a container. This value is typically specified for compound document objects. The presentation can be provided for the screen or printer.
		DVASPECT_DOCPRINT	Provide a representation of the control on the screen as though it were printed to a printer using the Print command from the File menu. The described data may represent a sequence of pages..
		DVASPECT_ICON	Provide an iconic representation of the control.
		DVASPECT_THUMBNAIL	Provide a thumbnail

		representation of an object so it can be displayed in a browsing tool. The thumbnail is approximately a 120 by 120 pixel, 16-color (recommended) device-independent bitmap potentially wrapped in a metafile.
lindex	LONG	Indicates the portion of the object that is of interest for the draw operation. Its interpretation varies depending on the value in the *dwAspect* parameter. See the **DVASPECT** enumeration for more information.
ptd	DVTARGETDEVICE*	Points to the target device structure that describes the device for which the object is to be rendered. If NULL, the view should be rendered for the default target device (typically the display). A value other than NULL is interpreted in conjunction with *hicTargetDev* and *hdcDraw*. For example, if *hdcDraw* specifies a printer as the device context, the *ptd* parameter points to a structure describing that printer device. The data may actually be printed if *hicTargetDev* is a valid value or it may be displayed in print preview mode if *hicTargetDev* is NULL.
hicTargetDev	HDC	Specifies the information context for the target device indicated by the *ptd* parameter from which the object can extract device metrics and test the device's capabilities. If *ptd* is NULL; the object should ignore the value in the *hicTargetDev* parameter.
pExtentInfo	DVEXTENTINFO*	Points to structure that specifies sizing data.
psizel	LPSIZEL*	Points to sizing data returned by control. The returned sizing data is set to -1 for any dimension that was not adjusted. That is if cx is -1 then the width was not adjusted, if cy is -1 then the height was not adjusted. If E_FAIL is returned indicating no size was adjusted then psizel may be NULL.
Return Value	HRESULT	S_OK successfully returned (or adjusted) the size; E_FAIL not implemented for given mode, or, size was not adjusted; E_NOTIMPL not implemented.

Event Coordinate Translation

Abstract: The coordinates used by controls when firing events have changed from himetric to points, in order to be consistent with methods and properties.

The 96 specification for controls requires that coordinates passed for events fired by the control change from being Hi-metric to being Points based. This change brings the event passing of coordinates in line with properties and methods and thus coordinate translation is no longer the responsibility of the container. This raise certain compatibility issues where a control fires events using a coordinate base that it is not expecting, this should only be an issue where a 96 control container is hosting an older pre-96 control as follows:

- When an older pre-96 container hosts a 96 control the control will present the event coordinates as points, this should not cause the container any problems as the container should recognize the parameter type.
- When a 96 container hosts a pre-96 control the control will present the container with coordinates and expect the container to any translation necessary. However the 96 container will be expecting a control to conform to the 96 controls specification and present its coordinates as points. The control uses the TranslateCoordinates method on the IOleControlSite interface provided by the container in the same way as it does for properties and methods to achieve this.

As a result the user of a 96 container hosting pre-96 controls will need to be aware that further translation of coordinates may be necessary when events are fired.

Textual Persistence of Controls

Abstract: Although developed separately and prior to the 96 controls specification the textual persistence of controls has not been documented previously so it has been included as part of this specification.

Overview

The set of interfaces is of slightly more general purpose than solely text persistence. It is technically "tagged property persistence". The OLE Control throws a bunch of properties into a "PropertyBag" at save time. It is handed back this bag at load time to pull its properties out of. The most important difference between this set of interfaces and the current Property Set scheme and any other Property Set interfaces that may be developed is that *these are persistence interfaces.* It is specifically designed to be an *efficient* way of transferring individual properties to and from a container-supplied persistent storage. Its purpose is not to be confused with those of Property Sets or the emerging IProperty[Set]Storage interfaces, whose goal is to provide a standard storage format for properties. The existing OLE control text persistence scheme is ample evidence that we shouldn't attempt to force one scheme to be used for the other's purpose.

Interfaces

```
interface IPersistPropertyBag : public IPersist
{
    HRESULT InitNew();
    HRESULT Save([in]LPPROPERTYBAG pPropBag, [in]BOOL fClearDirty, [in]BOOL
fSaveAllProperties);
    HRESULT Load([in]LPPROPERTYBAG pPropBag, [in]LPERRORLOG pErrLog);
};

interface IPropertyBag : public IUnknown
{
    HRESULT Write([in]LPCOLESTR lpstrName, [in]VARIANT* pvarValue);
    HRESULT Read([in]LPCOLESTR lpstrName, [in/out]VARIANT* pvarValue, [in]LPERRORLOG
pErrLog);
};

interface IErrorLog : public IUnknown
{
    HRESULT AddError([in]LPCOLESTR lpstrName, [in]LPEXCEPINFO pexcepinfo);
};
```

IPersistPropertyBag

IPersistPropertyBag::InitNew

HRESULT InitNew()

Initialises the storage. Called when the control is initialised.

Argument	Type	Description
Return Value	HRESULT	S_OK - The new storage object was successfully initialized. CO_E_ALREADYINITIALIZED -The component object has already been initialized. E_OUTOFMEMORY - The storage object was not initialized due to a lack of memory. E_FAIL - The storage object was not initialized due to some reason besides a lack of memory.

IPersistPropertyBag::Save

HRESULT Save([in]LPPROPERTYBAG pPropBag, [in]BOOL fClearDirty,

[in]BOOL fSaveAllProperties)

Called by the container to save the objects properties. The container passes in a pointer to an IPropertyBag interface that the control uses to write its properties.

Argument	Type	Description
pPropBag	IPropertyBag*	A pointer to an IPropertyBag interface that the control uses to write its properties.
fClearDirty	BOOL	Instructs the control to clear its dirty flag.
fSaveAllProperties	BOOL	Instructs the object whether to save all persistent properties (TRUE), or just the ones that have changed from their defaults (FALSE).
Return Value	HRESULT	S_OK - Successful. STG_E_MEDIUMFULL - The object was not saved because of a lack of space on the disk. E_FAIL - The object could not be saved due to errors other than a lack of disk space.

IPersistPropertyBag::Load

HRESULT Load([in]LPPROPERTYBAG pPropBag, [in]LPERRORLOG pErrLog)

Called by the container to load the control's properties.

Argument	Type	Description
pPropBag	IPropertyBag*	A pointer to an IPropertyBag interface that the control uses to read its properties.
pErrLog	IErrorLog*	A pointer to an IErrorLog interface that the control may use to log any errors that occur whilst loading the properties.
Return Value	HRESULT	S_OK - Successful. E_OUTOFMEMORY - The properties were not loaded due to a lack of memory. E_FAIL - The properties were not loaded due to some reason besides a lack of memory

IPropertyBag

IPropertyBag::Write

HRESULT Write([in]LPCOLESTR lpstrName, [in]VARIANT* pvarValue)

Called by the control to write each property in turn to the storage provided by the container.

Argument	Type	Description
lpstrName	LPCOLESTR	A pointer to a string holding the name of the property.
pvarValue	VARIANT*	A pointer to a Variant containing the value.
Return Value	HRESULT	S_OK - Successful. STG_E_MEDIUMFULL - The property was not saved because of a lack of space on the disk. E_FAIL - The property could not be saved due to errors other than a lack of disk space.

IPropertyBag::Read

HRESULT Read([in]LPCOLESTR lpstrName, [in/out]VARIANT* pvarValue,
[in]LPERRORLOG pErrLog)

Called by the control to read each property in turn from the storage provided by the container.

Argument	Type	Description
lpstrName	LPCOLESTR	A pointer to a string holding the name of the property.
pvarValue	VARIANT*	A pointer to a Variant containing the value.
PErrorLog	IErrorLog*	A Pointer to an IErrorLog interface used to log errors during the read process.
Return Value	HRESULT	S_OK - Successful. E_OUTOFMEMORY - The property was not loaded due to a lack of memory. E_FAIL - The property was not loaded due to some reason besides a lack of memory

IErrorLog

IErrorLog::AddError

HRESULT AddError([in]LPCOLESTR lpstrName, [in]LPEXCEPINFO pexcepinfo)

Called to log any errors that occur during the property load process.

	Type	Description
lpstrName	LPCOLESTR	A pointer to a string holding the name of the property whose read resulted in the error.
pexcepinfo	EXCEPINFO*	A pointer to an EXCEPINFO structure containing details of the error encountered.
Return Value	HRESULT	S_OK - Successful. E_FAIL - The error was not successfully added to the error log.

Interface Usage:

The following usage notes are in the context of text persistence, since this would likely be the most common usage of the interfaces. Keep in mind that this is not the *only* conceivable usage of these interfaces.

When a control is first instantiated, InitNew is called. Note that if a container has multiple save options (such as *Save as Text* and *Save as Stream*) the Save may be called on a different interface than that on which InitNew was called. This should not present a problem for the control. For consistency this interface will also have an InitNew method.

To save a control in text, the container would call IPPB::Save, passing in an IPropertyBag. The control would call IPB::Write for each persistent property, giving the name and value. In addition to the IPropertyBag, the Save method takes two BOOL parameters. fClearDirty instructs the object to clear its Dirty bit. fSaveAllProperties instructs the object whether to save all persistent properties (TRUE), or just the ones that have changed from their defaults (FALSE).

To load a control from text, the container would parse the text file and call IPPB::Load. An optional IErrorLog can be passed to IPPB::Load. Once the control's IPPB::Load is called, the control would call

IPB::Read for each property, passing in a Name and pointer to the destination Variant. If the vt field of the Variant is VT_EMPTY, the container would simply fill it in with the value in whatever type is easiest for it. If the vt field had a specific type, the container would coerce the value to the requested type. If it couldn't coerce to that type, it could either return an error or fill in the variant with a valid type and leave it to the caller to check if the return type equaled the requested type.

Errors can be reported on a per-property basis, assuming an IErrorLog was passed into IPPB::Load. The control can call IEL::AddError if the IPB::Read fails or returns an unsatisfactory value. The excepinfo argument in the final interface may not actually be OLE Automation's EXCEPINFO structure, it is just used here to give an idea of the type of information to be passed to the AddError method.

There is no explicit support for transferring BLOB properties (large chunks of binary data). It was decided that the interface would be much cleaner and simpler if these types of properties were persisted by way of an object-valued property, which would support IPersistStream, and whose persistent image would be the blob in question. The ultimate goal is to eliminate any copying of blobs.

If a control had an object property, it would Write a variant of type VT_UNKNOWN, passing in the object's IUnknown pointer as the value. The container could then QI for the supported Persistence interfaces and save accordingly. This has the effect of giving the topmost container the power to decide how to save each object, including their subobjects (down until the first level that didn't support this interface). This is desirable for text persistence, since the goal is to have a consistent look about the final text file. If there was a case where an object did not want its subobject saved in text (for whatever reason), even though it supported IPersistPropertyBag, it could call the subobject's IPersistStream::Save (or any other persistence interface) itself, supplying its own implementation of IStorage/IStream/IPropertyBag, then handing the result to the container in the form of a blob property. Hence each object still has ultimate control over its subobjects (as it should), but the mechanism is there to allow the top-level container to keep a consistent text file format.

To read an object property, the control would Read a variant of type VT_UNKNOWN. If the value of the variant's punkVal was NULL, the IPB implementation would create the object corresponding to the property name (hence it must save the CLSID when it Writes the property) and set the punkVal to the resulting IUnknown. If the punkVal already had a value in it, the IPB implementation would simply QI for the appropriate persistence interface and call Load on the given object. In the case of reading object-type properties, the IErrorLog that was passed to IPPB::Load should be passed on to IPB::Read. This is the only case where this parameter is needed.

Much like IPersistStream behavior, the IPropertyBag pointer passed into IPPB::Load/Save will only be valid for the duration of that call. The control cannot hold onto it to "scribble" to.

Standard DISPIDS

Abstract: A number of standard dispids have been defined for the 96 controls specification.

DISPID_MOUSEPOINTER

#define DISPID_MOUSEPOINTER -521
Property of type integer.
The Mousepointer property identifies standard mouse icons

Value	Description
0	(Default) Shape determined by the object.
1	Arrow
2	Cross (cross-hair pointer)
3	I Beam
4	Icon (small square within a square)
5	Size (four-pointed arrow pointing north, south, east, and west)
6	Size NE SW (double arrow pointing northeast and southwest)
7	Size N S (double arrow pointing north and south)
8	Size NW, SE
9	Size E W (double arrow pointing east and west)
10	Up Arrow
11	Hourglass (wait)
12	No Drop
13	Arrow and hourglass
14	Arrow and question mark
15	Size all
99	Custom icon specified by the MouseIcon property

DISPID_MOUSEICON

#define DISPID_MOUSEICON -522
Property of type Picture.

DISPID_PICTURE

#define DISPID_PICTURE -523
Property of type picture.

DISPID_VALID

#define DISPID_VALID -524 // Is data in control valid?
Property of type BOOL.
Used to determine if the control has valid data or not.

DISPID_ AMBIENT_PALETTE

#define DISPID_AMBIENT_PALETTE -726 // Container's HPAL
Used to allow the control to get the container’s HPAL. If the container supplies an ambient palette then that is the only palette that may be realized into the foreground. Controls that wish to realize their own palettes must do so in the background. If their is no ambient palette provided by the container then the active control may realize its palette in the foreground. Palette handling is further discussed in “Palette Behaviour for OLE Controls” which is shipped as part of the Microsoft Internet SDK.

Databinding

A new databinding attribute has been added to allow properties distinguish between communicating changes only when focus leaves the control or during all property change notifications.

The new attribute known as ImmediateBind is to allow controls to differentiate two different types of bindable properties. One type of bindable property needs to notify every change to the database, for example with a checkbox control where every change needs to be sent through to the underlying database even though the control has not lost the focus. However controls such as a listbox only wish to have the change of a property notified to the database when the control loses focus, as the user may have changed the highlighted selection with the arrow keys before finding the desired setting, to have the change notification sent to the database every time that the user hit the arrow key would be give unacceptable performance. The new immediate bind property allows individual bindable properties on a form to have this behavior specified, when this bit is set all changes will be notified.

The new ImmediateBind bit maps through to the new VARFLAG_FIMMEDIATEBIND (0x80) and the FUNCFLAG_FIMMEDIATEBIND (0x80) bits in the VARFLAGS and FUNCFLAGS enumerations for the ITypeInfo interface allowing for the properties attributes to be inspected.

Property Categorization

Overview

A new interface is provided to allow containers to categorize a control's properties. A control's properties can be categorized using Category IDs allowing a container to place properties together in a property browser in groups of similar properties. This interface is supported by a control and is optional but recommended as many containers will in future make use of this interface.

Using this interface a container can ask a control which category a property belongs to, and also ask for the text name of any control specific category. A number of standard categories are defined, but a control may also have its own control specific categories. Standard categories are distinguished by having negative category IDs, whilst control specific categories can have positive IDs.

The following standard categories are defined:

```
#define PROPCAT_Nil           -1
#define PROPCAT_Misc          -2
#define PROPCAT_Font          -3
#define PROPCAT_Position      -4
#define PROPCAT_Appearance    -5
#define PROPCAT_Behavior      -6
#define PROPCAT_Data          -7
#define PROPCAT_List          -8
#define PROPCAT_Text          -9
#define PROPCAT_Scale         -10
#define PROPCAT_DDE           -11
```

ICategorizeProperties Interface

```
typedef int PROPCAT;

interface ICategorizeProperties : public IUnknown
{
    HRESULT MapPropertyToCategory( [in] DISPID dispid, [out] PROPCAT* ppropcat );
    HRESULT GetCategoryName( [in] PROPCAT propcat, [in] LCID lcid, [out] BSTR* pbstrName);
};
```

ICategorizePriorities::MapPropertiesToCategory

HRESULT MapPropertyToCategory([in] DISPID dispid, [out] PROPCAT* ppropcat);
This method returns the category ID for the property whose dispid is passed in.

Argument	Type	Description
dispid	DISPID	The dispid of the property.
ppropcat	PROPCAT*	A pointer to the property category ID returned.
Return Value	HRESULT	S_OK - Successful.
		E_FAIL - The dispid supplied was invalid.

ICategorizePriorities::GetCategoryName

HRESULT GetCategoryName([in] PROPCAT propcat, [in] LCID lcid, [out] BSTR* pbstrName);
This method returns the text name of the category requested. This method need not be supported unless the control has any specific categories of its own, and should not be called for any of the standard category ids of which the container should already be aware.

Argument	Type	Description
propcat	PROPCAT	The ID of the property category requested.
lcid	LCID	The locale ID.
Return Value	HRESULT	S_OK - Successful.
		E_NOTIML - The method is not implemented
		E_FAIL - The propcat supplied was invalid.