 © Microsoft Corporation 1992-1993. All Rights Reserved.

2. User Model and User Interface Guidelines

This purpose of this section is to specify and illustrate the user model and the user interface implementation guidelines for OLE 2. The goals of the OLE 2 user interface are:

1. Express a unified model of compound document composition and interaction that allows users to efficiently accomplish their tasks.
2. Fully exploit the integration power of OLE 2 using current user interface frameworks and mechanisms.
3. Establish a sound user model of compound documents that meets current application needs and that will gracefully lead to more *data-centric* systems in the future.

Simply put the central user model for OLE 2 compound documents is:

> *A user creates and manipulates various types of information (objects) which reside in a containing document. As a user focuses on a particular object, its corresponding commands and tools become available allowing the user to interact with it directly from within the document (in-place activation). Objects may be transferred within and across documents and still retain their full-featured editing and operating capabilities (embedding). In addition, information may be connected so that changes in one object may automatically be reflected in another (linking).*

A crucial difference between the OLE 1 and OLE 2 user interface is the ability to perform *in-place activation*: whereas OLE 1 required an object to be edited back in its original application window, OLE 2 enables the user to activate and manipulate an object directly from within the document. OLE 2 makes this possible by presenting an object's commands and tools right within the container document's window, and as a result the conceptual model of containment is no longer compromised by removing objects from the document for editing. Although in-place activation is usually the most desirable way to manipulate objects, there are certain cases when the user may still wish to edit in a separate window, for particular viewing reasons or to view an object back in the original context from which it has been taken. Therefore OLE 2 objects will likely set some kind of in-place activation as their primary action (or *verb*), but may also provide the option to open separately, as OLE 1 objects must do.

In general, the concepts put forth by this user interface specification are a superset of the OLE 1 principles, as they intend to unify the style of interaction with objects from both releases. The distinction between OLE 1 and OLE 2 objects is made evident to the user only as some objects by default interact in-place while others open in a separate window; in all other respects they behave virtually the same.

This section begins with a compound document example, illustrating embedded and linked data and in-place activation. The illustrations used are taken directly from the OLE 2 User Interface mock-up which is a companion to this specification. The subsequent sections describe the specifics of how objects are created, edited, and transferred within documents. As expected, most attention is focused on the user interface integration issues between the object and container applications for in-place activation.

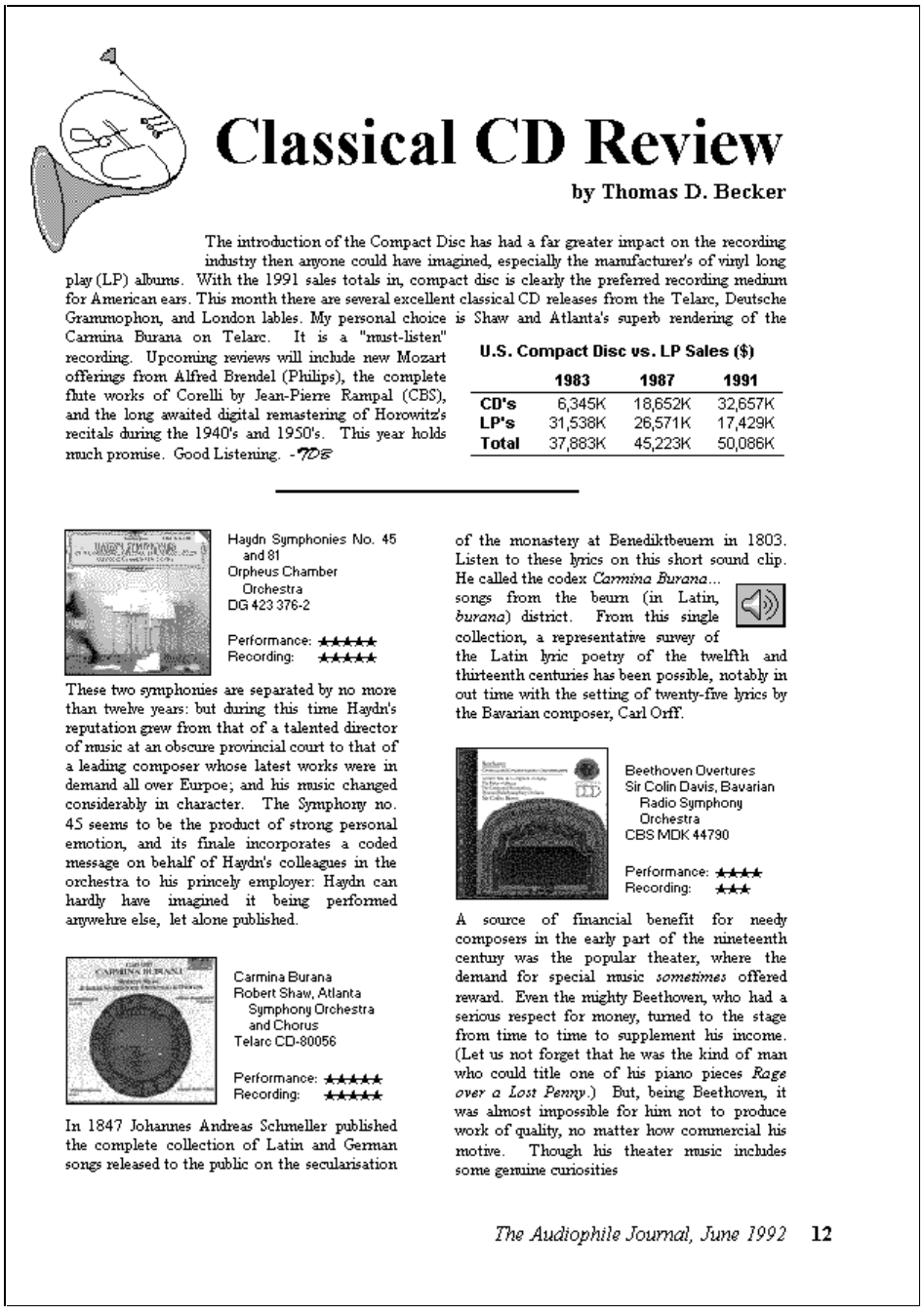
It is physically possible for OLE objects and containers to implement any user interface that is supported by the OLE 2 protocol, but because of the extremely integrated nature of OLE 2, it is more important than ever for them to subscribe to the same basic user model; otherwise, the users will not benefit from the strengths of OLE.

2.1. An In-place activation Example

Below is a fictitious music review from an audio journal containing a spreadsheet, sound recording, and pictures which have originated from various applications. The document itself, *CLASSCD.DOC*, was created using Microsoft Word but the spreadsheet, sound recording, and pictures came from Microsoft Excel, Microsoft Sound Recorder, and Microsoft Paintbrush respectively. Let us suppose that the author of this article has *embedded* the compact disc cover images, the sound recording, and the spreadsheet; that is, the data of these objects actually reside within the document *CLASSCD.DOC*. The French horn image,

 © Microsoft Corporation 1992-1993. All Rights Reserved.

though, is borrowed from another file, *HORNS.BMP*, and is merely *linked* into *CLASSCD.DOC*; the horn's data still resides within *HORNS.BMP*, but its image appears in *CLASSCD.DOC* and it will reflect any changes made to the original.



Classical CD Review

by Thomas D. Becker

The introduction of the Compact Disc has had a far greater impact on the recording industry then anyone could have imagined, especially the manufacturer's of vinyl long play (LP) albums. With the 1991 sales totals in, compact disc is clearly the preferred recording medium for American ears. This month there are several excellent classical CD releases from the Telarc, Deutsche Grammophon, and London lables. My personal choice is Shaw and Atlanta's superb rendering of the Carmina Burana on Telarc. It is a "must-listen" recording. Upcoming reviews will include new Mozart offerings from Alfred Brendel (Philips), the complete flute works of Corelli by Jean-Pierre Rampal (CBS), and the long awaited digital remastering of Horowitz's recitals during the 1940's and 1950's. This year holds much promise. Good Listening. -TDB

U.S. Compact Disc vs. LP Sales ($)

	1983	1987	1991
CD's	6,345K	18,652K	32,657K
LP's	31,538K	26,571K	17,429K
Total	37,883K	45,223K	50,086K

Haydn Symphonies No. 45 and 81
Orpheus Chamber Orchestra
DG 423 376-2

Performance: ★★★★★
Recording: ★★★★★

These two symphonies are separated by no more than twelve years: but during this time Haydn's reputation grew from that of a talented director of music at an obscure provincial court to that of a leading composer whose latest works were in demand all over Eurpoe; and his music changed considerably in character. The Symphony no. 45 seems to be the product of strong personal emotion, and its finale incorporates a coded message on behalf of Haydn's colleagues in the orchestra to his princely employer: Haydn can hardly have imagined it being performed anywehre else, let alone published.

Carmina Burana
Robert Shaw, Atlanta Symphony Orchestra and Chorus
Telarc CD-80056

Performance: ★★★★★
Recording: ★★★★★

In 1847 Johannes Andreas Schmeller published the complete collection of Latin and German songs released to the public on the secularisation of the monastery at Benediktbeuern in 1803. Listen to these lyrics on this short sound clip. He called the codex *Carmina Burana*... songs from the beurn (in Latin, *burana*) district. From this single collection, a representative survey of the Latin lyric poetry of the twelfth and thirteenth centuries has been possible, notably in out time with the setting of twenty-five lyrics by the Bavarian composer, Carl Orff.

Beethoven Overtures
Sir Colin Davis, Bavarian Radio Symphony Orchestra
CBS MDK 44790

Performance: ★★★★
Recording: ★★★

A source of financial benefit for needy composers in the early part of the nineteenth century was the popular theater, where the demand for special music *sometimes* offered reward. Even the mighty Beethoven, who had a serious respect for money, turned to the stage from time to time to supplement his income. (Let us not forget that he was the kind of man who could title one of his piano pieces *Rage over a Lost Penny*.) But, being Beethoven, it was almost impossible for him not to produce work of quality, no matter how commercial his motive. Though his theater music includes some genuine curiosities

The Audiophile Journal, June 1992 **12**

Figure 1. Article with linked and embedded objects.

How was this music review actually created? First Microsoft Word was opened and the text was entered. Next the various objects were either fetched from other files using the Insert Object command or were brought in via the clipboard and the Paste Special command (both are explained in later sections). Below is the document loaded into Microsoft Word.

 © Microsoft Corporation 1992-1993. All Rights Reserved.

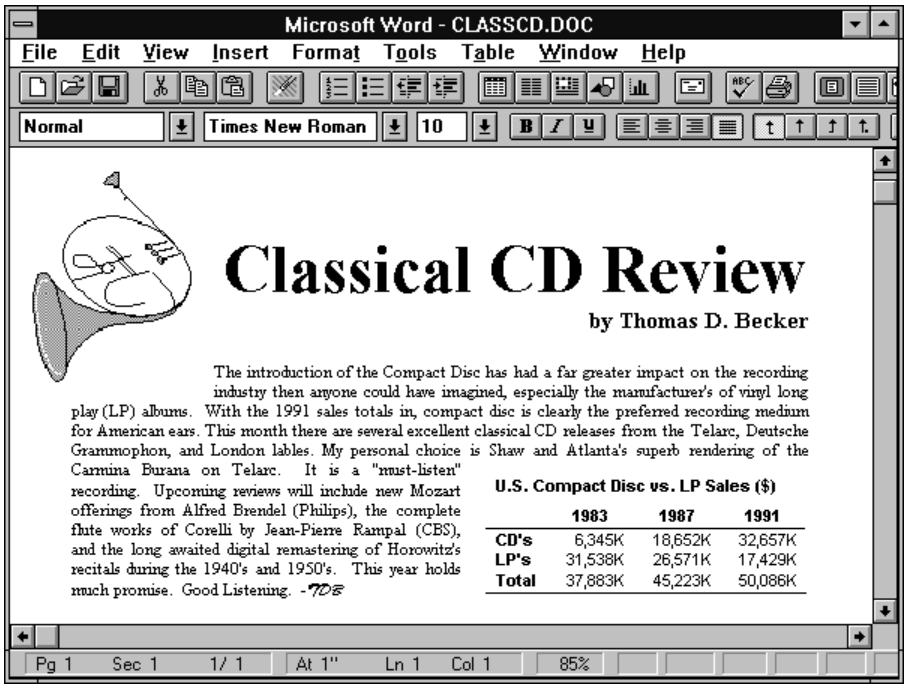


Figure 2. OLE 2 compound document within Microsoft Word.

The objects within the Microsoft Word document are just like any other picture with regards to selection, and how it responds to atomic level commands like positioning, resizing, and justification. Unlike the native text, however, the objects are *opaque*, that is, their internal content is not exposed to the containing Microsoft Word application. Figure 3 shows the linked horn as part of a multiple selection.

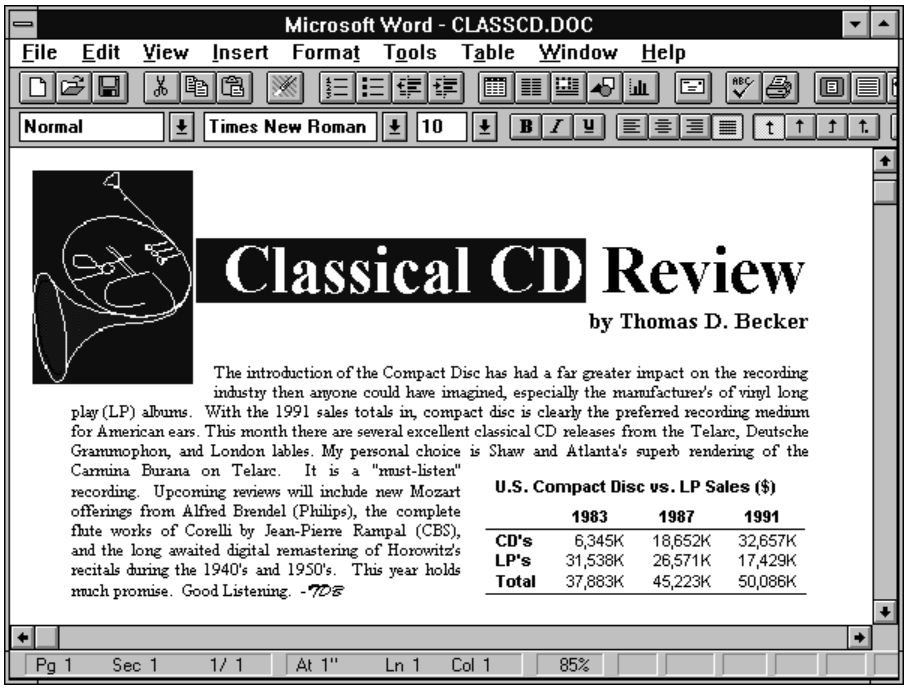


Figure 3. An object as part of a multiple selection

A single mouse click on the embedded Microsoft Excel worksheet selects it by itself and resize handles appear since that resizing is meaningful within Microsoft Word (see Figure 4). In other scenarios where the container may not support resizing, these handles would not appear. Now that the object is selected,

 © Microsoft Corporation 1992-1993. All Rights Reserved.

the worksheet's OLE registered verbs may now be accessed through the Edit / Worksheet submenu. Notice that the status line also indicates it may be edited by double-clicking the mouse on it.

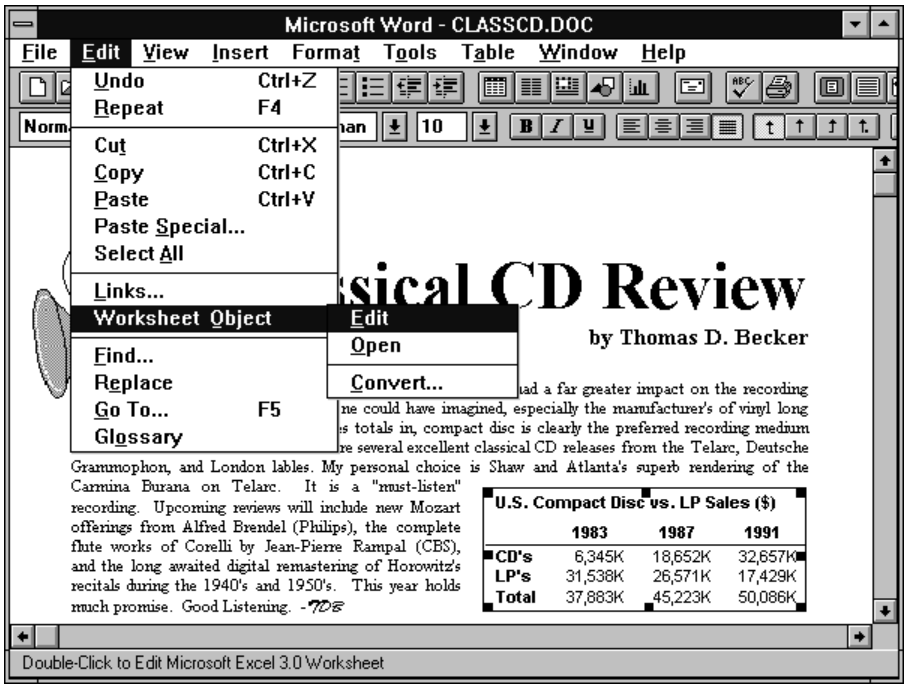


Figure 4. Singly selected embedded Microsoft Excel worksheet.

With OLE 1, the user double-clicked the embedded object to launch its application in another window, where the object may be edited. Double-clicking an OLE 2 object, on the other hand, usually activates the object and allows the user to interact with it *in-place*. Figure 5 shows how the Microsoft Excel user interface appears when the worksheet is activated. The current window takes on the user interface of the object's supporting application. The window title, editing menus, control bars, status lines, and potentially even frame adornments (like Microsoft Excel's row/column buttons) appear enabling the user to edit the

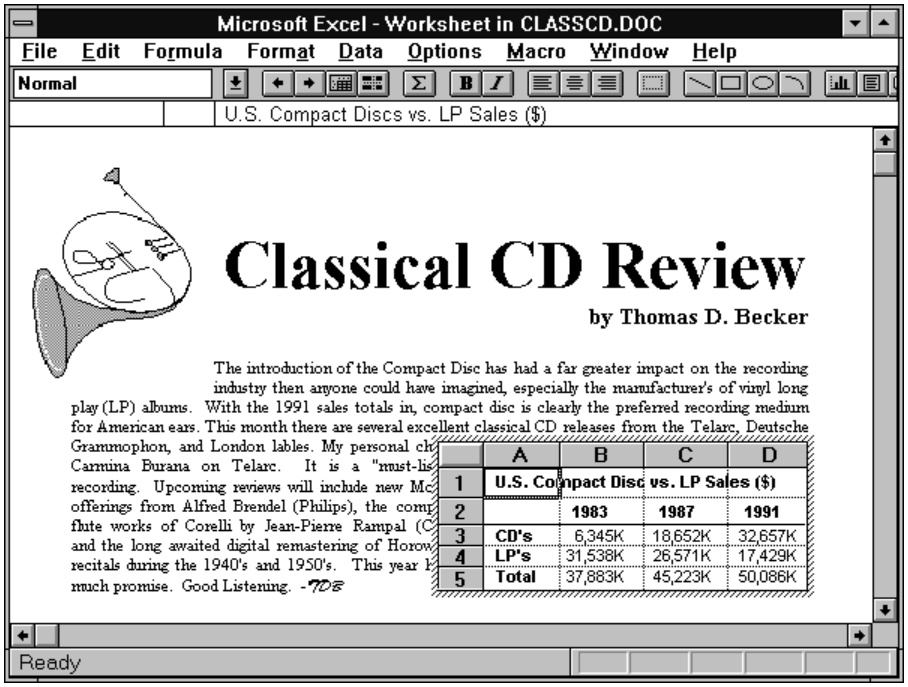


Figure 5. In-place activated Microsoft Excel Worksheet.

 © Microsoft Corporation 1992-1993. All Rights Reserved.

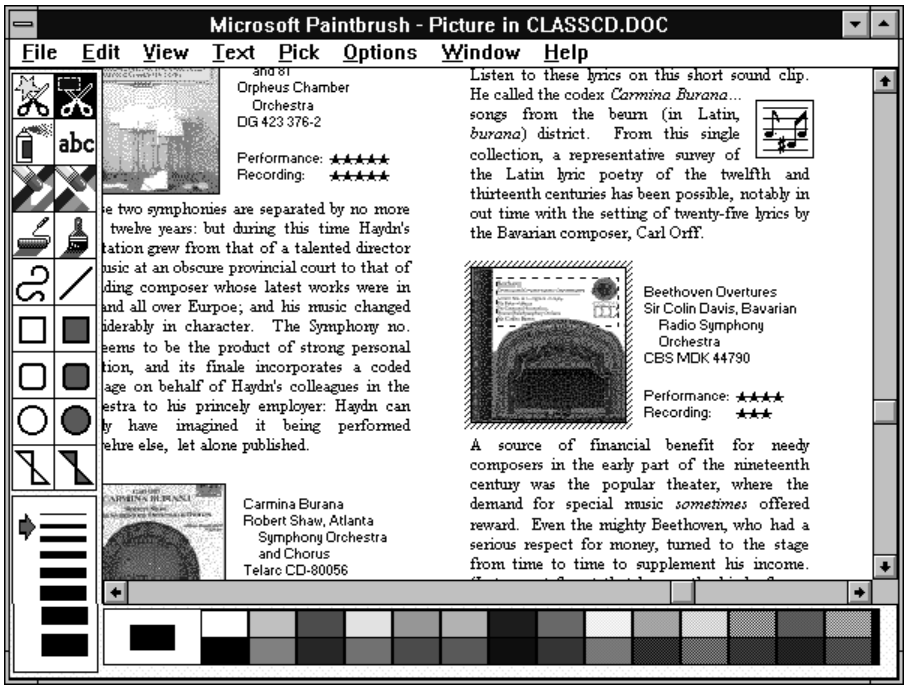


Figure 6. In-place activated Microsoft Paintbrush Picture.

object *just as if it were in its original application.* Document and workspace level interfaces such as the File and Window menus remain under the control of the top level container (Microsoft Word) and are still accessible during in-place activation. A black hatch border appears around the activated object to suggest the extent of the in-place activation context. The active object's menus and tools apply within this bordered area. Also notice since Microsoft Excel's control bars take less space than Microsoft Word's, more of the Microsoft Word text has been exposed.

The CD cover images may be edited in the same fashion: double-clicking one of them results in Microsoft Paintbrush coming to the fore, allowing the user to make full use of the graphic editing tools within the active image. Getting back to Microsoft Word is done simply by clicking anywhere back in the document outside of the active object.

Since OLE 2 supports any levels of nested embeddings and links, the user may "tunnel" through several objects. Figure 7 is another document *ANNUAL.DOC* containing an embedded Microsoft Excel Worksheet which has an embedded Microsoft Graph of its own. Single-clicking the graph merely selects it as an object within Excel.

Issuing the edit verb (or double-clicking) activates the graph in-place, bringing up the graph's application; this is illustrated in Figure 8. Notice that at any given time, only the interface for the immediately active object and the top level container (in this case Microsoft Word) is presented; intervening parent objects do not remain active.

Interacting with OLE 2 compound documents may be characterized as the user working with information, whether it is an OLE object or not, under the normal selection and editing rules of its immediate container application. To interact with the internals of an embedded object, the user simply double-clicks on it and the interface of its originating application is blended into the current window. The user interface integration model divides control between the application that handles the document as a whole and the one that edits the active object. This blending of Microsoft Excel and Microsoft Word interfaces that was illustrated in Figure 5 is annotated in Figures 9 and 10. Microsoft Excel takes over most of the menu bar (including the Help menu), the control bar, the cursor, the area immediately around the worksheet, and the status line. The container application Microsoft Word retains control over the File and Window menus, the document scrollbar, and any MDI controls.

 © Microsoft Corporation 1992-1993. All Rights Reserved.

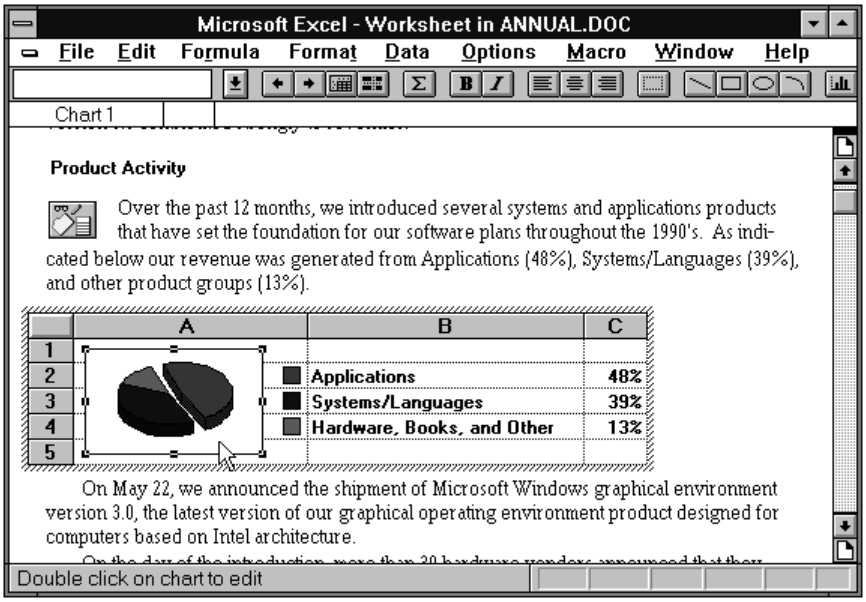


Figure 7. In-place activated Microsoft Excel worksheet.

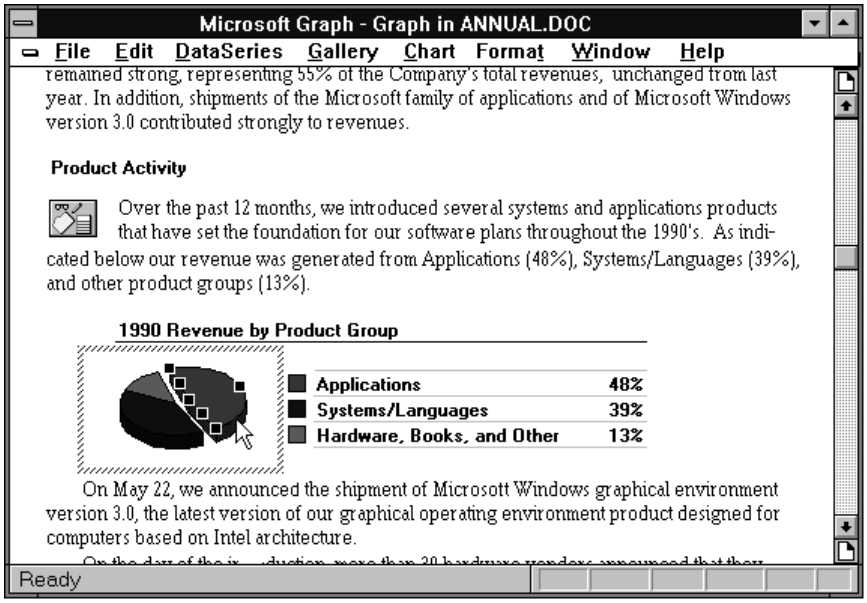


Figure 8. In-place activated graph.

 © Microsoft Corporation 1992-1993. All Rights Reserved.

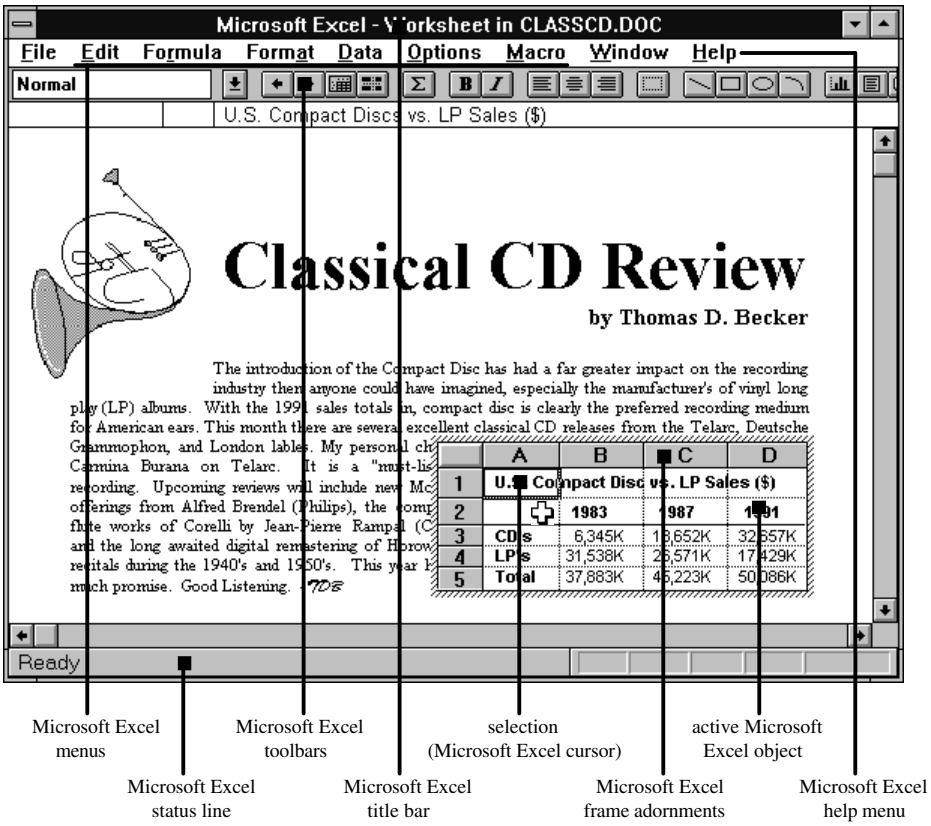


Figure 9. Object interface components.

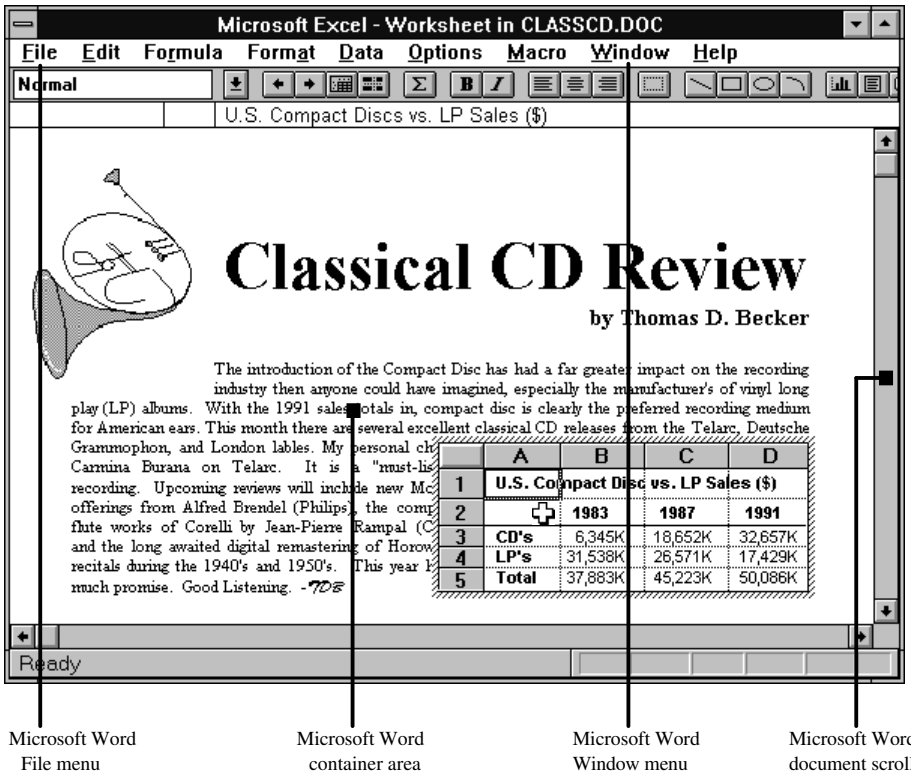


Figure 10. Container interface components.

 © Microsoft Corporation 1992-1993. All Rights Reserved.

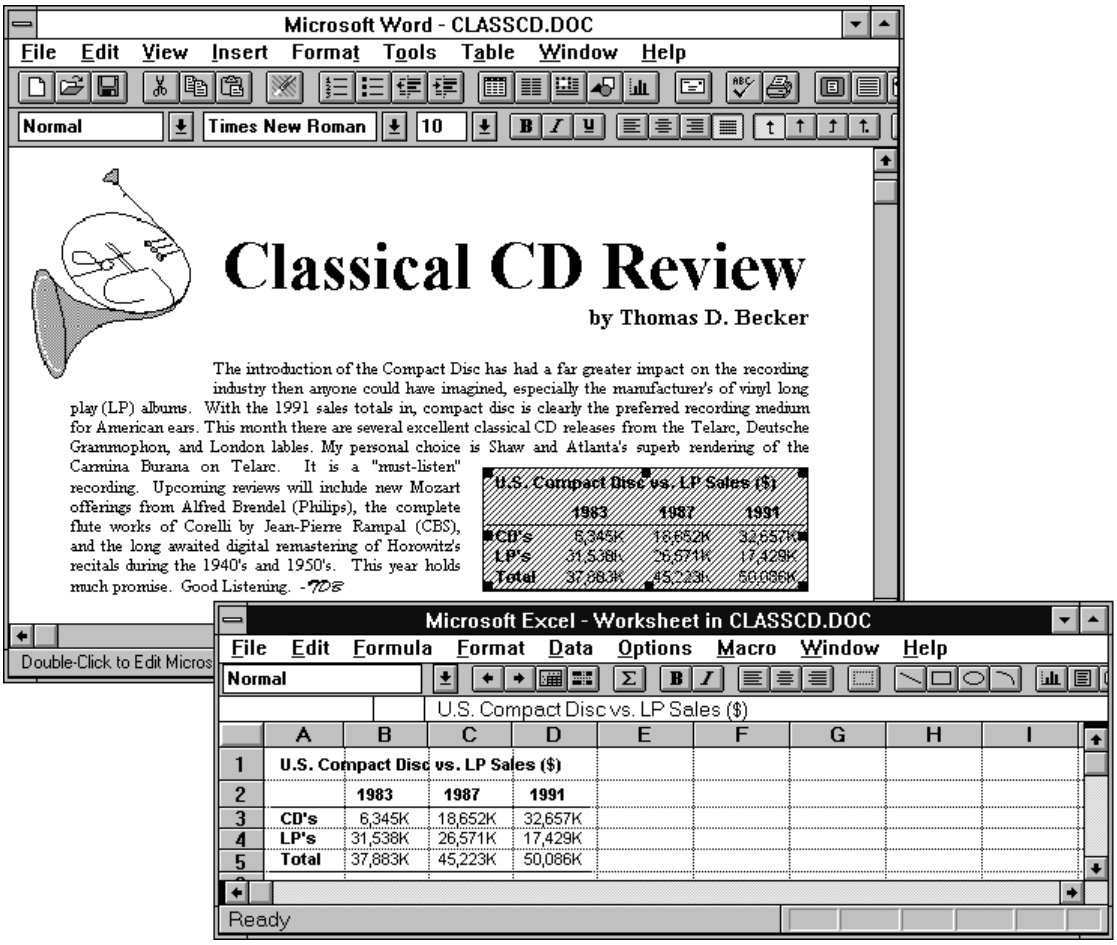


Figure 11. An opened Microsoft Excel Worksheet.

The figures thus far have shown the embedded worksheet when it is activated in-place; the OLE 1 "open" style of editing is still available under OLE 2 as the "open" verb and brings up a separate Microsoft Excel window as pictured below. Notice when an object is open, its appearance in the container document is masked with the same black hatch pattern as an indication that the object is open in another window, and the object's window title mentions "Worksheet in CLASSCD.DOC" to associate it with the container document.

Unlike in OLE 1 however, it is no longer necessary to explicitly "update" or save changes back to the container document. The updates happen automatically presenting the illusion that the user is still working directly on the object still in *CLASSCD.DOC* albeit in a separate viewer. The object in the container and the object in the open viewer are to be thought of as the same object. Note that although an object may be opened, it is still a functioning member of its container document, still capable of being selected and participating in container application commands.

These examples demonstrate editing an embedded object. Editing a *linked* object is very similar with one important exception: unlike an embedded object whose data actually resides in the compound document, the data for a linked object remains back in the document from which it originated, its *source* document. The "link" speaks of the association between the real data in the source document to its superficial appearance in any number of other *destination* documents. So unlike the embedded table and CD images which truly live and may be edited within *CLASSCD.DOC*, the linked horn always points the user back to its real presence in *HORNS.BMP* for editing. The figure below shows how the source document *HORNS.BMP* appears for the user to edit the horn. It is important to note that changes made to the middle horn will not only be reflected back in *CLASSCD.DOC* but in every other document which has linked to

the same portion of the *HORNS.BMP* document. This illustrates both the power and the potential danger of using links in documents.

At first glance editing a linked object appears to be the same as "open" editing an embedded object. Note however that the horn does not have the "open" hatch pattern nor does the second window title bar indicate that the object is within *CLASSCD.DOC*; it is in a completely independent document *HORNS.BMP*. Additionally a link loading dialog (below is shown) as the source document is being fetched.

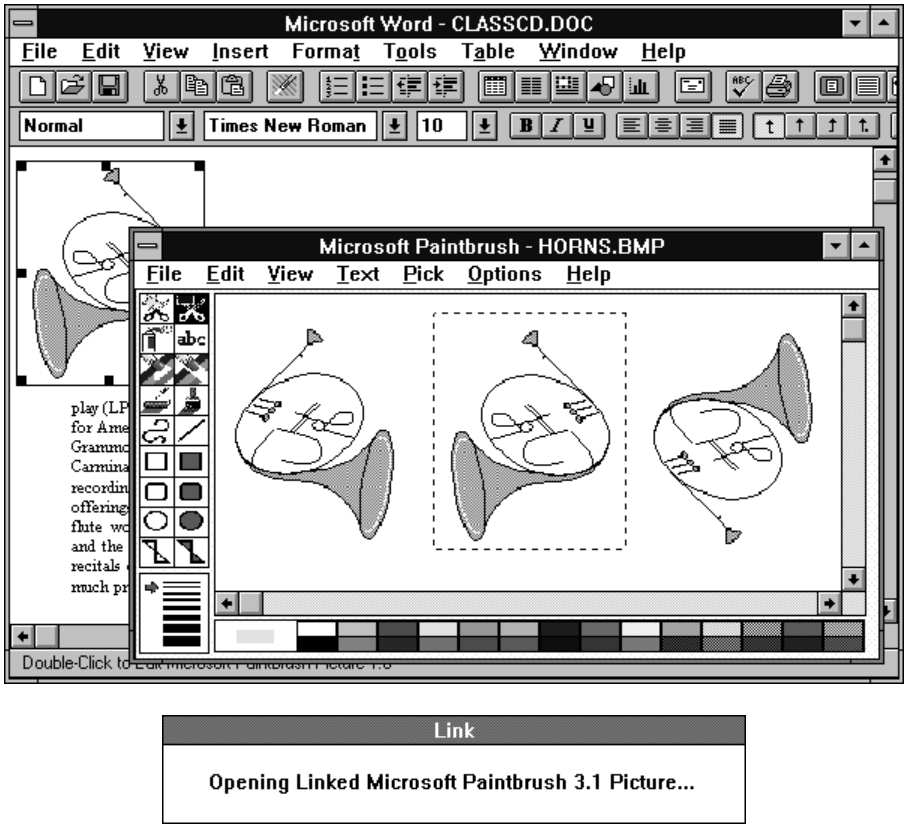


Figure 12. Editing a link source.

This example has briefly stepped through the composition and modification of an OLE 2 compound document, and hopefully has conveyed the flavor of in-place activation. The sections that follow will point out specific guidelines for the user interface of OLE 2 container and object applications.

2.2. Object Type

As a preface to this object-oriented user interface, the user's notion of what an *object* is and what an object's *type* is must first be discussed. An OLE object, as portrayed to the user, is a unit of information that resides in a document and whose behavior is constant no matter where it is located; its behavior is defined by the object itself rather than being determined by the document that holds it. The user may interact with an object as a whole (via OLE verbs and container commands), or with its contents by engaging its proper tools (its application). Objects may be nested arbitrarily within other objects, and objects may be moved, copied, or linked from one location to another.

An object's *type* (e.g., CorelDRAW! 3.0 Drawing) is the human readable form of its class[2] name and conveys to the user the object's *behavior* or *capability,* and is not necessarily meant to imply its storage format. For instance, two different types may coincidentally use the same storage format (e.g., .bmp, .rtf),

[2] The registered class name is never to be surfaced to user, it is for internal registration and programming only.

but their types are distinct since they are likely handled by different applications and thus exhibit different behavior. An object's type is expressed as a string (maximum of 40 characters) in one of the three recommended forms below:

1. <*brandname*> <*application*> [<*version*>] <*data type*>

   (e.g. Microsoft Excel 4.0 Worksheet)

2. <*brandname-application*> [<*version*>] <*data type*>

   For cases when the brandname and application are one in the same (e.g. CorelDRAW! 3.0 Drawing, WordPerfect Text) .

3. <*brandname*> <*application*> [<*version*>]

   When the application sufficiently describes the data type (e.g. Microsoft Graph 3.0).

The four components of the type name convey useful clues to the user:

1. <*brandname*>

   Communicates product identity and differentiation.

2. <*application*>

   Indicates which application is responsible for activating the object.

3. <*data type*>

   Suggests the basic nature or category of the object (i.e., drawing, spreadsheet, sound), and should be a maximum of 15 characters.

4. <*version*>

   When there are multiple versions of the same basic type, a version number is necessary to distinguish types for purposes of upgrading.

These object type names provide users with a precise language for referring to objects. Since object type names appear throughout the OLE interface in dialogs and menu commands, users will become very conscious of an object's type and its associated behavior. The full type name (also referred to as <*full type name*>) is to be used in dialogs, most messages, and status lines.

Because of their length, however, only their <*data type*> should be used in menus (both pull-down and pop-up), title bars, and in the list pane of the Links dialog. The data type is considered to the *short type name* of the full type name (referred to as <*short type name*>. If a short type name is not available for an object (because it is an OLE 1 object or the string was simply not registered), then full type name should be used instead. For example, a Microsoft Excel 4.0 Worksheet is simply referred to as a "Worksheet" in menus, title bars (Worksheet in CLASSCD.DOC), and in the listpane of the Links dialog. All dialogs that display the full type name must allocate enough space for 40 characters in width. By comparison dialogs must accommodate 15 characters when using the short type name.

2.3. Class Installation, Version Control, Emulation, and Conversion

The model for installing, converting, and emulating object types is in many ways analogous to how applications and files accomplish these tasks today. OLE 2 accommodates:

1. Class-wide and per-object version upgrading.
2. Per-object type conversion.
3. Activation of an object with alternative applications (type emulation).

Let's examine how this accomplished by first looking at the installation process illustrated by the diagram in Figure 13. When an object class is registered (during an object application installation), it should first detect if a previous version of the same class already exists in the registration database. If one does, the user is given the option to upgrade all of the older version objects to the new version automatically the next time each is activated. (Of course if a previous version is not detected, the new version would simply be registered without prompting.) When appropriate, an upgrade dialog should mention any important

 © Microsoft Corporation 1992-1993. All Rights Reserved.

differences between the versions which might help the user decide whether or not to upgrade. If (s)he elects to upgrade all of the older objects, the old version is replaced by the new one in the registration so that from that point on, those previous version objects will activate with the new application and be saved in the new version's format (class-wide version upgrading).

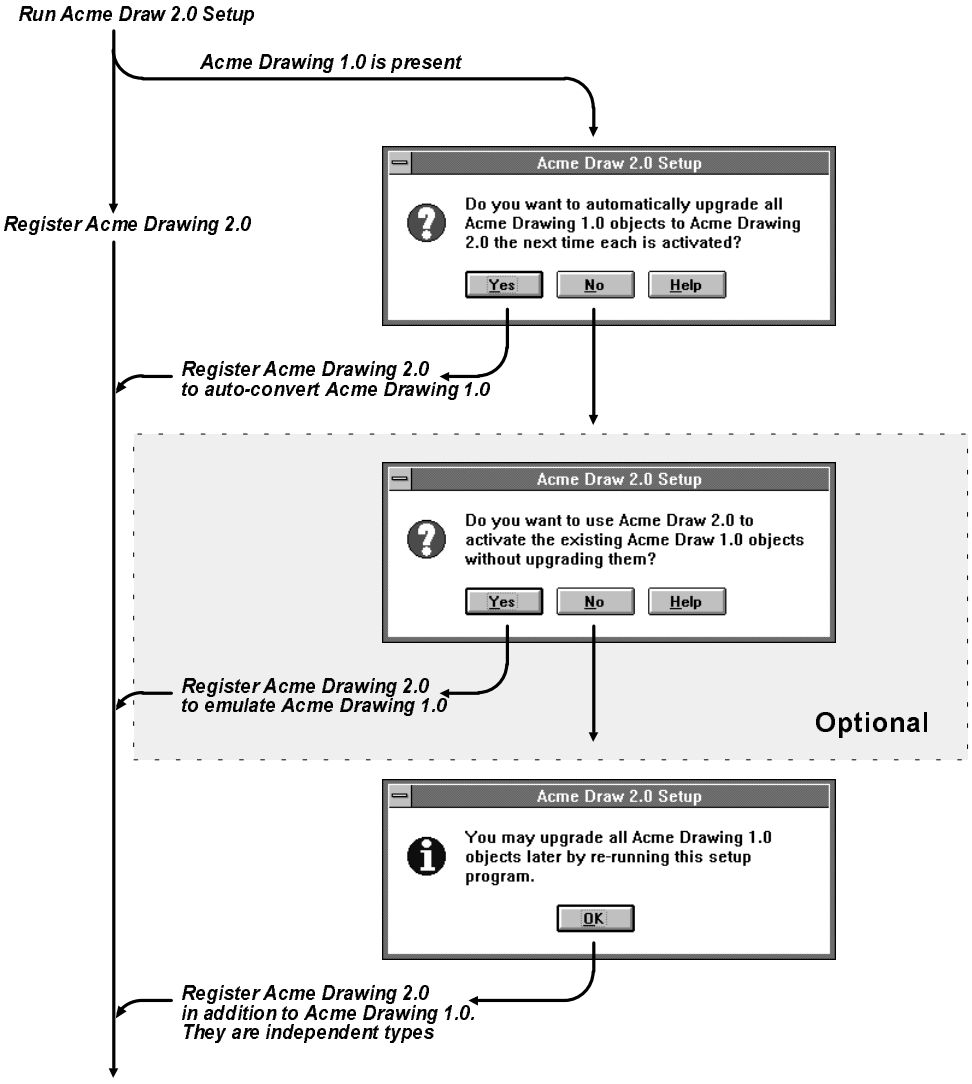


Figure 13. Installing a new version of a object application.

Optionally, the new application may offer to emulate the older version objects without actually converting them to the new format. This allows the user to use the desirable features of the new application while maintaining the object in its old format as long as possible. However, if the user makes use of a particular feature of the new application which cannot be accommodated by the old type's format, the user should be prompted with the option of abandoning such edits in order to stay within the old format or to keep the edits and accept the necessary conversion to the new format. If the user specifies no upgrading or emulating, the new application is registered *in addition* to the old one, and the two types are considered completely independent. The user may re-run the object application installation program any time later to re-specify version upgrading or emulation intentions.

Figure 13 illustrates the decision tree and dialogs for installation of an object application (Acme Drawing 2 is the new application being installed in the following example).

As part of the class registration, the setup program should register a list of formats that the installing application is capable of reading and a list of formats it is capable of both reading and writing. By matching these lists to those registered by other types, it can be determined which other types the new

application is capable of converting and emulating. Conversion and emulation of types is surfaced to the user through the "Convert" dialog explained below. Although an object will normally be activated by the application that matches its type, there are three noted exceptions.

1. *Activating an object whose version has been upgraded (auto-convert).* If the user activates an object whose version has been reassigned to a newer version application, the object is launched with the new application and is saved back under the new version.

2. *Conversion.* If the user would like to permanently convert an object from one type to another, (s)he selects the particular object and chooses the "Convert..." from the Edit / Object submenu. This will yield the "Convert" dialog, which when the "Convert to:" radio button is chosen displays a list of other types to which the object may be converted. This radio button is chosen by default if the selected object's type is registered in the database. (This list is composed of all types which have registered themselves as capable of reading the selected object's format, and *do not* necessarily guarantee that a reverse conversion is possible). The list also contains the selected object's current type. This enables the user to change whether to Display As Icon without necessarily converting the selection's type. If a new type is chosen from list and the dialog is committed, the selected object is immediately and silently converted to the new type. (If the object were opened, the container closes it before performing conversion.) If on the other hand, the user entered the Convert dialog by way of trying to edit an unregistered object, the new type application should immediately activate after the Convert dialog and the actual conversion happens as part of loading the object.

3. *Type Emulation*. Since users will be exchanging OLE documents between different machines, it will be common for them to receive an object for which they do not have the matching application necessary to activate it. In addition, users may wish from time to time to use non-default applications on their own objects in order to exploit unique editing features of a particular application for instance. Both of these scenarios are supported by the same "Convert" dialog. By choosing the second radio button captioned "Activate as:" (chosen by default if the object's type is not registered), the subset of types in the list pane which are capable of emulating the selected object are displayed; the remaining types which are only capable of conversion do not appear. The type which the object is activating the object is selected by default in the list. Note that the object's real type should always be present in the list so the user may choose to activate it normally again. By choosing a type and confirming the dialog, every object of the selected object's type will henceforth be activated as an object of the alternate type; specifically, they will take on the alternate type's verbs in the Edit/Object submenu and be activated using the alternate application, but they will still keep their original type name throughout the entire user interface as they continue to be stored in their original type's format. Since the alternate application does not actually convert the objects, they may continued to be exchanged among users in their original type. (Beneath the covers, this dialog registers the alternate application for the selected object's type in the database, so as a result the emulation only applies to a particular machine; each machine will activate the type with the particular application registered in its own database.) If the container has other already loaded instances of the type which is to be emulated, it is recommended that the container reload each under the emulating type the next time the user encounters them. This will ensure the illusion that all instances are immediately emulated according to the Activate As.... command.

   An error dialog will appear when a user attempts to activate (double-click) an unregistered object which provides immediate access to the "Convert" dialog. Additionally, the user may explicitly access the dialog through the Edit / Object > Convert... command (which incidentally is the only command available for an unregistered object since its verbs are unknown).

Since the "Convert" dialog may be invoked at any time, the user may re-specify an alternate type whenever (s)he wishes. Even if there are no conversions or alternate types available for the selected

 © Microsoft Corporation 1992-1993. All Rights Reserved.

object's type, the "Convert" dialog should still be available as a means of simply respecifying the icon of the object. Notice how the "Result" help text elaborates on the outcome of the conversion and emulation options. Below is a table outlining the result text that should be used for the Convert dialog.

When an *embedded* object is selected...

Function	Result Text
Convert to Original Type	The selected <original type> will not be converted.
Convert to Original Type + Display as Icon	The selected <original type> will not be converted. It will be displayed as an icon.
Convert to Different Type	Permanently changes the selected <original type> to a <selected type>.
Convert to Different Type + Display as Icon	Permanently changes the selected <original type> to a <selected type>. It will be displayed as an icon.

When a *linked* object is selected...

Function	Result Text
Convert to Original Type (only)	The selected <original type> will not be converted.
Convert to Original Type (only) + Display as Icon	The selected <original type> will not be converted. It will be displayed as an icon.

When either a *linked* or *embedded* object is selected (Display As Icon - disabled) ...

Function	Result Text
Activate as Original Type	Every <original type> will be activated as a <selected type>.
Activate as Emulating Type	Every <original type> will be activated as a <selected type>, but it will not be converted.

Table 1. Result text for Convert dialog

Figure 14 shows how these three cases are handled on a system that has three registered applications, Acme Drawing 1 and 2 and Megasoft Drawing 1. Beginning at the point "Select an Acme Drawing 1", one would normally follow the straight down path of activation and de-activation. However, if the Acme Drawing 1 were replaced by Acme Drawing 2 or if the user wished to convert or emulate the Acme Drawing 1, the appropriate alternate routes are taken. The top right section of the figure shows the result of trying to activate an unregistered ProDraw Drawing 1, and the ensuing error and "Convert" dialogs. Note that in this example both Acme Drawing 2 and Megasoft Drawing 1 are possible conversions of the ProDraw Drawing 1, but only Megasoft Drawing 1 is a legal alternate type. Figure 15 shows the close-ups of the recommended dialogs used in Figure 14.

Take, for example, a user who specified to activate all Acme Drawing 1.0 objects as Acme Drawing 2.0. (S)he could have made this decision either while running the Acme Drawing 2.0 setup program or at any point afterwards using the Activate As option in the Convert dialog. If the user makes use of features unique to Acme Drawing 2.0 which cannot be accommodated in the Acme Drawing 1.0 format, the following warning message should be given at the moment the object is deactivated (for in-place) or closed (for open). Answering yes will perform the conversion and save the object with all of its changes intact; answering no will lose certain changes in order to preserve the object in the Acme Drawing 1.0 format. This dialog (figure 13) should not only be issued between different versions of the same type, but for competitor type emulations as well which may incur a loss when saving in the other format. This warning can be as specific to the nature of the data loss as the application is able and willing to supply.

© Microsoft Corporation 1992-1993. All Rights Reserved.

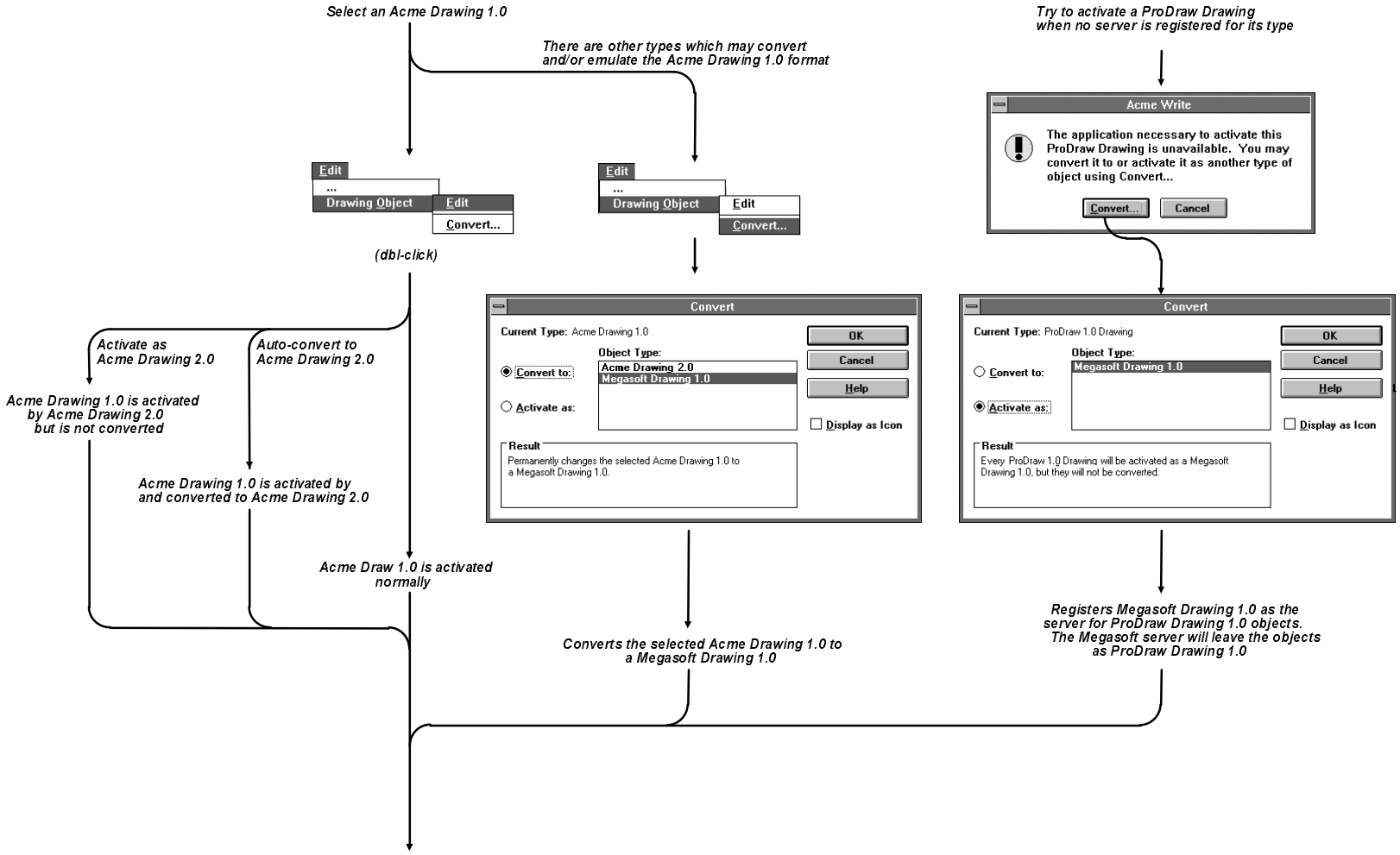


Figure 14. Upgrading, converting, and emulating object types.

 © Microsoft Corporation 1992-1993. All Rights Reserved.

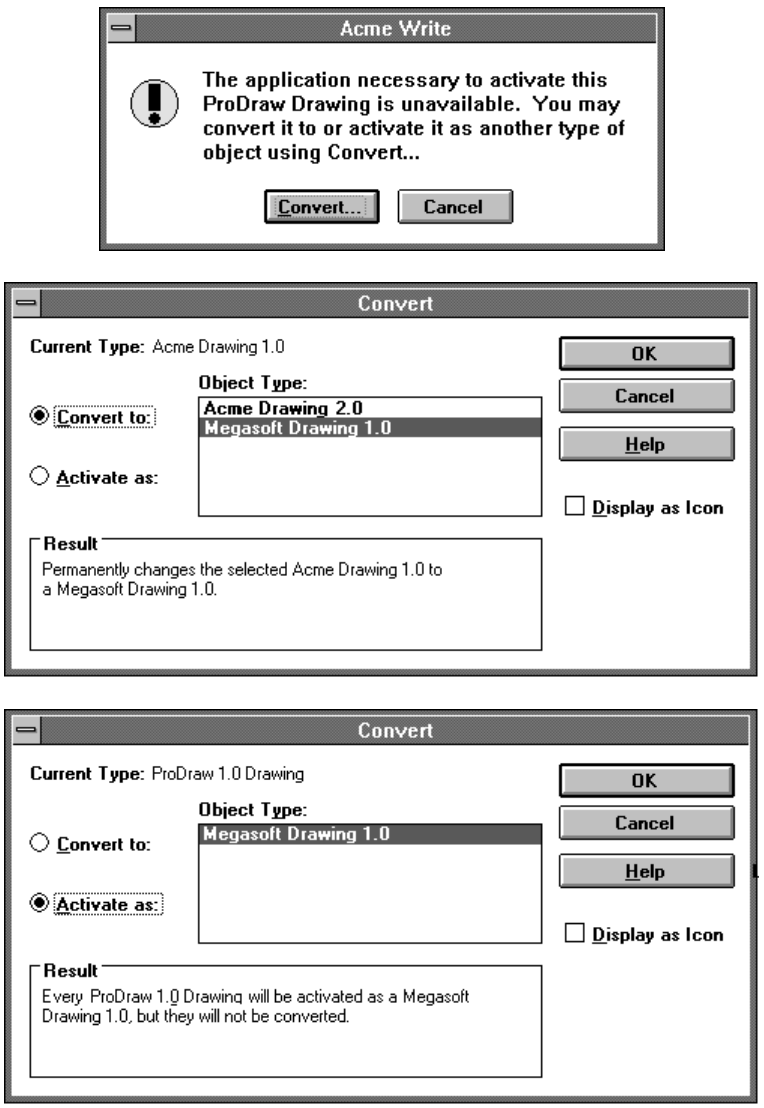


Figure 15. Close-up of conversion and emulation dialogs.

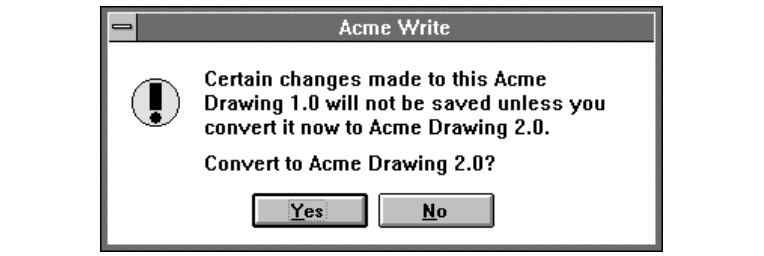


Figure 16. Warning about emulation data loss.

 © Microsoft Corporation 1992-1993. All Rights Reserved.

2.4. Creating Objects

Fundamental to creating compound documents is how users create individual objects. In addition to building objects from scratch, users may *specialize* or *convert* existing objects into new ones.

1. *Specialization*. The user makes a copy of an existing object which is similar (same type, similar properties) to the object (s)he wishes to create, and then fashions the copy into the desired new object. Users may organize their prototypical objects in "template" documents within the file system, and open them on the desktop to be used as palettes when creating new objects. In cases when an appropriate prototype does not exist (when an object type has just been installed for instance) the user may instantiate an application's default object through the "Insert Object" dialog (explained below), and then modify it to the desired result. Specializing prototypes is not a new OLE mechanism as much as it is a judicious copying from previous work, offering a highly extensible and stylized way of managing templates while exploiting users' familiarity with file storage hierarchy.
2. *Conversion*. In other cases the user may wish to create a new object from an object of a completely different type, as in creating a chart from a table. To do this the user selects an object and converts it to a new type using the "Convert" dialog described above.

The "Insert Object" dialog in Figure 17 allows the user to access new objects by object type or by inserting an existing file as an object. An "Insert Object..." command provides access to this dialog and should be placed within the menu responsible for instantiating or importing new objects into the document. If no such menu exists, use the Edit menu. As a general rule, commands which appear near the bottom of the menu are harder to discover for users, so if the container application wishes to emphasize its OLE abilities, it should locate the OLE dialog commands near the top of the appropriate menu. The user can embed a new object by choosing a type from the list pane and pressing "OK". Certain types of new objects may make use of the current selection in the container document and build themselves accordingly. A Microsoft Graph 3.0 may for instance fashion itself to reflect the data in a selected Word table. To ensure predictable insertion behavior, the following guideline should be followed: if a newly inserted object is *not* based on the current selection, the selection should be *replaced* by the new object; this is effectively a paste over. If the new object will remain *connected* to the selection (as in the Graph & Word table example), the new object should be inserted *in addition to* the selection and become the new selection. Lastly, if the object is based on but will not remain connected to the selection, it is up to the new object's application (that is, not the container application) whether or not to remove the given selection. In this last case, it is dependent on the specific use of an object type whether replace or add is more meaningful.

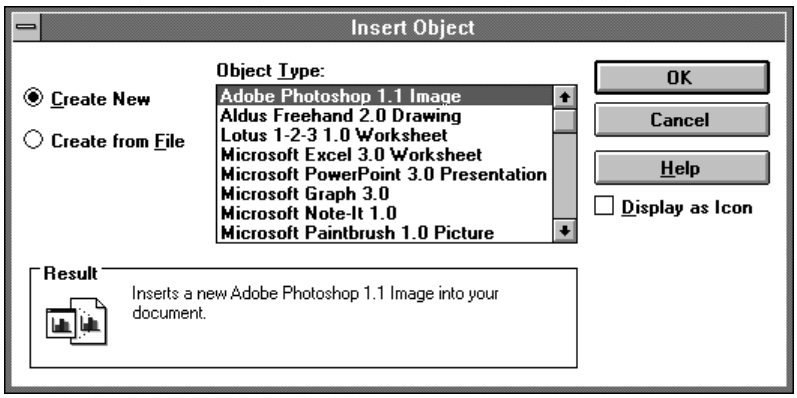


Figure 17. The Insert Object Dialog

 © Microsoft Corporation 1992-1993. All Rights Reserved.

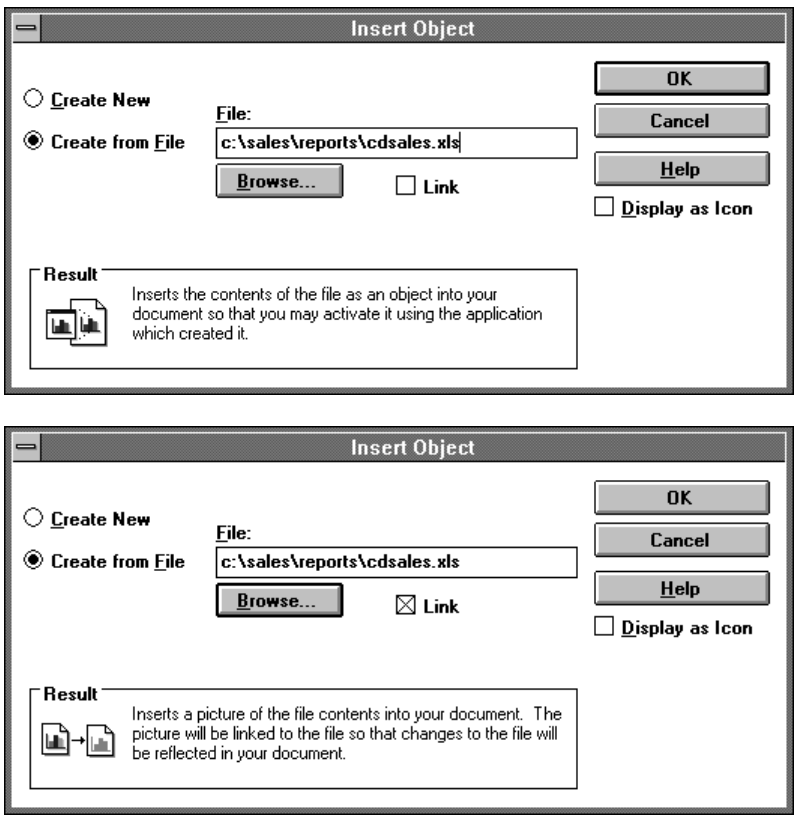


Figure 17. The Insert Object dialog, continued

The user may either embed or link an existing file as an object by pressing the "Create from File" and "Link File" buttons respectively. When a file button is chosen, the new type list disappears and an edit control and browse button appear in its place. The edit control initially displays the current directory (i.e. c:\current\) as the selection. The user may type over or add to the current directory path when specifying the file. The type list is completely unrelated to the file options since it is the file which ultimately specifies the type of the inserted or linked object. At the bottom of the Insert Object dialog, note the result text and a picture to describe the final outcome of the insertion. Below is a table which outlines the syntax of result text to use within the Insert Object dialog.

Function	Result Text
Create New	Inserts a new <object type name> into your document.
Create New as Icon	Inserts a new <object type name> into your document as an icon.
Create From File	Inserts the contents of the file as an object into your document so that you may activate it using the application which created it.
Create From File as Icon	Inserts the contents of the file as an object into your document so that you may activate it using the application which created it. It will be displayed as an icon.
Create From File + Link	Inserts a picture of the file contents into your document. The picture will be linked to the file so that changes to the file will be reflected in your document.
Create From File as Icon + Link	Inserts an icon into your document which represents the file. The icon will be linked to the file so that changes to the file will be reflected in your document.

Table 2. Result text for Insert Object

 © Microsoft Corporation 1992-1993. All Rights Reserved.

After an in-place object has been inserted, its application becomes active. If the insert object has an opened application, then an opened-style hatch rectangle appears in the container until an image from the application is available for an update.

Additionally, container applications may wish to present buttons in their control bar which insert objects directly. These buttons behave functionally the same as using the "Insert Object" dialog, but do it in a more efficient manner. Below are two buttons, one to create a new Microsoft Draw object and another to create a new Microsoft Graph object. The Microsoft Draw button inserts a default blank Microsoft Draw object, similarly to the "Insert Object" dialog; while the Microsoft Graph button creates a graph built from a currently selected table.

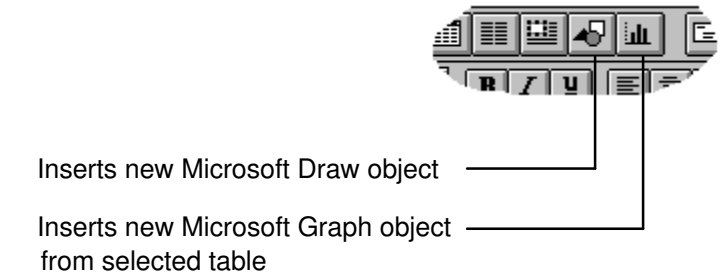


Figure 18. Control bar shortcut for creating new objects.

Whether the inserted object is new or previously exists, the user may specify to insert it as an icon in appearance by checking the "Display As Icon" option on the Insert Object dialog. In fact, if the user has chosen a non-OLE file for insertion, it may only be inserted as an icon, effectively "packaging" the file. Since a newly created object does not exist in another file, it may only be inserted as an *embedded* object. Only objects created from existing files may be inserted as *linked* objects. When this option is checked, the icon appears beneath the checkbox and the "Change Icon..." button is enabled. The "Change Icon..." button yields a dialog allowing the user to choose another icon and optionally give the icon a label. (A discussion of OLE 2 and the Packager is presented at the end of this chapter). The "Display as Icon" option on the Convert.. dialog allows the user to respecify at any time whether to have the object appear as an icon. In the Change Icon dialog the *current icon* is of course the icon that is presently assigned to the object (it in fact persists with the object as it moves to other machines); the *default icon* is the default icon for the object's type; if the object has no OLE type, then the blank "document" icon should be used. The *from file* option allows the user to override the default icon with one from an arbitrary file. The from file field is prefilled with the pathname of the current icon, and it is parsed and checked to be a legal path when it loses focus. Once the icon is chosen and the dialog is committed, it becomes the current icon.

The Convert dialog described earlier may be used to change the icon of an object at any time. When the Convert dialog is invoked for an object which already has an icon assigned, that icon is assumed to be a user chosen icon and as such the dialog will not automatically change it to the default icon of the new type after the conversion. If the user wishes to adopt the default icon of the new conversion type, then (s)he may enter the Change Icon dialog and choose the Default icon radio button. By doing so, the Convert dialog will dynamically display the appropriate icon of the selected type in the Convert type list.

There are also guidelines for determining the default label for the icon. If an item is *embedded* as an icon (whether it be by Paste Special, Insert Object, or drag/drop from the File Manager), its default label should be either of the following:

1. For an OLE object the label should be the short type name of its type name (i.e. Picture, Worksheet, ...).
2. For an OLE object which has not registered a short type name, it should use its full type name (i.e. Microsoft Paintbrush Picture, Microsoft Excel Worksheet)
3. For an item which is not associated with an OLE class the label is simply "Document".

 © Microsoft Corporation 1992-1993. All Rights Reserved.

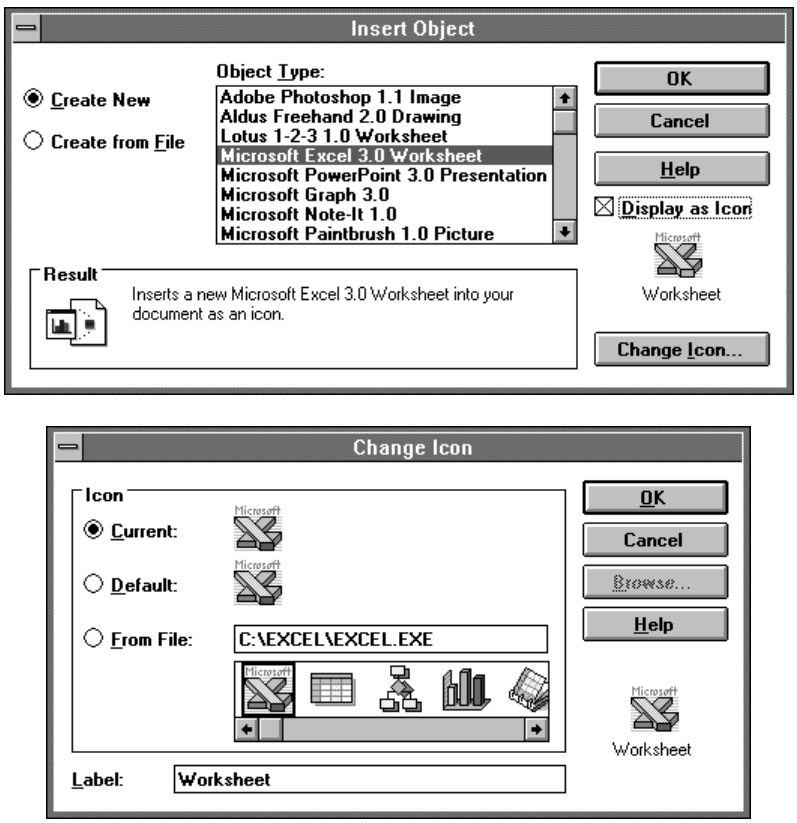


Figure 19. The Change Icon dialog

If a whole file is linked into a container as an icon (via Insert Object / Create from File + Link or by transfer from the File Manager), its default label is the source file's 8.3 name in lower case (i.e. source.xls). If a portion of a document is linked via the Paste Special dialog as an icon, its default label should be formed from the display name of the link source. In general, the display name of a link source may have a completely arbitrary syntax. Thus, the following algorithm for determining the default label is inherently heuristic. It is suggested that the default label be formed as follows.

1. From the end of the display name of the link source, scan for the last and second last occurrence of the following characters:

   \ / ! :

   The label should be formed from the display name starting at the penultimate scanned-for character through the end of the string.

2. If this scan in fact does not consume the whole display name, then the first line of the label should begin with "...". Following this, the portion of the display name between the penultimate and ultimate scanned-for characters (inclusive) should be displayed.

3. The second line of the label should contain the portion of the display name that follows the ultimate scanned-for character.

The occurrence of only one scanned-for character should be handled in the obvious way. Further, if the whole label fits on one line, then there is no point in breaking it into pieces. It is acceptable that applications in fact instead simply display the full display name of the link source, though clearly the above algorithm will produce results more pleasing to the user.

In general the label should be at most two lines, each line should be no greater than 10 capital W's in the rendered font in width, and the text of each line should be centered with respect to the icon image. The label of an icon persists with the object as it is transferred between containers and may only be changed afterwards by the user via the Change Icon dialog.

 © Microsoft Corporation 1992-1993. All Rights Reserved.

2.5. States and Visualizations of Objects

With editing in place, OLE 2 enhances the conceptual structure of a compound document by exposing in a single window the entire hierarchy of containment. In OLE 1 there were two kinds of objects to be considered: the container and the embedded objects it contained. That those objects themselves might also contain embedded objects was of no consequence, since this fact was not relevant until they were opened into their own windows. The container and the open object were always the same, and it was not necessary to think of embedded objects as containers themselves. In OLE 2 this situation changes: the open object becomes the root of a hierarchy of embedded objects, *each* of which may be a container in its own right:

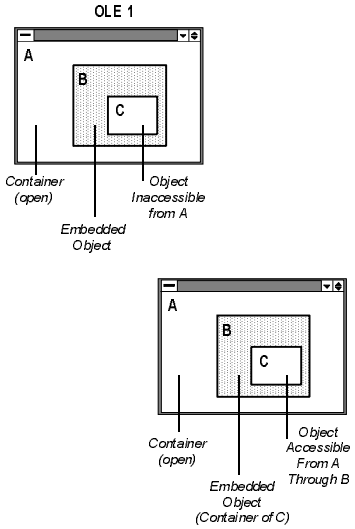


Figure 20. OLE 1 and OLE 2 object hierarchies.

As the user navigates through this hierarchy, OLE objects assume different states and appearances. An OLE object may be *inactive*, *selected*, *active*, or *open.*

2.5.1. Inactive

An object is said to be inactive when it is neither active nor part of a selection. It is displayed in its *presentation* form which is (usually) conveyed through its cached meta-file description. It may be desirable to know whether an object is embedded or linked at a glance without having to interact with it. Container applications should provide a "show objects" option that places a single pixel wide black *solid* border around the extent of an embedded object and a *dotted* border (Figure 22) around linked objects. If the container application cannot guarantee that a linked object is up to date with its source (because an automatic update was unsuccessful or the link is manual), the dotted border should appear in the Windows "disabled text" color (typically gray), suggesting the link is likely out of date. Only the container document's first level objects should be bordered. For example, if in Figure 7 "show objects" were chosen from the outer-level container (Microsoft Word), then only the worksheet object would be bordered, not the graph object nested inside of it. These borders should be clearly distinct from the visualizations for the other states, and are useful to distinguish embedded from linked objects.

An inactive object may be selected by single clicking anywhere within its extent, or it may be double-clicked which will perform its primary verb. The state diagram in Figure 27 shows that double-clicking will activate or open the object for editing, the usual primary verbs.

U.S. Compact Disc vs. LP Sales ($)

	1983	1987	1991
CD's	6,345K	18,652K	32,657K
LP's	31,538K	26,571K	17,429K
Total	37,883K	45,223K	50,086K

Figure 21. Unmodified presentation

2.5.2. Selected

The embedded object is selected when it is either single clicked or included in a multiple selection. It is selected (and deselected) and rendered according to the normal highlight rules of its container (see Figure 3), and responds to appropriate commands as any selected object of the container would. When the object is the singly selected, object-specific operations may be performed on it; the container retrieves these verbs from the system registry. When the object is selected the container may supply handles (for resizing, etc.) which affect the object as a unit with respect to the container. It is recommended that resizing an OLE object while it is selected results in a scaling operation, since there is no mechanism by which the container can communicate a cropping area that would be honored by the object when it is active.

 © Microsoft Corporation 1992-1993. All Rights Reserved.

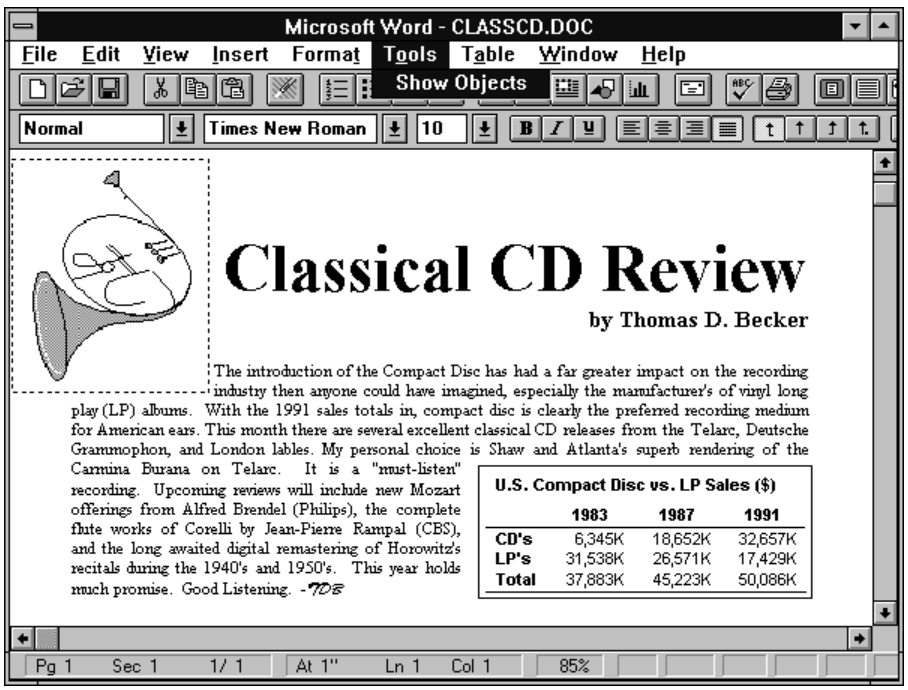


Figure 22. The Show Objects option.

When an object is singly selected, any of its registered verbs may be applied. “Edit” and “Open” will activate and open the object respectively, but other verbs (e.g. “Play”) might perform and then leave the object selected. Any number of single clicks will simply reselect the object, while clicking outside will deselect the object. Verbs whose appropriateness depends on the state of the object should ideally enable and disable appropriately. For example, a media object which has Play and Rewind as verbs should disable Rewind when the object is already at the beginning. Similarly if an open object has two verbs Edit (for in-place editing) and Open (for opened editing), Edit should be disabled since the object cannot directly achieve the in-place active state without first closing.

2.5.3. Active

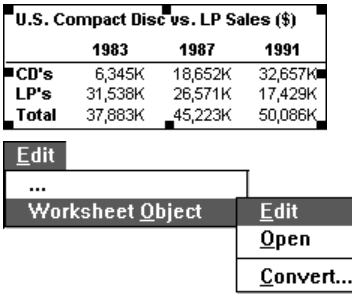


Figure 23. Selected OLE object.

OLE 2 objects may enter an *active* state in which the user may interact with the object’s contents in-place, reusing its container document’s window for its application’s menus and interface controls. The user can make an object active either by performing its appropriate verb (“Edit”), double-clicking it (since for many objects the primary verb will be “Edit”), or selecting the object and pressing Enter. If Enter already has reserved meaning within the container, then Alt + Enter is recommended. When an object becomes active its application’s menus and interface controls are grafted into the document’s window and apply over the extent of the active object. Frame adornments appear outside the extent of the object, and thus may cover neighboring material in the document temporarily. Row/column headers (as pictured below), handles, or scrollbars are examples of frame adornments an object may wish to present. Scrollbars would allow the scrolling of a large spreadsheet within the object’s viewport for example. The object and its frame adornments are surrounded by a black diagonal hatch border as an indication of the active state and to suggest the area of focus. The hatch is always black; it does not change color as focus changes between windows. There is only one object activated at a time; there is no attempt to activate all objects that use the same application as a set.[3] The hatch pattern is comprised of right-ascending diagonal lines as

[3] If for instance all Paintbrush Bitmaps were activated together, commands such as "Select All" or "Clear All" are ambiguous. The user would not know which object(s) would be affected by such document-scoped commands.

illustrated below. The object takes on the appearance which is best suited for its own editing: frame adornments may appear, table gridlines, handles, and other editing aids. The hatch border is considered to be part of the object's territory so it is the object's cursor that appears when the mouse hovers above the border. Clicking in the hatch pattern (and not on resize handles) should be re-interpreted by the object as clicking just inside the edge of the border. The hatch area is effectively a click slop zone that prevents inadvertent deactivations and makes it easier to select the contents of the object which lies right along its edge. The examples below show the border around an active worksheet (notice the border surrounds frame adornments).

Should the container may set at a view-scale (zoom ratio) which the object cannot match in order to perform in-place activation, the object should instead open into a separate window; if the object does not support an open mode, then it should not respond to the verb but issue an appropriate error message (in a dialog) indicating why.

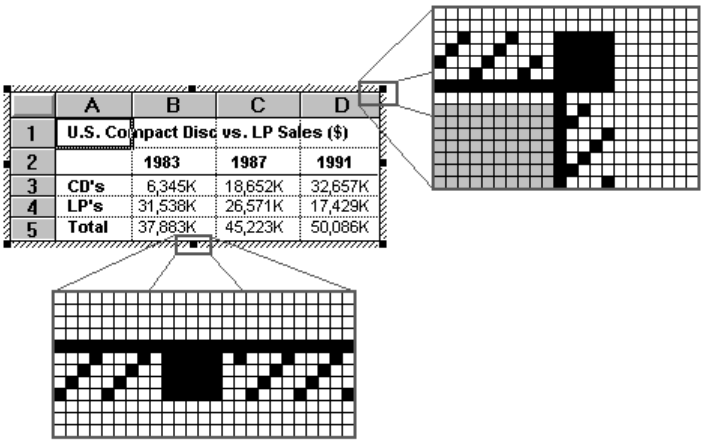


Figure 24. Hatch border around in-place activated objects.

Note that at any given instant there is at most one object that is active in-place per container. A single click in the container area, or a double click on a new OLE object (which may be nested in the currently active object) deactivates the current object and gives the focus to the new object.

An active object may be deactivated by clicking outside its extent in the container document or by pressing the Escape key. If an object uses the Escape key at all times, it is recommended that Shift+Escape is used to deactivate, after which it becomes the selected element of its container.

Edits made to an active object immediately and automatically become apart of the container document, just like edits to native data. Consequently, there is no “update changes?” prompt when an in-place active object deactivates. Of course, changes to the entire document, embedded or otherwise, can be abandoned by declining to save the file to disk. As we shall later see, in-place active objects participate in the Undo stack of the window in which they are activated.

Those objects which support resizing while in-place active should include square resize handles within the active hatch pattern. The solid black handles should be of the same width as the hatch pattern and have a single white pixel separation from the diagonal lines. It is recommended that in-place resizing exposes more or less of the object's content (adding or removing rows/columns in the case of this worksheet). In-place resizing should be seen as adjusting the viewport rather than scaling the object's appearance. Certain objects however may default to in-place scaling if cropping is not meaningful.

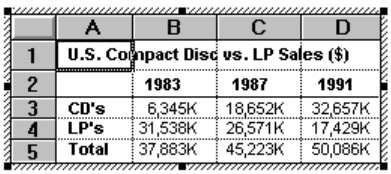


Figure 25. In-place resizing.

2.5.4. Open

The semantics of this state are similar to the open state introduced in OLE 1 with one important distinction in the user model. Whereas in OLE 1 the open object application was seen as a separate application which updates its changes back to the container document, OLE 2 presents an open object application as merely an alternate viewer *onto* the same object still within the container document. There is no sense of a local version kept within the open object application: the opened object in the document and the object within the open window are one in the same. Therefore edits immediately and automatically are reflected in the object (beneath the hatch pattern, see Figure 11) in the document and there is no longer the need for the update confirmation upon exiting the open window. Nevertheless, open object applications may still wish to include an "Update <source file>" command for the user to insist an update at any time. This is useful if the application's "real-time" image updates are not very frequent since they may in general be computationally expensive. In any event, an update is always automatically performed at the time that the open window is closed. "Import File..."-like commands are still appropriate as they will conceptually import a file into the document directly. This new interpretation of open object applications better unifies open and in-place activation: in either case the user is interacting with objects at the document level directly without updating latencies. The only difference is the matter of *where* the interaction is performed: in-place or in an alternate viewer looking at the same spot in the document.

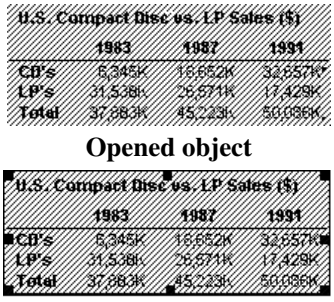


Figure 26. Open object appearance

It is recommended for an OLE 2 object which is capable of in-place activation to also include the "Open" verb in order to give the user the opportunity of seeing more of the object or seeing the object in a different view state. Ideally an in-place object should support in-place editing at arbitrary view scales since its container may be scaled unpredictably. If an object cannot accommodate in-place editing in its container's current view scale or if its container does not even support in-place editing, the object should "Open" into a separate window for editing. When an object is opened, it is the selected object of the container document. Note also that when the container document is printed, the presentation form of objects should be used; neither the open nor active hatch patterns should ever appear in the printed document since these are meta-appearances (like selection indication) and not part of the content.

Any number of objects may be in the opened state. The object is deactivated (closed) by closing its application window which updates the embedded object in the container document. Since object application windows can be amodal, the user is free to switch between object application and document windows and edit independently[4]. It follows that the opened object may participate in the container's selections as usual.

2.5.5. Object State Transitions

The in-place activation examples has shown the OLE 2 standard *outside-in* rule of activation; that is, any number of single-clicks within or on the border of an object will select the object as a whole, and an explicit activation verb or double-clicking is necessary in order to tunnel into the object and work with its contents. All OLE 2 objects should follow the outside-in rule of activation. From the user's perspective, outside-in objects have a tendency to be selected, and require explicit action to activate them. The outside-in rule is motivated by: 1) many object applications require several seconds to load, and thus activating objects indiscriminately upon every mouse event would be unmanageable; instead the user makes a deliberate action to activate the object, and 2) outside-in objects may be easily selected since the user need only click somewhere within the object's extent. Container applications should display the northwest arrow cursor above an outside-in OLE object when it is not activated, as an indication that it behaves as a single opaque item. Figure 27 shows the state transition diagram for outside-in objects. Not pictured in the

[4] If a mini-server is implemented as modal, it must be closed before the user may return to the container document.

 © Microsoft Corporation 1992-1993. All Rights Reserved.

diagram is the effect of double-clicking an opened object; this brings the object application window to the front and activates it (similar to surfacing a window by double-clicking its open icon).

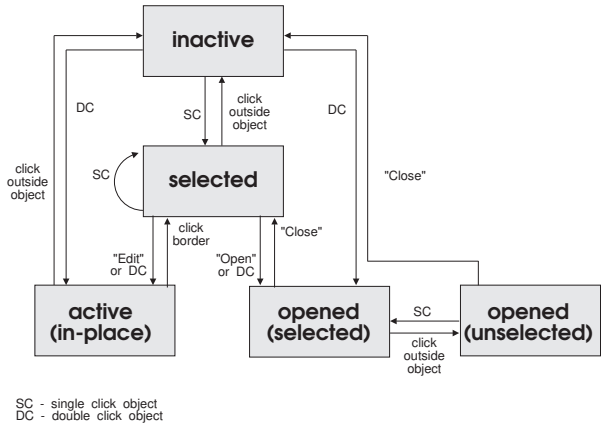


Figure 27. Object state transition diagram for outside-in object applications.

2.5.6. Undo for Active and Open objects

Because different applications take control of a window in the case of in-place activation, there is a question about how commands like "Undo" or "Redo" are handled. How are the actions performed within an in-place object reconciled with actions performed on the native data of the container with regards to undo? OLE 2's undo model specifies there is *a single undo stack per open window;* that is, all undo-able actions whether generated by in-place object applications or the container's application conceptually accumulate on the same undo state sequence. This means that issuing "Undo" from either the container's menus or in-place object application's menus will undo the last undo-able action performed in that open window, regardless if it occurred within or outside an in-place object. If the container has the focus and the last action in the window occurred within an embedded object, then undo will wake the embedded object, revert the action, and leave the embedded object active in-place.

How does this rule apply to open object applications? Again since each open window manages a single stack of undo-able states, actions performed in an open object application are local to that object's window and consequently must be undone from there; actions in the open object application (even if they cause updates in the container) do not contribute to the undo state of the container.

Sending a verb to an object (or double-clicking) is *not* an undo-able action, so it therefore does not add to a container's undo thread. This includes opening an object into another window for editing: if the user unintentionally opens an object, (s)he will dismiss the object application normally since the "open" action cannot be undone from the container.

Figure 28 shows two windows, container Window A which has an in-place active object and an open object application Window B. Between the two windows there have been a total of nine undo-able actions performed in the order and at the location indicated by the circled numbers. The resulting undo stacks are displayed beneath the windows.[5] Notice that the in-place and native actions within Window A have been serialized into the same stack, while the actions in Window B have accumulated on its own separate stack.

The actions discussed so far are bound to a single window. What about actions that span multiple windows, such as drag and drop? The ideal situation would be that in general when a single action spans multiple windows, the action as a whole would be undoable from the last window involved since in almost all cases the user will be focused on that window when (s)he realizes the mistake. So if the user had drag and dropped an item from Window A into Window B of Figure 28, the action would be appended to Window B's undo thread; undoing it would undo the whole drag-drop operation. Unfortunately, in OLE 2 no tech-

[5] The thread of undo states pictured here is not necessarily meant to imply n-level undo. It is meant as a timeline of actions which may be undone at 0, 1, or more levels, depending on what the container/object cooperation supports.

 © Microsoft Corporation 1992-1993. All Rights Reserved.

nical support exists for this sort of multi-window undo coordination. As a result, multi-window actions at best create independently undoable actions in each window that participates in the action.

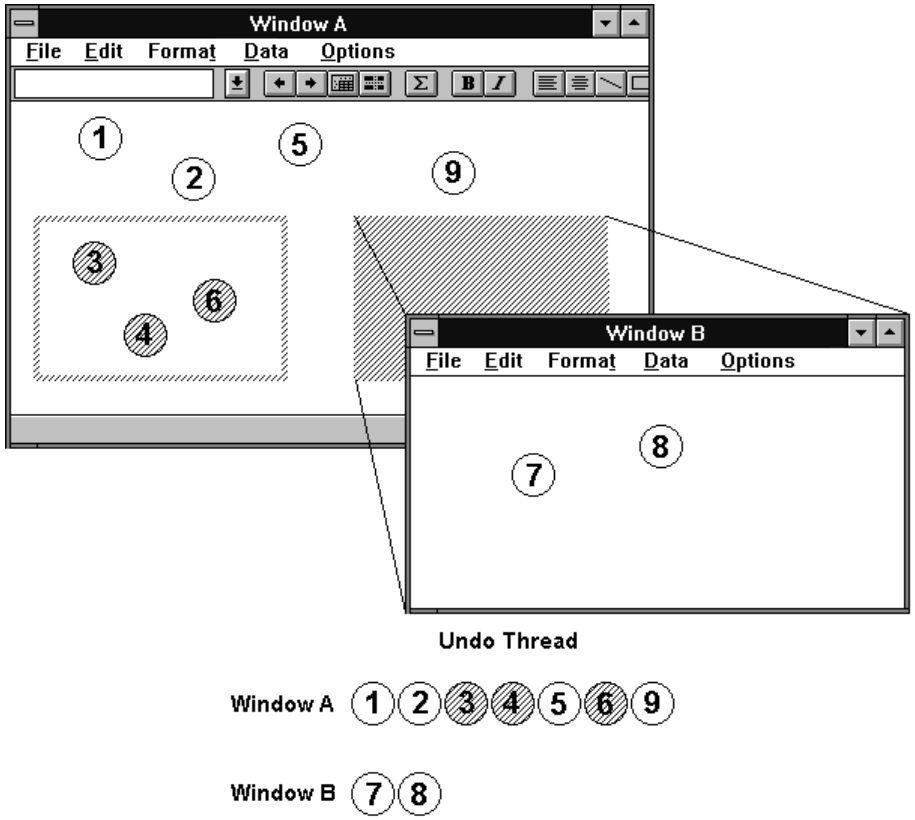


Figure 28. Undo stacks of in-place and open objects.

2.6. Commands

As different objects are selected or activated, different commands need to be made accessible through the user interface. This is the case even in OLE 1, where an object's *verbs* need to be exposed when it is selected. In essence, an OLE 1 interface consists of commands for the container and, if an embedded object is selected, additional commands for that object. Thus, in OLE 1, there are at most two players in the interface at any time: the container (which controls workspace and editing operations, and whose commands require no special categorization) and the embedded object, for which OLE 1 introduced object verbs through the OLE registration.

2.6.1. Menus

The addition of in-place activation functionality presents potential problems, as several objects may simultaneously need to expose their commands in the interface. If no restrictions are imposed on where these commands can appear, users will be faced with interfaces that change unpredictably, and they will be unable to reliably identify the object to which a given command applies. Developers, for their part, will be faced with task of having to accommodate complex negotiations among objects in order to build the menus. In order to avoid these problems, OLE 2 defines a classification of menus that segregates the interface based on menu groupings. This classification is designed to enhance the usability of the interface by regularizing and limiting the changes that occur in the interface as different objects come and go from the interface. Below are the description of these menu categories.

Workspace Menu(s)

The top container in an application window controls the *workspace* of that window. That is, it is responsible for the organization of windows (MDI), file level operations, and how edits are ultimately saved.

The top container should supply a single "File" menu which contains document / file level commands such as Open, Close, Save, and Print. In OLE 1, if the container was an opened object application,[6] the commands in its "File" menu were modified according to the guidelines in the Chapter 9, "Object Linking and Embedding," of *The Windows Interface.*[7] These recommendations included the use of a "Update container document?"-type dialog when the open editing window was closed. With OLE 2, updates are automatic between the open object application window and the container document. There is no need for the update confirmation upon closing the open window: closing the open window should create an undoable action in the object's container.

MDI application should also supply a "Window" menu which controls the child windows of the application.

Workspace Menus remain in the menu bar at all times no matter which object is active.

Since at first users may not expect the File menu to remain under the container application's control, they may think File / Exit only exits the in-place object application and returns to the main document. To their surprise they would see the whole document closed if they did this. To minimize the chance of this happening, container applications should use "Exit <container app>" instead of just "Exit" to make it more explicit what the command really affects.

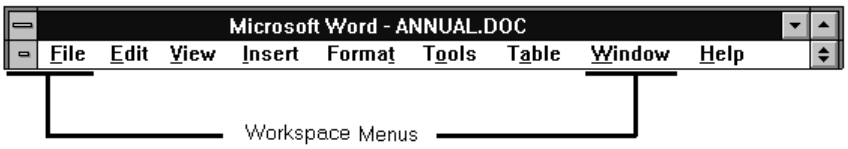


Figure 29. Workspace Menus

Active Editor Menu(s)

Nearly all OLE objects must supply Active Editor menus which hold the bulk of its commands, those which actually operate on its content. Commands like moving, deleting, search/replacing, creating new items, applying tools, styles, and help would be located in these menus. Active Editor commands apply only within the extent of the active object. Active Editor Menus occupy the bulk of the menu bar and may be slightly different depending on whether the object is activated in-place or opened. As the name suggests, they are executed by whichever object is currently active (which may be the top container application). Objects which use direct-manipulation or OLE verbs as their sole user interface need not provide Active Editor menus, nor need they alter the menu bar when activated. None of these menus should be named "File" or "Window" (appropriately localized) since those titles are used by Workspace Menus.

Notice the title bar always displays the name of the application which has its Active Editor menus present. This can either be the name of the container application or the name of the object application during in-place activation. In-place objects which *do not* change the menus will not have their application name displayed in the title bar; the container application name will remain in this case. If the title bar *does* display the in-place object application name, then it should in addition prepend '<*short type name*> in' before the document name. So the syntax for an SDI (or an MDI with maximized child) title bar is:

<*active editor name*> - [<*short type name*> in] <*container-document*>.

---

[6] Of course this discussion only relates to the menu bars of *full* servers, since *mini*-servers are dialog boxes and do not have menu bars.
[7] *The Windows Interface, An Application Design Guide,* Microsoft Press, Redmond, WA, USA, 1992

For an MDI container whose active child window is restored (refer to Figure 38), the syntax is:

MDI title: <active editor>

Child title: [<*short type name*> in] <*container-document*>

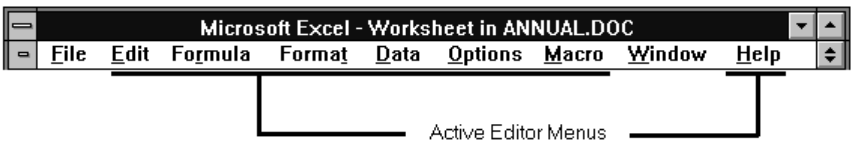


Figure 30. Active Editor Menus

The View menu is categorized as an Active Editor menu and as such it holds viewing commands which only apply within the active object Note that if a container application wishes to keep its document/window level "viewing" commands available while an object is active, it should locate them on a Workspace menu (preferably the "Window" menu if it is present).

Selected Object (Sub)Menu

All containers must include this single menu containing the verbs for any currently selected OLE object. This menu is preferably a submenu of an Active Editor Menu (e.g., Edit; this is the interface suggested for OLE 1 as pictured below), or possibly as its own menu on the menu bar. The submenu uses the short type name of the object type name and is prepended with the word "Linked" if the object is a link. So the syntax for the menu is: [Linked] <*short type name*> Object -> <*verb0*>, ..., <*verbN*>, Convert.... The first letter of the appropriately localized word "Object" should be underscored as the stable mnemonic character for keyboard users. When there is no object selected, the menu command is simply "Object" and it is disabled.

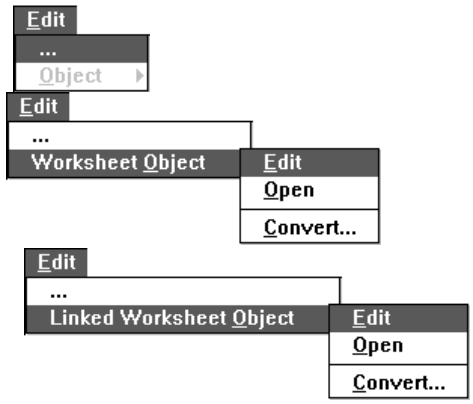


Figure 31. Selected Object Menu

To summarize how these menus mesh on the menu bar:

1. Workspace Menus are present all times in each window whether it is the top container or an opened object application.
2. The "File" menu commands in an open object application window are qualified to reference the container document (OLE 1 User Interface Style Guide), and no longer bring up the update confirmation. As in OLE 1.0, open object applications (particularly SDI applications) are encouraged to replace their Open and New commands with something like Import in order not to sever the embedding connection with its container.
3. The currently active object (possibly the container itself) supplies the Active Editor Menu(s) as the bulk of the menu bar.
4. If an object is selected, its commands are available through its Selected Object Menu within an Active Editor Menu.

2.6.2. Keyboard Commands

In addition to integrating Workspace, Active Editor, and Selected Object Menus, the keyboard commands associated with these menus must be blended as well. These keyboard commands include:

1. *Mnemonic Keys*. Those Alt key sequences that pull down menus and choose commands according to which letters are underscored. The rule is Alt+<letter> will pull down the corresponding menu, followed by additional letters which invoke the command.
2. *Shortcut Keys*. These are the quick access key commands which may use any combination of the Ctrl, Shift, and Alt modifier keys.

The integration strategy is intended to offer all of the mnemonic and shortcut keys used in Workspace, Active Editor, and Selected Object menus to the user without ambiguity. To do this we ensure that the reserved key sequences in each of the three categories (as defined in *The Windows Interface*) are not reused by the other categories.

Mnemonic Keys

The only chance for the mnemonics in Workspace and Active Editor Menus to collide is among their menu titles, since presumably command mnemonics within each menu have already been uniquely assigned. Therefore containers and objects must have an understanding of which menu title letters they may use.

1. Workspace Menus should use File and Window as their mnemonics (or the appropriately localized File and Window words).
2. In-place object applications may underscore letters other than those used for workspace menus in their Active Editor Menu titles. (If an in-place object application has previously existed as a stand alone application, its menus likely avoid these letters already.)
3. OLE verbs should be registered with uniquely assigned mnemonics.

If despite these guidelines the same mnemonic is used more than once, pressing Alt + <letter> will cycle through the candidates highlighting the next match each time it is pressed. To actually engage the item (whether it is a menu title or command) requires pressing the return key when it is highlighted.[8]

Shortcut Keys

In this case the object application is given the first opportunity to field a shortcut key, passing it on to the container if no match was made. Here again it is important for the object application to avoid using shortcut keys that are likely to be taken by the container application. Both container and object applications should use the shortcut key guidelines outlined in *The Windows Interface* for their Workspace and Active Editor Menu commands respectively. Active Editor commands should make use of the standard editing shortcut keys and be mindful not to use Workspace shortcut keys. There is no provision for registering shortcut keys for Selected Object commands.

In the event that the container and object applications do share a common shortcut key, the active editor takes it. That is, if the object application is active it will get the shortcut key with no negotiation or hand off with the container application. Likewise, if the container application is the active editor (even if an OLE object is selected), it will service the key. In these collision cases, users will need to set the focus to the application that they wish to handle the shortcut key.

2.6.3. Pop-up Menus

Just as keyboard commands are shortcuts to the pull-down menu commands, pop-up menus (also called shortcut or context menus) provide quick access to frequently used and contextual commands using the mouse. Although pop-up menus are implemented by applications themselves, this section recommends how pop-up menus are to be used in conjunction OLE objects.

Pop-up menus as defined here are an auxiliary command interface and therefore contain a subset of the pull-down menu commands. Pop-up menus are “popped-up” at the pointer’s location eliminating the need to navigate to a menu or button bar and they do not take up any dedicated screen real estate since they are only displayed upon demand.

[8] This behaviour is built in to the Windows 3.1 menu managment code.

 © Microsoft Corporation 1992-1993. All Rights Reserved.

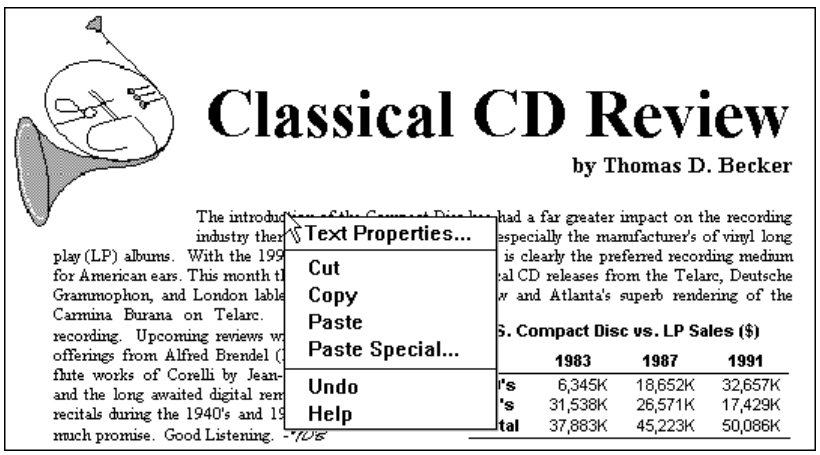


Figure 32. Pop-up menu for a text selection.

A pop-up menu looks similar to a standard drop-down menu in that it contains a frame, a shadow, and a list of commands with ellipses and separator lines. By pressing down the right mouse button, a pop-up menu appears 2 pixels beneath and to the right of the current pointer position, displaying commands that relate to the object(s) beneath the pointer. No items on the pop-up menu should initially be hilighted; it is only after the mouse has entered into the menu that a command should be hilighted. The position of the menu allows the user to conveniently move the pointer down into the menu, but does not initially position the pointer on an item to prevent the user from inadvertently selecting a command. If the pointer is positioned such that the menu would be clipped/appear off screen (i.e. at the bottom or right side of the screen), the menu is adjusted so that if fully appears on the screen. Generally, this means that the top-left corner of the menu is slid to the left to avoid the right screen edge or slid up to avoid the bottom screen edge. If the menu would appear both off the bottom and the right edge of the screen, the pop-up menu would be positioned to the left and above the pointer.

The pop-up menu is displayed as the right mouse button is pressed down over an object or selection of objects. If the pointer is moved into the menu and the button is released, the command beneath the pointer is executed. If instead the button is simply released at the button down point (specifically within 4 pixels of the button down point), then the menu remains displayed. However, if the pointer is moved and the button is released outside the menu, then the menu is removed (canceled). This is identical to the way drop-down menus display on a click in the menu bar.

Clicking either mouse button within the menu while the menu is displayed triggers that command and removes the menu. While the mouse button is held down, selection highlight feedback is provided on the selected command. The action is not executed until the button is released on the command, allowing the user to drag through the menu and highlight other commands or drag off and release to cancel the menu.

While a pop-up menu is displayed, clicking either mouse button outside the menu cancels the menu. This implies that there is at most one pop-up menu displayed at any time. If the left button is used, the menu is removed and a new selection is begun. Unlike current drop-down menus, the top command of a pop-up menu should not be initially hilighted when the menu first appears; it is only after the mouse pointer has entered the menu that any command should be hilighted. However, if the right button is clicked, only the menu is removed. As with drop-down menus, the ESC key will cancel the menu at any point.

Generally, a pop-up menu is invoked over an explicit selection; that is, a selection made with the Select button (e.g. click or drag, Shift + click/drag, Ctrl + click/drag). However, pressing the right button over certain objects will both determine the selection *and* bring up the pop-up menu. For example, a range of text or a collection of items must likely be first selected by the user before invoking its pop-up menu; pressing the right button over a more discrete target like a spreadsheet cell or an icon, however, might automatically make it the current selection and display its pop-up menu. This closely follows the rule for dragging and dropping since some items must first be selected before dragging while others may be dragged directly. The direct pop-up approach is functionally equivalent to having first selected an item and then pressing the right button for its menu; it is merely an efficiency.

Similarly, the right button may be pressed over items that live outside of a normal selection context: items like interface controls or the screen background. For instance, a pop-up menu over a scroll bar may display useful navigation commands; a pop-up menu over a control bar button may display commands to re-program its function. Invoking pop-up menus on these objects has no effect on the selection within window's content; they are completely separate domains.

The pop-up menu interface is primarily designed as an optimization for mouse users. There are no mnemonic keys or shortcut keys associated specifically with a pop-up menu, since those devices are defined and are already learned in terms of the drop-down menus.

Pop-up menus are intended by design to be an efficient means of accessing common, contextual commands. Therefore, it defeats the purpose of the interface to include too many commands or to include multi-level cascade/hierarchical submenus. (Cascade submenus are acceptable, but should be limited to a single level. This follows the same guidelines as drop-down menus. Multi-level menus of any variety results in complexity and inefficiency in accessing commands.) Pop-up menus should contain those frequently used commands and not simply repeat the entire menu tree found on menu bars. The pop-up may include commands that specifically apply to the menu's target or its particular context. As a guideline, it is recommended that applications arrange commands from top to bottom in decreasing frequency of use: the most frequently used commands should be at the top to minimize overall mouse travel For uniformity, pop-up menus on a selected OLE object should display the object's verbs in order and in-line, unlike the cascade verb menu recommendation for the Edit pull-down menu. It is also recommended that the pop-up menus have no more than ten commands, bearing in mind single level cascade menus or dialogs are acceptable.

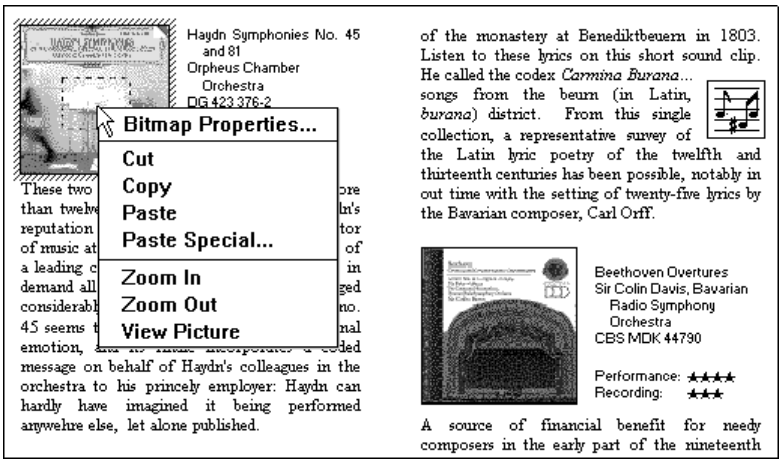


Figure 33. Possible pop-up menu for a selection in an active Microsoft Paintbrush Picture.

When an object is active, like the Microsoft Paintbrush Picture in Figure 33, its pop-menus are in effect for its selections. Notice how a pop-up menu for a bitmap selection might be implemented.

When an object is selected, the pop-up menu provided by the container should list the objects verbs "in-line" rather than in a cascade menu as on the Edit.Object menu. The syntax should follow: <*verb*> [Linked] <*short type name*> (i.e. Edit Picture, Play Linked Recording). It is not necessary to append the word "Object" after the *short type name* since mnemonics are not present in pop-up menus and including "Object" would make the menu unreasonably large. The Convert... command follows the verbs as shown in Figure 34.

2.6.4. Control Bars, Frame Adornments, Floating Palettes

OLE 2 strives to present a object application's commands and interface controls in their entirety, so that users may enjoy uncompromised functionality during in-place activation. OLE 2 employs a replacement strategy for arbitrating these type of interface controls. Just as with menus on the menu bar, control bars, palettes, and frame adornments come and go as entire sets, there is no attempt to integrate at the widget or command level.[9] This is relatively simple for floating palettes which are independent from the container window and are solely under the control of the active object. It is more complex for control bars and frame

[9] Such fine grain integration was seriously investigated, since it appears very desirable. However, no workable solution was found.

adornments which may use the same space in the container window, requiring some repainting, relocation, or resizing of objects.

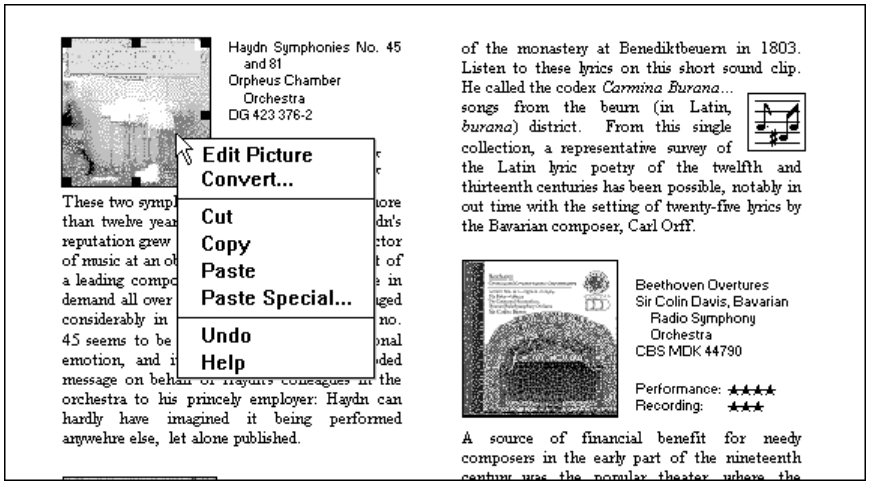


Figure 34. Pop-up menu for an embedded Microsoft Paintbrush Picture in Microsoft Word.

In keeping with the OLE 2 theme of document centered interaction, swapping active objects' interfaces should disturb the document's appearance and position as little as possible. Integrating drop-down and pop-up menus is fairly straightforward since they are confined within a particular area and already follow some standard conventions. Control bars and the like are less predictably integrated, but the following guidelines should help blend the interfaces more seamlessly. Figures 35 and 36 show some examples of these devices.

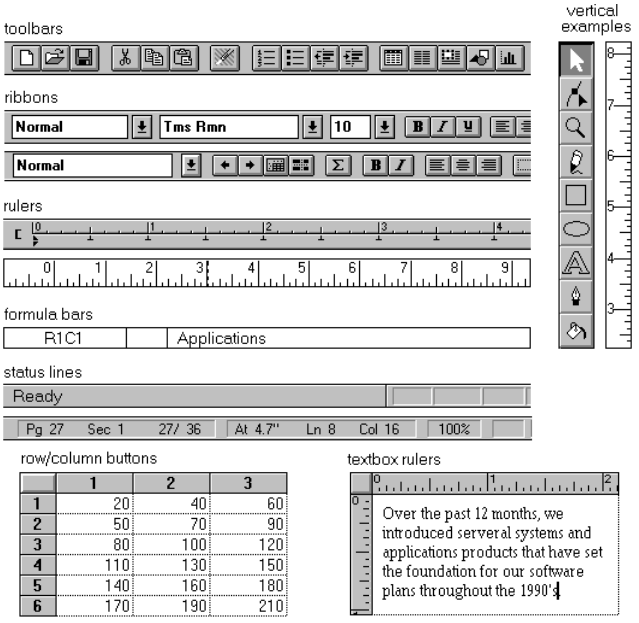


Figure 35. Control bar and frame adornment examples

From OLE's perspective control bars, frame adornments, and floating palettes are all basically the same devices differing primarily in their location and the degree of shared control between container and object. There are four locations in the interface where these types of controls may reside, and the choice of its location is principally determined by the scope over which it applies.

1. *Object Frame.* Object specific controls like a table header or a local coordinate ruler can be placed directly adjacent to the object itself for tightly coupled interaction between the object and its interface. An object (such as a spreadsheet) may include scrollbars if its content extends beyond the boundaries of its frame.
2. *Pane Frame.* Controls which are specific to a view or a single document should be located at the pane level. Rulers and viewing tools are common examples.
3. *Application Window Frame.* Tools which apply to the entire document (or documents in the case of a MDI) may be attached just inside any edge of the application window frame. Popular examples include ribbons, drawing tools, and status lines. Because of Windows behavior, however, the MDI child windows will shift up and down as different sized toolbars come and go with context switches. This can be disruptive to the user's task so it

is recommended that toolbars are either floated or placed elsewhere when MDI child windows are in the "restored" state.

4. *Floating.* Objects may wish to "float" their tools above the document allowing the user to arrange them as desired.

These are illustrated in Figure 37.

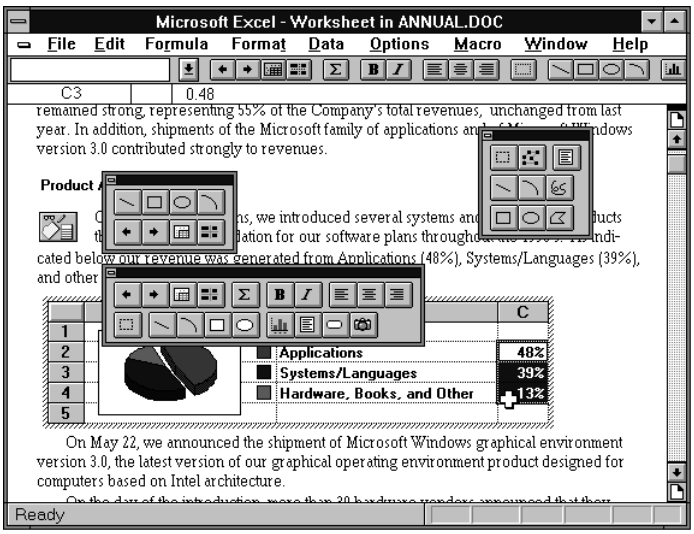


Figure 36. Floating palette examples.

As an object becomes active it requests a specific area from its container to post its tools. The container may decide to:

1. Replace its tool(s) with the object's if the requested space is already occupied by a container tool.
2. Add the object's tool(s) requested space is not already occupied by a container tool.
3. Refuse to put up the tool(s)at all (least desirable).

Since container control bars may still be visible while an object is active (like the pane ruler in Figure 37), they are still available for use simply by interacting with them, which may result in reactivating the container application (the container decides whether it necessitates its own reactivation, or it is fine to leave the object activated). The protocols for this negotiation is described later in the API part of this specification. If control bars contain "workspace" commands such as save, print, or open icons they must be disabled and preferably visually distinguished as unavailable.

As windows are resized and the document is scrolled, these interface controls will be forced to clip with respect to their containers.

1. An *active object and its frame adornments* will be clipped by its immediate window pane just like all document content; frame adornments can be thought of as handles which lie in the same plane as the object. When the object is clipped, the visible part of the object can be edited in-place and the visible frame adornments are operational. Some container applications may scroll at certain increments which will prevent portions of an embedded object from being edited in place. Consider for example a large picture embedded in an Microsoft Excel worksheet cell. Microsoft Excel scrolls vertically in complete row increments meaning that the top of the pane is always aligned with the top edge of a row. If the embedded picture is too large to fit within the pane at one time, then its bottom portion will always be clipped and consequently never viewed or edited in-place. In cases like this the user will likely open the picture into its own open window for editing. Frame adornments of nested embedded objects (e.g. the graph within the worksheet in ANNUAL.DOC) are likewise clipped by the immediate window pane, but not by the extent of any parent object. Objects which are at the very edge of their container's extent or boundary may potentially surface adornments which will extend beyond the bounds of the container's defined area. In this

 © Microsoft Corporation 1992-1993. All Rights Reserved.

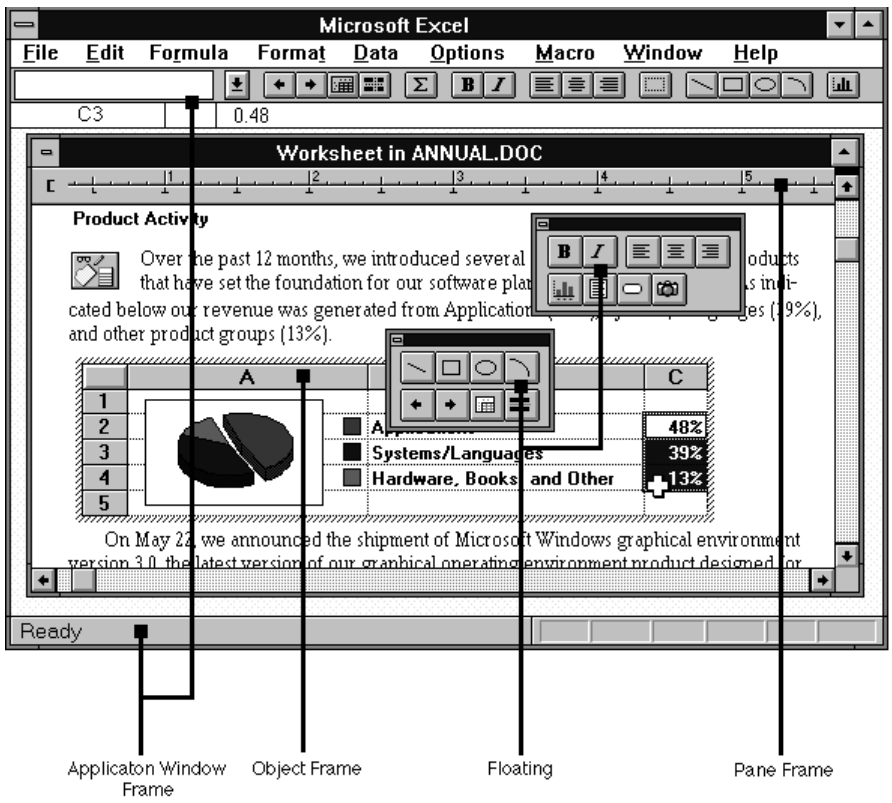


Figure 37. Four possible positions for interface controls.

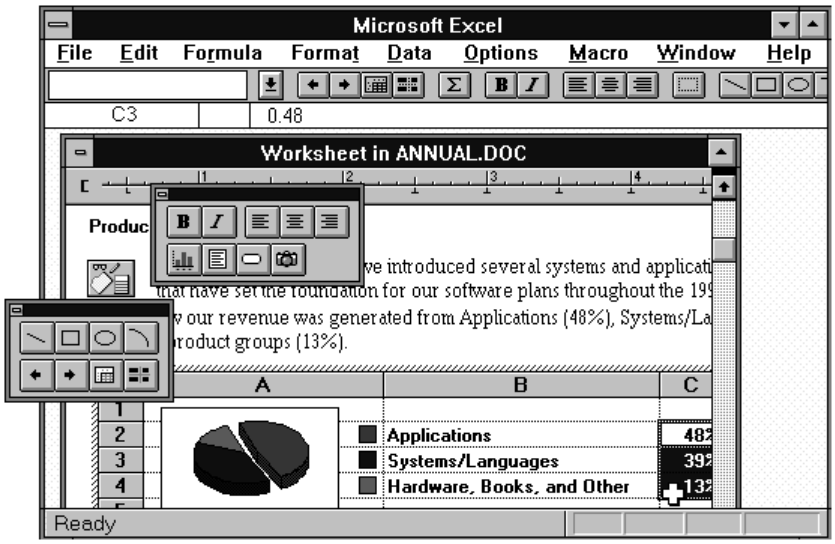


Figure 38. Interface control clipping.

case, if the container can display items which extend beyond the edge, it should display all the adornments; otherwise, the adornments should be clipped at edge of the container. The container should not temporarily move the object with respect to the content just to accommodate the adornments appearance.

2. A *pane level* control may be potentially clipped by the application window frame (in the case of MDI), and of course an *application window level* control may be clipped by other Windows applications.
3. A *floating control* floats above all windows.

Alternating between different size control bars in the document pane should have no effect on the content's position with respect to the pane. In other words, there will be no automatic scrolling provision to guarantee the active object remains in view, since in many cases that is not achievable. Consequently pane control bars are thought to lay atop the document's content, requiring the container application to repaint parts of the pane as control bars are remove or replaced with smaller ones. By preserving the content's position, the user's focus will not be disturbed by the active object bumping down to accommodate the new set of control bars, and bumping back up when it is deactivated. This rule is to be followed even if the arriving control bars will clip the object of interest. There is one exception when the container application may wish to scroll the document as a result of a control bar change: if the document is at its very top or bottom, and removing a control bar would expose an undefined area (area beyond the document's extent), then the document may scroll to take up the space left by the control bar.

2.7. Linked versus Embedded Objects

An *embedded* OLE object is a piece of data that retains the original editing and operating capability of its creating application while it physically resides in a document of another application. An embedded object is *wholly* and *exclusively* contained within that one document, and is consequently saved and moved along with that document. With OLE 2 a *linked* object is an updating image of another piece of data (native or embedded) which physically resides somewhere else; at another point in the same document or within a different document altogether. A linked object does not contain *real* underlying data per se, rather it is *stand-in* that enables its source to be visually present in other documents and provides the user of those documents access to the source's verbs. In fact, from all appearances the user may interact with a linked object as if the source were indeed there. So the norm of using an OLE document is interacting with embedded and linked data with little attention paid to which is which. Of course there are some additional issues with linked objects and this sections points them out.

2.7.1. Automatic and Manual Updating

When a link is created it is by default an *automatic* link, that is, whenever the appearance of its source data changes, the link's appearance likewise changes without any request from the user. Therefore it is not advised to issue any kind of "Update Automatic Links Now?" dialogs, but rather always perform the update displaying the progress indicator pictured in Figure 51. If the user wishes to exercise control over when links are updated, (s)he should then designate them as manual instead of than having question if automatic links have been updated or not. The user may specify a link to be *manual* via the Links... dialog which will only update its appearance upon request by the user. This request is issued by choosing the "Update Now" button within the Links... dialog and optionally in conjunction with container's usual "update fields" or "recalc" action.

Automatic and manual links may be cascaded to achieve useful document flow networks. A publisher of a chart A might like to distribute his pricing data on a controlled basis so that subscribers of the chart see the data only when it is ready. The publisher makes available a manually-linked chart B to his subscribers on a read-only basis; meaning that only the publisher has the right to tell B when to update. Therefore whenever chart A is in an acceptable state for distribution, the publisher updates B for public consumption. On the subscription side, some subscribers may be interested in always having the up to date version of chart B, so they would set up an automatic link from B into their documents. Other more discriminating subscribers would manually link to chart B, giving them the option of accepting or refusing updates from B. The chart network may be similarly cascaded to permit subscribers of B to embellish the chart and in turn offer their versions on a controlled basis to additional subscribers.

2.7.2. Verbs and Links

As explained above, an object responds to the same set of verbs defined for its particular type regardless if it is embedded or linked. When a user issues a verb to a linked object, the verb is conceptually passed back to the source object for processing. In certain cases the linked object will exhibit the result of a verb; in

other cases the source object will be surfaced to handle the verb. Issuing verbs like “play” or “rewind” to a sound recording link, for instance, will appear to operate the object in-place. However, if the user issues a verb which is intended to alter the object’s content data (such as “open” or “edit”), the link *source* is exposed in order to respond to the verb instead of the linked object itself. Links may only “play” in-place but not “edit” in-place since that would wrongly imply that the actual data resides (even if only temporarily) in the document holding the link. It is critical not to blur the model that link sources hold the actual data and do not travel to the calling link’s document for editing. So, in order for link sources to respond to *editing* verbs, the source object is fully awakened (with all of its containing objects and document) to properly respond to the verb. By far the common case for this will be double-clicking a link whose primary verb is “edit”. The source document will open revealing the link source object ready for editing. At first glance this appears similar to “open”-ing an embedded object into a separate window, but it is truly different. Editing a linked object is functionally identical to launching an application on the link source’s document through the normal means. Unlike an opened embedded object which is bound to the container application window and causes the hatch pattern across the object’s face, there is no special association between the link source and link destination application windows. The two applications operate and close independently of each other. The link source may be on an unmounted network server, so it may require the user to first establish access to the source before OLE can open it. If the user tries to open an available source document, the container should post a message of the nature “Please connect to \\foo\bar and retry opening the Worksheet link.”[10]

It is the responsibility of the object application when implementing the execution of an editing verb to notify its immediate container that the source needs to be exposed. Below are the three different situations in which an object might notify its container:

1. *Embedded*. If the verb is executed by an *embedded* object, the container document and parent objects are already exposed so there is no need to wake the object’s container.
2. *Already Opened Link Source Document*. If the verb is issued to a link source whose document is already opened, then the currently active view of that document (in the case of a multi-view or multi-pane application) will scroll as necessary and unveil the source object ready for interaction.
3. *Closed Linked Source Document*. The verb notifies its immediate container requesting it to expose the source to the user. All additional containers are opened resulting in the link source being fully exposed and ready for editing.

2.7.3. Types and Links

A link’s type is a cached copy of its source’s type at the time when the last update was issued. Since it is possible to change the type of a link source object, all links derived from a converted object will bear the old type and verbs until either an update occurs or the link source is launched. Since out of date links may potentially display obsolete verbs to the user, the following recommendation is given for verb mismatch handling:

When a verb is invoked on a link object, the link object compares the cached type with the current type of the link source. If they are the same, the link object forwards the verb invocation on to the source. If they are different, the link object instead so informs its container. In response, the container can do one of the following:

1. If the verb issued from the old link is syntactically identical to one of verbs registered for the source’s new type, execute the verb index of that new verb.
2. If the issued verb is no longer supported by the link source’s new type, issue an error message of the form in Figure 39.

[10] If a link's source is contained within a read-only document, the effects of "editing" verbs will not be saved.

 © Microsoft Corporation 1992-1993. All Rights Reserved.

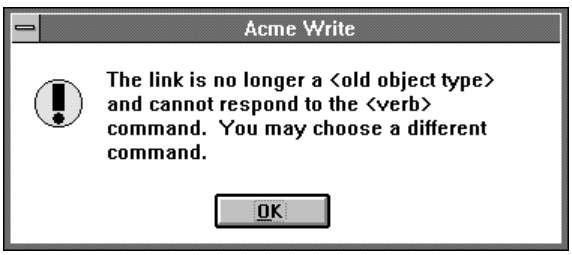


Figure 39. Error message for obsolete link verb

Afterwards in both cases the link adopts the source's new type and bear the new verbs in its menu from then on.

2.7.4. Maintaining Links

A link has three properties: its *type* (or *class* as its known internally), the *name* of its source data, and its updating basis which is either *automatic* (default) or *manual*. The container application supplies a Links... dialog which allows the user to alter the last two of these properties. (A link's type is determined by its source's type which may be changed by converting it to another type). The user may wish to switch between *automatic* and *manual* for the reasons previously described, and may wish to change its source's name if for instance the source has been relocated in another file, directory, database, object, or some other container.

Figure 40 shows the OLE 2 Links dialog. The list pane in the dialog displays the links which are contained in the document. Each line in the list contains the link's source name, object type (short type name), and whether it updates automatically or manually. If a link's source cannot be found, "Unavailable" should appear in the status column.

Allow 15 characters for the short type name field, and enough space for Automatic and Manual to appear completely. As each link in the list is selected, its type, name, and updating basis appear in their entirety at the bottom of the dialog. Break Link will effectively disconnect the selected link, Update Now will force the selected link to connect to its sources and retrieve the latest information, Open Source will open the source document for the selected link, and Change Source will invoke a dialog similar to the standard FileOpen dialog to allow the user to respecify the link source. The Open Source button should be default when clicking within the links list, therefore double-clicking a list item will open the source of the particular link.

The user may enter by typing a source name which in fact does not designate an object which presently exists. Upon hitting OK, the Change Source dialog will prompt "Invalid Source. Do you wish to correct it? Yes, No". Answering Yes will return to the Change Source dialog so the string may be corrected; answering No will cause the container to hold on to the unparsed display name of the link source until such time later as the user successfully causes the link to connect to a newly created object that satisfies the dangling reference. Some containers may choose to allow their users to only connect to presently-valid links.

If multiple links share in common any portion of their source's names (i.e. links which are contained in the *AUDIO* directory, or are inside the file *HORNS.BMP* for instance), then if the user edits that common portion for one link, the container may give him the option to make the same change for the other similar links. This dialog is the utility for users to redirect links to new locations such as in cases when directory or files names have been changed.[11]

[11] Technically, this is supported using the calculus of monikers involving the operations IMoniker::CommonPrefixWith, ::RelativePathTo, and ::ComposeWith to determine which links are "similar," what the similarity is, and what the retargeted link source should be. For further information, see later in this specification.

 © Microsoft Corporation 1992-1993. All Rights Reserved.

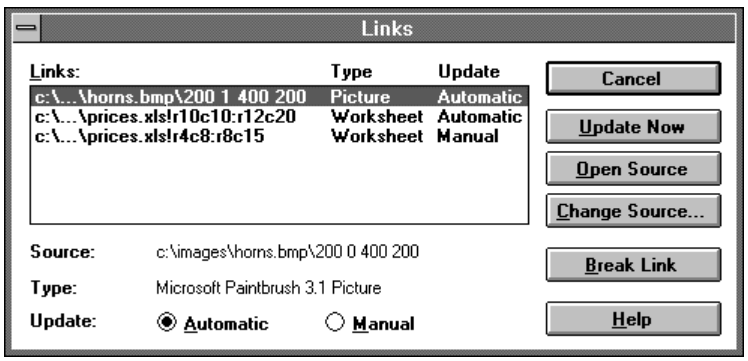


Figure 40. Links Dialog

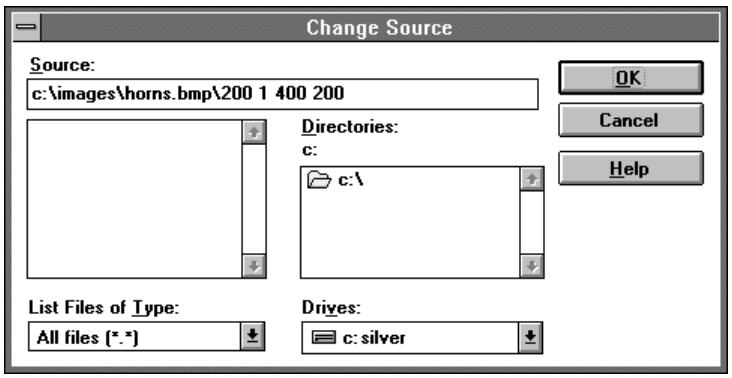


Figure 41. Change Source dialog

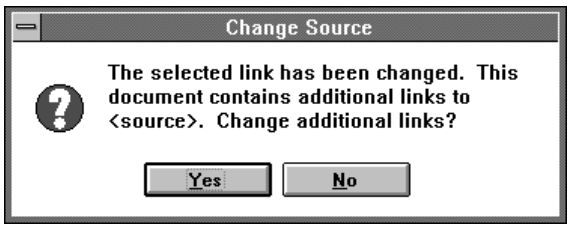


Figure 42. Changing additional links with same source

2.8. Object Transfer Model

So far this specification has described how objects are created and edited, and how OLE 2 integrates command and tool interfaces. This section explains how object are moved, copied, and linked within and across documents. This is done using two different methods: the familiar *clipboard method* invoked through commands from either drop-down or pop-up menus, and the *drag/drop method* which enables users to directly drag objects from one location to another. The universally available clipboard method allows the user to move, copy, and link data, and optionally specify data formats; the drag/drop method allows the user to quickly perform moves, copies, and links when little navigation is needed. The two approaches cooperate to give the user both a global and expedient means of arranging information in documents.

2.8.1. Clipboard Method

Using the clipboard method a user will:

1. Make a selection

 © Microsoft Corporation 1992-1993. All Rights Reserved.

2. Choose either Cut (Ctrl+X) or Copy (Ctrl+V) from the menu (either the Edit pull-down menu or a pop-up menu).
3. Navigate or edit as desired
4. Choose an insertion point (or a selection if it is to be replaced)
5. Choose Paste [<*short type name*>] (Ctrl+V) or Paste Special... to insert the clipboard contents.

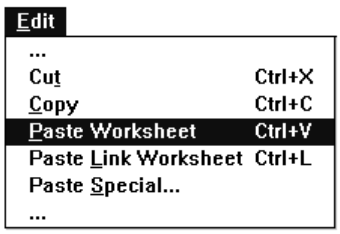
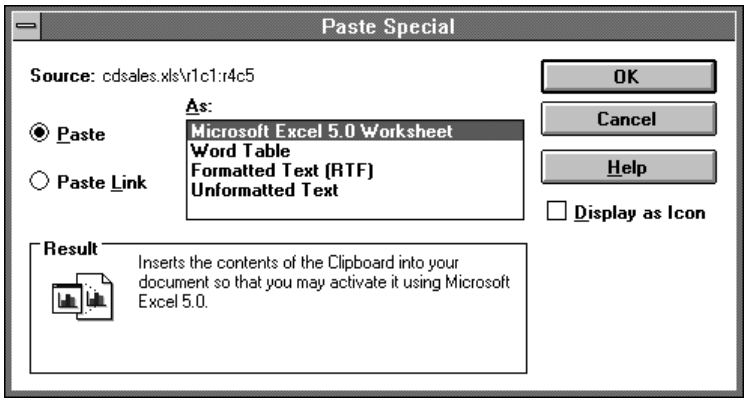


Figure 43. The Paste [*short type name*] and Paste Link [*short type name*] commands

Cut/Paste removes the selection from the source and relocates it at the insertion point, while Copy/Paste inserts an independent duplicate of the selection leaving the original unaffected. Choosing Paste Special... will produce the dialog in Figure 44, which gives the user explicit control over how to Paste the clipboard's contents. (The default Paste will embed the object if possible, unless the destination document can edit the data completely with its own tools, in which case the destination application pastes the data in the format it desires.) If the default Paste will result in an embedding, then command should read Paste <*short type name*> as an indication of that (e.g., Paste Worksheet, Paste Recording). The same is true for Paste Link if the container so desires to include this command on the edit menu (e.g., Paste Link Worksheet, Paste Link Recording). The simple Paste command with no <*short type name*> extension simply means the data will be pasted as native information without OLE involvement. The Paste Special dialog contains the Paste and Paste Link radio buttons, lists the possible formats associated with each radio buttons, and references the display name of the source data (using the same portion of the display name of the link source as was the case for the default icon label; see earlier in this chapter). At the bottom of the dialog is section of result text which describes the result of the currently chosen function and format. Notice the use of pictures beside the help text to communicate the end result of the operation. Also note the same “Display as Icon” option found in the other dialogs.

When the Paste option is chosen, the format list shows the full object type name first (without the trailing “object”) followed by other native data forms. When the clipboard contains an already linked object, its object type should be preceded by "Linked" in the format list. For example, if the user copied a linked Microsoft Excel 5.0 Worksheet to the clipboard, Paste Special would show "Linked Microsoft Excel 5.0 Worksheet" under the Paste format options since a Paste would insert an exact duplicate of the original linked worksheet. Native data formats should be expressed in the same terms as the destination application uses in its own menus. To be especially clear, native format names should begin with the destination application's name, as in *Word* Table. When Paste Link is chosen both the object type (full form) appears and any native format which will support linking. The default formats for the Paste and Paste Link radio options are exactly the same as what the default Paste [<*short type name*>] and Paste Link [<*short type name*>] would use as formats.



 © Microsoft Corporation 1992-1993. All Rights Reserved.

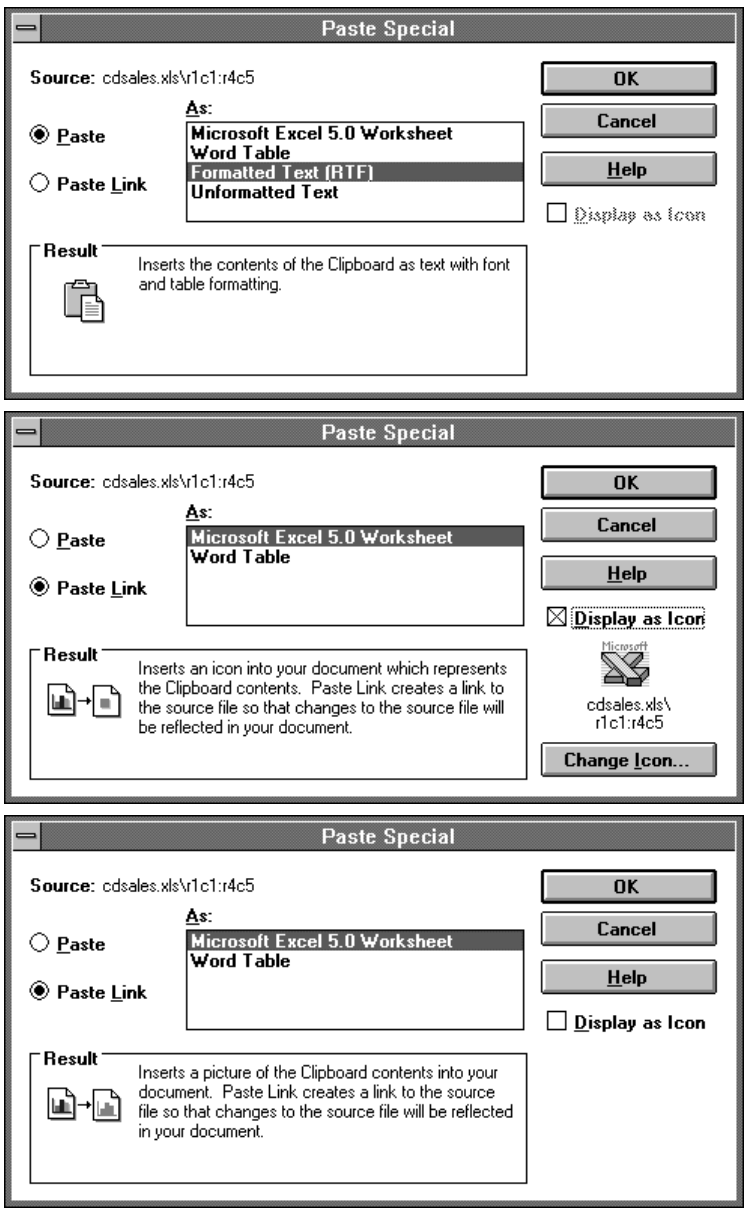


Figure 44. The Paste Special dialog

Notice the use of the result text to indicate the effects of the paste operation. Below is the recommended result text for the Paste Special dialog.

Function	Result Text
Paste.[Linked] Object	Inserts the contents of the Clipboard into your document so you that you may activate it using <*object app name*>.
Paste.[Linked] Object as Icon	Inserts the contents of the Clipboard into your document so you that you may activate it using <*object app name*>. It will be displayed as an icon.
Paste.Data	Inserts the contents of the Clipboard into your document as <*native type name, and optionally an additional help sentence*>.
Paste Link.Object	Inserts a picture of the Clipboard contents into your document. Paste Link creates a link to the source file so that changes to the source file will be reflected in your document.

 © Microsoft Corporation 1992-1993. All Rights Reserved.

Paste Link.Object as Icon	Inserts an icon into your document which represents the Clipboard contents. Paste Link creates a link to the source file so that changes to the source file will be reflected in your document.
Paste Link.Data	Inserts the contents of the Clipboard into your document as *<native type name>*. Paste Link creates a link to the source file so that changes to the source file will be reflected in your document.

Table 3. Paste Special Result Text

2.8.2. Drag/Drop Method

The clipboard method is particularly useful when the user must navigate or edit between choosing the selection and the insertion point, but for short distance transfers the user may prefer to simply *drag* an object and *drop* it into place. Using the drag/drop method a user can directly transfer objects from documents to documents as well as supply objects to system resources such as printers and mailboxes. The steps for performing a drag/drop are:

1. Press the left mouse button down over the object to *grab* it. For objects such as text or table cells, it may be necessary to first select the desired range. Also objects may be dragged by a provided move handle, its frame border, or anywhere within its extent.
2. Holding the left mouse button down, move the object to the desired destination. During the drag motion a pre-indication of the insertion point or potential action is supplied by the destination as feedback to the user. When dragging above text this feedback might be an greyed I-beam tracking the insertion point, jumping between characters; for tables it might be an indication of which cell(s) would accept the drop.
3. Release the left button over the desired point, and the object drops on the destination.

This unmodified drag (no modifier keys) is typically a move operation, but the drop target is free to interpret this as a copy or a link to accomplish the most meaningful or safest default (i.e. dragging across disk volumes might be a copy). In addition the source may constrain the transfer possibilities because of access privileges, and thus some amount of negotiation between source and destination will determine the unmodified drag's function. In addition to transferring data within documents, the drag/drop method provides an efficient way of supplying operands to resource objects like printers and outboxes, or transferring objects between list panes, dialogs, and special palettes. Because of the peculiarity of these destinations they will likely uniquely determine the meaning of a default drag.

For example, if the following cell was selected from the embedded Microsoft Excel worksheet, the user would start the drag by grabbing the selection's border, Microsoft Excel's drag handle rule.

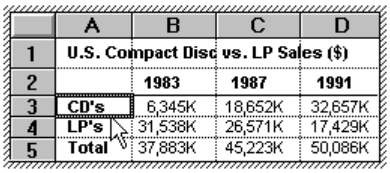


Figure 45. Select and start drag.

The "move" pointer appears by default and the user has a choice of eight different categories of destinations. These are enumerated in Figure 46. The significance of each of the categories in this figure is described below.

1. *Drag Scrolling*. Dragging will by normally pass over pane borders, but if the user would like to scroll to an out of view drop target, (s)he may scroll the pane while dragging the object. By placing the pointer just 11 pixels inside any one of the pane's four boundaries the pane will scroll in direction of that edge (up in the case of Figure 46). The corresponding scrollbar arrow will immediately depress when the pointer is within range, but the actual scrolling will begin after the cursor has been within 11 pixels (on a VGA screen) of the pane edge for an uninterrupted 50 milliseconds, and will continue until the cursor moves outside the 11 pixel region. The cursor arrow is filled with

black immediately and remains so while it is within the 11 pixel hit area. After the document has scrolled to the proper point, the user may drop the object at the desired destination.

2. *Container Area*. An object may be dropped back in the container of the currently active object, Microsoft Word in this case. Notice it shows the dotted I-beam between characters as Microsoft Word's drag feedback.
3. *Same Selection.* Dragging back to the original selection voids the drag/drop operation.
4. *Active Object*. This drag is completely internal to Microsoft Excel and is handled the same as normal dragging in the Microsoft Excel application.
5. *Illegal Destination*. When the destination cannot accept a drop (such as the screen background here), the "prohibited" cursor should be displayed.
6. *System Resource (Icon)*. Here the data is being dropped on a particular printer which highlights with the selection color. Again notice that copy is the default operation for this drag/drop.
7. *Another Document*. Here the drag has left the pane (and application window), and is being dropped into another application, Microsoft Paintbrush. Notice how Paintbrush handles the dropped text. Bear in mind that targets 1-4 are possible in *any* window, not just the active application window. The plus near the arrow head indicates that copy is the default drag function.

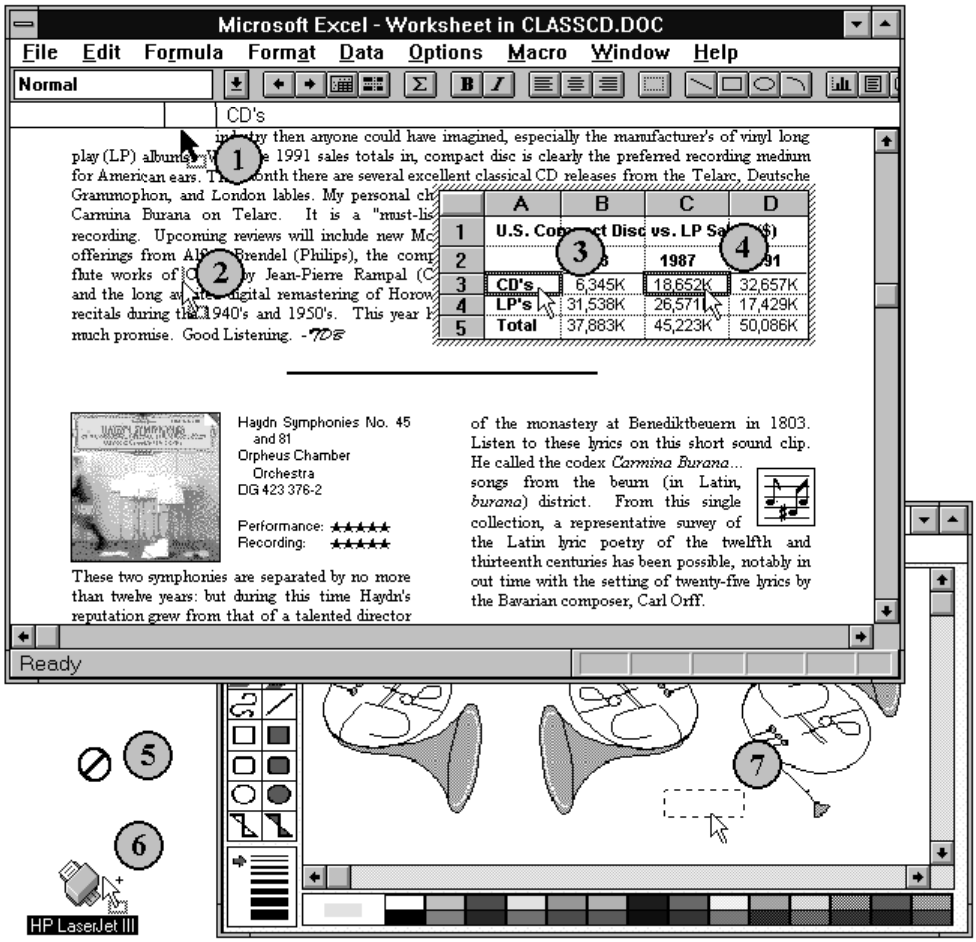


Figure 46. Possible drag/drop destinations.

Unmodified drag should almost always perform a move operation. Only in exceptional cases (like dragging to a printer) should unmodified drag not perform a move. Ctrl-drag should always perform a copy operation. Shift-ctrl drag is the suggested modifier for linking, and Alt-drag is suggested for insisting a move operation (mainly for backwards compatibility with existing applications which use it now). In cases where a drop of a particular object or the insisted operation is not permitted (because of improper

type or context), the "no smoking" sign is given as an indication that a drop will not happen; more generally the cursor's appearance will generally suggest which type of transfer will be performed as feedback to the user.[12]

The drag is qualified by the modifier keys only while they are depressed; releasing the modifiers before the drop will result in a default drop. These drag/drops produce the same end result as Cut/Paste, Copy/Paste, and Copy/Paste Link respectively, but there is no impact on the state of the clipboard since the two transfer methods are completely independent.

(No modifier)	ALT	CTRL	SHIFT+CTRL
determined by destination (recommended move)	force move	force copy	link

Table 4. Drag/drop interpretation with modifier keys.

It is further suggested that the "drop-point" indicator (checkered I-beam) shown in Figure 47 be displayed beneath the cursor as target feedback when dragging over text areas. Since text areas are a very common drop site, it will be particularly necessary for the user to see this consistent feedback.

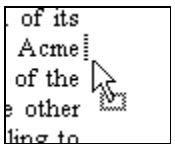


Figure 47. Text drop target feedback.

When dragging an icon from the Windows File Manager into a document, it should effectively perform an "insert object from file," and expose the content of the object if it all possible. Of course if the file cannot be displayed as content (possibly because it is not OLE aware), then it should continue to be displayed as an icon. If the file is drag-linked from File Manager as an icon, its label should be its 8.3 filename (lowercase). Below are the recommend interpretations of the modified drag and drop operations.

2.8.3. OLE 2 and the Packager

OLE 1 introduced the "Packager" which packaged non-OLE files and MS-DOS commands as icons which could then be embedded into other documents. OLE 2 has augmented the Insert Object and Paste Special dialogs to subsume part of the functionality of the Packager (the ability to display objects as icons) so that it is less often needed as a separate utility.

Linking and Embedding a Portion of a File as an Icon.

After cutting or copying a portion of an OLE aware file to the clipboard, one may use Paste Special's new "Icon" format for linking and embedding the data as an icon representation. The verbs of the resulting icon are those defined for its OLE class.

Linking and Embedding Whole Files into documents[13]

Using the File Manager a file can be copied to the clipboard and then embedded or linked into a document via Paste Special. If the file's type does not belong to a registered OLE application, then a Package is created as in OLE 1. If the file, however, belongs to a registered OLE application, then the type name format will also appear in the Paste Special format list enabling the user to link or embed the file in its exposed form.

The From File... command of the Insert Object may also be used to link or embed whole files (see Figure 17). Similarly to Paste Special, inserting a non-OLE file (as determined by its file extension) will force the "Display as Icon" checkbox to be true, and inserting an OLE file has the option of icon or content view.

---

[12] See the *The Windows Interface* for most of these cursor appearances. The "link" cursor which is unspecified in *The Windows Interface* appears as an arrow with an equal sign at the top right. These cursors are also supplied as part of the OLE2.DLL; applications will not normally need to create them on their own.

[13] Technical details regarding the interaction of Win3.1 Drag Drop, inserting files, etc., and OLE2 are found in the later chapter in this specification on "Drag Drop and the Clipboard."

 © Microsoft Corporation 1992-1993. All Rights Reserved.

Dragging and dropping a non-OLE file icon from the File Manager into a document produces the same result as embedding the file in its icon representation labeled with its DOS filename.

2.9. General Dialog and Status Line Messages

If the user attempts to launch an object application (for example, a mini-application) that cannot be run as a stand-alone application, the error message shown in Figure 48 should be issued.

Figure 48. Warning Message When an Object Application Cannot Be Run Stand-Alone

If the container application fails to locate the requested object application when user selects the entry from the Insert Object dialog, or when the user double clicks on an object, the following error message should be displayed. Browse... invokes the standard File Open dialog, and the user supplied path should be entered in the registration as the new object application pathname.

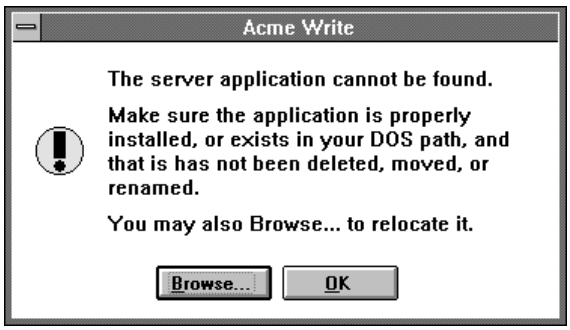


Figure 49. Warning Message When Object Application Can Not Be Found

An object application may be busy or unavailable for several reasons. For example, it may be busy printing, it may be waiting for user input to a error message dialog, or it may be hung. If the object application is not available, the warning message in Figure 50 should be displayed. Details as to when exactly to consider an object application as busy or unavailable are found later in this specification in the chapter on Concurrency Control.

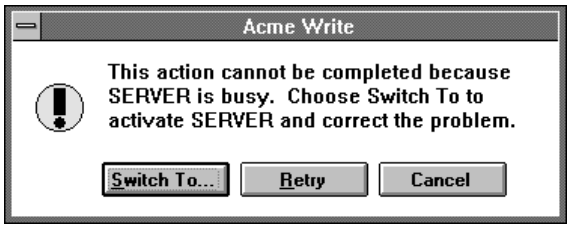


Figure 50. Object application Busy Warning Message

The progress indicator shown in Figure 51 should be displayed while the links are being updated (i.e. when a document that contains automatic links is opened). The Stop button interrupts the update process and prevents any further links from being updated.

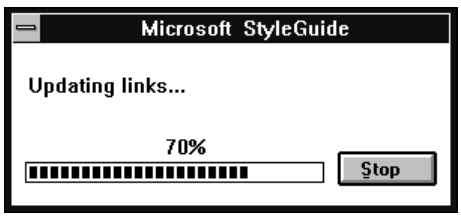


Figure 51. Progress Indicator for Link Updating

If some of the linked files are unavailable, the warning dialog shown in Figure 53 is displayed. This dialog contains two buttons, OK and Links... . The OK button closes the dialog without updating the links. The Links... button displays the Links dialog (see Figure 40) with all the links listed. Unavailable linked files are marked with the word "Unavailable" in the third column of the list. The user can attempt to locate the unavailable files by using the Change Source dialog (see Figure 41), which is available from the Change Source... command button in the Links dialog.

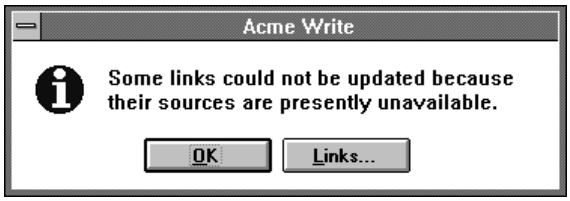


Figure 52. Warning Message for Unavailable Links

If the user issues a command to a link whose source is unavailable, the warning message in Figure 53 should be issued. The unavailable link will be marked as such in the Links dialog.

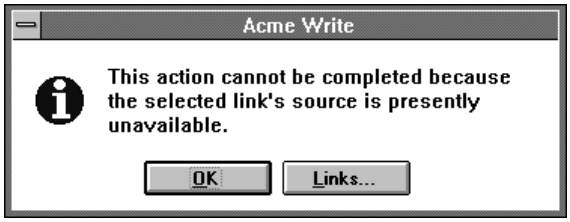


Figure 53. Warning Message for Issuing a Command to an Unavailable Link

Finally, the following table list suggested status line messages for various OLE-related commands.

Menu Commands	**Status Line Message**
File Menu	
Update <*container-document*>	Updates the appearance of this <*full type name*> in <*container-document*>
Save Copy As...	Save a copy of <*descriptive typelass name*> in a separate file
Exit & Return to <*container-document*>	Exit <*object application* > and return to <*container-document*>
Edit Menu	
Paste <*short type name*>	Inserts Clipboard contents as <*full type name*> [14]
Paste Special...	Inserts Clipboard contents as a linked object, embedded object, or other format
Paste Link <*short type name*>	Inserts a link to <*full type name*> Object from <*source-document*>
Insert Object...	Inserts a new object
<*verb*>[15] [Linked] <*short type name*>-	*None*

[14] *<Descriptive Class Name>* is identical to the initially highlighted value in the Paste Special Data Type list. This status line message indicates the data format used to paste clipboard contents.

[Linked] <*short type name*> Object ->	Apply the following commands to <*full type name*> Object
[Linked] <*short type name*> Object -> <*verb*>	*None*
Links...	Allows links to be viewed, updated, opened, or removed.
Options (Preferences) Menu	
Show Objects	Displays the borders around objects (toggle)
Mouse Interface	
When an object is selected	Double-click or press [Alt +] Enter to <*primary-verb*>[16] <*full type name*> Object

Table 5. Status Line Messages

[15] If no verb in the registration database is specified, "Activate" should be used as default. If the container has implemented the Convert dialog, then it is recommended to place the Convert... command on the [Linked] <*shortform*> Object -> submenu beneath the object's verb(s).

[16] Note that the verb that is stored in the registration database will contain & (mnemonic indicator) that need to be stripped out before the verb is displayed on the status line.