

## Selecting a Window Model

Deciding how to present your application's collection of related tasks or processes requires considering a number of design factors: your intended audience and their skill level, the presentation of object or task, effective use of the space on the display, and evolution towards data-centered design.

### Presentation of Object or Task

What an object represents and how it is used and relates to other objects influences how you present its view. Simple objects that are self-contained may not require a primary window, or only require a set of menu commands and a property sheet to edit their properties.

An object with user-accessible content in addition to properties, such as a document, only requires a primary window. The single document window interface can be sufficient when the object's primary presentation or use is as a single unit, even when containing different types. Alternative views can easily be supported with controls that allow the user to change the view. Simple, simultaneous views of the same data can even be supported by splitting the window into panes. The system uses the single document window style of interface for most of the components it includes, such as folders.

MDI, workspaces, workbooks, and projects are more effective when the composition of an object requires multiple views or the nature of the user's tasks requires views of multiple objects. These constructs provide a grouping and focus for a set of specific user activities, within the larger environment of the desktop.

MDI is best suited for viewing homogeneous types. The user cannot mix different objects within the same MDI parent windows unless you supply them as part of the application. On the other hand, you can use MDI to support simultaneous views of different objects.

Use a workbook when you want to optimize quick user navigation of multiple views. A workbook simplifies the task by eliminating the management of child windows, but in doing so, it limits the user's ability to see simultaneous views.

Workspaces and projects provide flexibility for viewing and mixing of objects and their windows. Use a workspace as you would MDI, when you want to clearly segregate the icons and their windows used in a task. Use a project when you do not want to constrain any child windows.

A project provides the greatest flexibility for user placement and arrangement of its windows. It does so, however, at the expense of an increase in complexity because it may be more difficult for a user to differentiate the child window of a project from windows of other applications.

## Display Layout

Consider the requirements for layout of information. For very high resolution displays, the use of menu bars, toolbars, and status bars poses little problem for providing adequate display of the information being viewed in a window. Similarly, the appearance of these common interface elements in each window has little impact on the overall presentation. At VGA resolution, however, this can be an issue. The interface components for a set of windows should not so dominate the user's work area that the user cannot easily view or manipulate their data.

MDI, workspaces, workbooks, and projects all allow some interface components to be shared among multiple views. Within shared elements, it must be clear when a particular interface component applies. Although you can automatically switch the content of those components, consider what functions are common across views or child windows and present them in a consistent way to provide for stability in the interface. For example, if multiple views share a Print toolbar button, present that button in a consistent location. If the button's placement constantly shifts when the user switches the view, the user's efficiency in performing the task may decrease. Note that shared interfaces may make user customization of interface components more complex because you need to indicate whether the customization applies to the current context or across all views.

Regardless of the window model you chose, always consider allowing users to determine which interface components they wish to have displayed. Doing so means that you also need to consider how to make basic functionality available if the user hides a particular component. For example, pop-up menus can often supplement the interface when the user hides the menu bar.

## Data-Centered Design

A single document window interface provides the best support for a simple, data-centered design and may be the easiest for users to learn; MDI supports a more conventional application-centered design. It is best suited to multiple views of the same data or contexts where the application does not represent views of user data. You can use workspaces, workbooks, and projects to provide single document window interfaces while preserving some of the management techniques provided by MDI.

## Combination of Alternatives

Single document window interfaces, MDIs, workspaces, workbooks, and projects are not exclusive design techniques. It may be advantageous to combine these techniques. For example, documents can be presented within a workspace. You can also design workbooks and projects as objects within a workspace. In similar fashion, a project might contain a workbook as one of its objects.





# Integrating with the System



Users appreciate seamless integration between the system and their applications. This chapter covers information about integrating your software with the system and how to extend its features, including using the registry to store information about your application, installing your application, using appropriate naming conventions, and supporting shell features, such as the taskbar, Control Panel, and Recycle Bin.

This chapter is only intended to provide an overview. Details required for some conventions go beyond the scope of this guide. For information about these conventions, see the documentation included in the Microsoft Win32 Software Development Kit (SDK). In addition, some of these conventions and features may not be supported in all releases. For more information about specific releases, see Appendix D, “Supporting Specific Versions of Windows.”

## The Registry

Windows provides a special repository called the *registry* that serves as a central configuration database for user-, application-, and computer-specific information. Although the registry is not intended for direct user access, the information placed in it affects your application’s user interface. Registered information determines the icons,

commands, and other features displayed for files. The registry also makes it easier to manage and support configuration information used by your application and eliminates redundant information stored in different locations.


The registry is a hierarchical structure. Each node in the tree is called a key. Each key can contain subkeys and data entries called values. Key names cannot include a space, backslash (\), or wildcard character (\* or ?). In the **HKEY\_CLASSES\_ROOT** key, names beginning with a period (.) are reserved for special syntax (filename extensions), but you can include a period within a key name. The name of a subkey must be unique with respect to its parent key. Key names are not localized into other languages, although their values may be.


A key can have any number of values. A value entry has three parts: the name of the value, its data type, and the value itself. Value entries larger than 2048 bytes should be stored as files with their filenames stored in the registry.

When the user installs your application, register keys for where application data is stored, for filename extensions, icons, shell commands, OLE registration data, and for any special extensions. To register your application's information, you can create a registration file and use the Registry Editor to merge this file into the system registry. You can also use other utilities that support this function, or use the system-supplied registry functions to access or manipulate registry data.

## Registering Application State Information

Use the registry to store state information for your application. Typically, the data you store here will be information you may have stored in initialization (.INI) files in previous releases of Windows. Create subkeys under the **Software** subkey in the **HKEY\_LOCAL\_MACHINE** and **HKEY\_CURRENT\_USER** keys that include information about your application.

 The example registry entries in this chapter represent only the hierarchical relationship of the keys. For more information about the registry and registry file formats, see the documentation included in the Win32 SDK.

 To use memory most efficiently, the system stores only the registry entries that have been installed and that are required for operation. Applications should never fail to write a registry entry because it is not already installed. To ensure this happens, use registry creation functions when adding an entry.

```

HKEY_LOCAL_MACHINE
  Software
    CompanyName
    ProductName
    Version
...
HKEY_CURRENT_USER
  Software
    CompanyName
    ProductName
    Version

```

Use your application's **HKEY\_LOCAL\_MACHINE** entry as the location to store computer-specific data and the **HKEY\_CURRENT\_USER** entry to store user-specific data. The latter key allows you to store settings to tailor your application for individual users working with the same computer. Under your application's subkey, you can define your own structure for the information. Although the system still supports initialization files for backward compatibility, use the registry wherever possible to store your application's state information instead.

Use these keys to save your application's state whenever appropriate, such as when the user closes its primary window. In most cases, it is best to restore a window to its previous state when the user reopens it.

When the user shuts down the system with your application's window open, you may optionally store information in the registry so that the application's state is restored when the user starts up Windows. (The system does this for folders.) To have your application's state restored, store your window and application state information under its registry entries when the system notifies your application that it is shutting down. Store the state information in your application's entries under **HKEY\_CURRENT\_USER** and add a value name–value pair to the **RunOnce** subkey that corresponds to your application. When the user restarts the system, it runs the command line you supply. Once your application runs, you can use the data you stored to restore its state.



```
HKEY_CURRENT_USER
  Software
    Microsoft
      Windows
        CurrentVersion
          RunOnce application identifier = command line
```


If you have multiple instances open, you can include value name entries for each or consolidate them as a single entry and use command-line switches that are most appropriate for your application. For example, you can include entries like the following.

```
WordPad Document 1 = C:\Program Files\Wordpad.exe Letter to Bill /restore
WordPad Document 2 = C:\Program Files\Wordpad.exe Letter to Paul /restore
Paint = C:\Program Files\Paint.exe Abstract.bmp Cubist.bmp
```

As long as you provide a valid command-line string that your application can process, you can format the entry in a way that best fits your application.

You can also include a **RunOnce** entry under the **HKEY\_LOCAL\_MACHINE** key. When using this entry, however, the system runs the application before starting up. You can use this entry for applications that may need to query the user for information that affects how Windows starts. Just remember that any entry here will affect all users of the computer.

**RunOnce** entries are automatically removed from the registry once the system starts up. Therefore, you need not remove or update the entries, but your application must always save its state when the user shuts down the system. The system also supports a **Run** subkey in both the **HKEY\_CURRENT\_USER** and **HKEY\_LOCAL\_MACHINE** keys. The system runs any value name entries under this subkey after the system starts up, but does not remove those entries from the registry. For example, a virus check program can be installed to run automatically after the system starts up. You can also support this functionality by placing a file or shortcut to a file in the Startup folder. The registry stores the location of the Startup folder, as a value in **HKEY\_CURRENT\_USER \Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders**.

 The system's ability to restore an application's state depends on the availability of the application and its data files. If they have been deleted or the user has logged in over the network where the same files are not available, the system may not be able to restore the state.

## Registering Application Path Information

The system supports “per application” paths. If you register a path, Windows sets the PATH environment variable to be the registered path when it starts your application. You set your application’s path in the **App Paths** subkey under the **HKEY\_LOCAL\_MACHINE** key. Create a new key using your application’s executable filename as its name. Set this key’s Default value to the path of your executable file. The system uses this entry to locate your application if it fails to find it in the current path; for example, if the user chooses the Run command on the Start menu and only includes the filename of the application, or if a shortcut icon doesn’t include a path setting. To identify the location of dynamic-link libraries placed in a separate directory, you can also include another value entry called Path and set its value to the path of your dynamic-link libraries.

```

HKEY_LOCAL_MACHINE
  Software
    Microsoft
      Windows
        CurrentVersion
          App Paths
            Application Executable Filename = path
            Path = path
  
```

The system will automatically update the path and default entries if the user moves or renames the application’s executable file using the system shell user interface.

Register any system-wide shared dynamic-link libraries in a subkey under a **SharedDLLs** subkey of **HKEY\_LOCAL\_MACHINE** key. If the file already exists, increment the entry’s usage count index. For more information about the usage count index, see the section, “Installation,” later in this chapter.

```

HKEY_LOCAL_MACHINE
  Software
    Microsoft
      Windows
        CurrentVersion
          SharedDLLs filename [= usage count index]
  
```



## Registering File Extensions

If your application creates and maintains files, register entries for the file types that you expose directly to users and that you want users to be able to easily differentiate. For every file type you register, include at least two entries: a filename-extension key entry and an application (class) identification key entry.

If you do not register an extension for a file type, it will be displayed with the system's generic file object icon, as shown in Figure 10.1, and its extension will always be displayed. In addition, the user will not be able to double-click the file to open it. (Open With will be the icon's default command.)



Figure 10.1 System-generated icons for unregistered types

### The Filename Extension Key

The filename extension entry maps a filename extension to an application identifier. To register an extension, create a subkey in the **HKEY\_CLASSES\_ROOT** key using the three-letter extension (including a period) and set its value to an application identifier.

#### **HKEY\_CLASSES\_ROOT**

*.ext = ApplicationIdentifier*

For the value of the application identifier (also known as programmatic identifier or Prog ID), use a string that uniquely identifies a given class. This string is used internally by the system and is not exposed directly to users (unless explicitly exported with a special registry utility); therefore, you need not localize this entry.

Avoid assigning multiple extensions to the same application identifier. To ensure that each file type can be distinguished by the user, define each extension such that each has a unique application identifier. If you have utility files that the user does not interact with directly, you should still register an extension (and icon) for them, preferably the same extension so that they can be identified. In addition, mark them with the hidden file attribute.

The system provides no arbitration for applications that use the same extensions. So define unique identifiers and check the registry to avoid writing over and replacing existing extension entries, a practice which may seriously affect the user's existing files. More specifically, avoid registering an extension that conflicts or redefines the common filename extensions used by the system. Examples of these extensions are shown in Table 10.1.

**Table 10.1 Common Filename Extensions Supported by Windows**

<b>Extension</b>	<b>Type description</b>
.386	Windows virtual device driver
.3GR	Screen grabber for MS-DOS-based applications
.ACM	Audio compression manager driver
.ADF	Administration configuration files
.ANI	Animated pointer
.AVI	Video clip
.AWD	FAX viewer document
.AWP	FAX key viewer
.AWS	FAX signature viewer
.BAK	Backed-up file
.BAT	MS-DOS batch file
.BFC	Briefcase
.BIN	Binary data file
.BMP	Picture (Windows bitmap)
.CAB	Windows Setup file
.CAL	Windows Calendar file
.CDA	CD audio track
.CFG	Configuration file

**Chapter 10** Integrating with the System

*(Continued)*

<b>Extension</b>	<b>Type description</b>
CNT	Help contents
COM	MS-DOS – based application
CPD	FAX cover page
CPE	FAX cover page
CPI	International code page
CPL	Control Panel extension
CRD	Windows Cardfile document
CSV	Command-separated data file
CUR	Cursor (pointer)
DAT	System data file
DCX	FAX viewer document
DLL	Application extension (dynamic-link library)
DOC	WordPad document
DOS	MS-DOS file (also extension for NDIS2 net card and protocol drivers)
DRV	Device driver
EXE	Application
FND	Saved search
FON	Font file
FOT	Shortcut to font
GR3	Windows 3.0 screen grabber
GRP	Program group file
HLP	Help file
HT	HyperTerminal™ file
ICM	ICM profile
ICO	Icon
IDF	MIDI instrument definition
INF	Setup information
INI	Initialization file (configuration settings)

*(Continued)*

<b>Extension</b>	<b>Type description</b>
KBD	Keyboard layout
LGO	Windows logo driver
LIB	Static-link library
LNK	Shortcut
LOG	Log file
MCI	MCI command set
MDB	File viewer extension
MID	MIDI sequence
MIF	MIDI instrument file
MMF	Microsoft Mail message file
MMM	Animation
MPD	Mini-port driver
MSG	Microsoft® Exchange mail document
MSN	Microsoft Network home base
NLS	Natural language services driver
PAB	Microsoft Exchange personal address book
PCX	Bitmap picture (PCX format)
PDR	Port driver
PF	ICM profile
PIF	Shortcut to MS-DOS-based application
PPD	PostScript® printer description file
PRT	Printer formatted file (result of Print to File option)
PST	Microsoft Exchange personal information store
PWL	Password list
QIC	Backup set for Microsoft Backup
REC	Windows Recorder file
REG	Application registration file
RLE	Picture (RLE format)
RMI	MIDI sequence

*(Continued)*

<b>Extension</b>	<b>Type description</b>
RTF	Document (rich-text format)
SCR	Screen saver
SET	File set for Microsoft Backup
SHB	Shortcut into a document
SHS	Scrap
SPD	PostScript printer description file
SWP	Virtual memory storage
SYS	System file
TIF	Picture (TIFF® format)
TMP	Temporary file
TRN	Translation file
TSP	Windows telephony service provider
TTF	TrueType® font
TXT	Text document
VBX	Microsoft Visual Basic® control file
VER	Version description file
VXD	Virtual device driver
WAV	Sound wave
WPC	WordPad file converter
WRI	Windows Write document

It is a good idea to investigate extensions commonly used by popular applications so you can avoid creating a new extension that might conflict with them, unless you intend to replace or superset the functionality of those applications.



## The Application Identifier Key

The second registry entry you create for a file type is its class-definition (Prog ID) key. Using the same string as the application identifier you used for the extension's value, create a key, and assign a type name as the value of the key.

### **HKEY\_CLASSES\_ROOT**

*.ext = ApplicationIdentifier*

***ApplicationIdentifier = Type Name***

Under this key, you specify shell and OLE properties of the class. Provide this entry even if you do not have any extra information to place under this key; doing so provides a label for users to identify the file type. In addition, you use this entry to register the icon for the file type.

Define the type name (also known as the *MainUserTypeName*) as the human-readable form of its application identifier or class name. It should convey to the user the object's name, behavior, or capability. A type name can include all of the following elements:

1. *Company Name*  
Communicates product identity.
2. *Application Name*  
Indicates which application is responsible for activating a data object.
3. *Data Type*  
Indicates the basic category of the object (for example, drawing, spreadsheet, or sound). Limit the number of characters to a maximum of 15.
4. *Version*  
When there are multiple versions of the same basic type, for upgrading purposes, you may want to include a version number to distinguish types.

When defining your type name, use title capitalization. The name can include up to a maximum of 40 characters. Use one of the following three recommended forms:


1. *Company Name Application Name [Version] Data Type*  
For example, Microsoft Excel Worksheet.
2. *Company Name-Application Name [Version] Data Type*  
For cases when the company name and application are the same — for example, ExampleWare 2.0 Document.
3. *Company Name Application Name [Version]*  
When the application sufficiently describes the data type — for example, Microsoft Graph.

These type names provide the user with a precise language for referring to objects. Because object type names appear throughout the interface, the user becomes conscious of an object's type and its associated behavior. However, because of their length, you may also want to include a short type name. A *short type name* is the data type portion of the full type name. Applications that support OLE always include a short type name entry in the registry. Use the short type name in drop-down and pop-up menus. For example, a Microsoft® Excel Worksheet is simply referred to as a “Worksheet” in menus.

To provide a short type name, add an **AuxUserType** subkey under the application's registered **CLSID** subkey (which is under the **CLSID** key).

```
HKEY_CLASSES_ROOT
    .ext = ApplicationIdentifier
...
    ApplicationIdentifier = Type Name
        CLSID = {CLSID identifier}
...
    CLSID
        {CLSID identifier}
            AuxUserType
                2 = Short Type Name
```

If a short type name is not available for an object because the string was not registered, use the full type name instead. All controls that display the full type name must allocate enough space for 40 characters in width. By comparison, controls need only accommodate 15 characters when using the short type name.

 For more information about registering type names and other information you should include under the **CLSID** key, see the OLE documentation included in the Win32 SDK.

## Supporting Creation

The system supports the creation of new objects in system containers, such as folders and the desktop. Register information for each file type that you want the system to include. The registered type will appear on the New command that the system includes on menus for the desktop, folders, and the Open and Save As common dialog boxes. This provides a more data-centered design because the user can create a new object without having to locate and run the associated application.

To register a file type for inclusion, create a subkey using the Application Identifier under the extension's subkey in **HKEY\_CLASSES\_ROOT**. Under it, also create the **ShellNew** subkey.

### HKEY\_CLASSES\_ROOT

```
.ext = ApplicationIdentifier
    ApplicationIdentifier
    ShellNew Value Name = Value
```

Assign a value entry to the **ShellNew** subkey with one of the four methods for creating a file with this extension.

Value name	Value	Result
NullFile	“ ”	Creates a new file of this type as a null (empty) file.
Data	<i>binary data</i>	Creates a new file containing the binary data.
FileName	<i>path</i>	Creates a new file by copying the specified file.
Command	<i>filename</i>	Carries out the command. Use this to run your own application code to create a new file (for example, run a wizard).

The system also will automatically provide a unique filename for the new file using the type name you register.

When using a Command value, place your application file (that creates the new file) in the directory that the system uses to store these files. To determine the path for that directory, check the setting for the Templates value in the **Shell Folders** subkey found in **HKEY\_CURRENT\_USER\Software\Microsoft\Windows\CurrentVersion\Explorer**. Then you need only register the filename for the command.

## Registering Icons

The system uses the registry to determine which icon to display for a specific file. You register an icon for every data file type that your application supports and that you want the user to be able to distinguish easily. Create a **DefaultIcon** subkey entry under the application identifier subkey you created and define its value as the filename containing the icon. Typically, you use the application's executable filename and the index of the icon within the file. The index value corresponds to the icon resource within the file. A positive number represents the icon's position in the file. A negative number corresponds to the inverse of the resource ID number of the icon. The icon for your application should always be the first icon resource in your executable file. The system always uses the first icon resource to represent executable files. This means the index value for your data files will be a number greater than 0.

### HKEY\_CLASSES\_ROOT

*ApplicationIdentifier* = Type Name  
**DefaultIcon** = path [,index]

Instead of registering the application's executable file, you can register the name of a dynamic link library file (.DLL), an icon file (.ICO), or bitmap file (.BMP) to supply your data file icons. If an icon does not exist or is not registered, the system supplies an icon derived from the icon of the file type's registered application. If no icon is available for the application, the system supplies a generic icon. These icons do not make your files uniquely identifiable, so design and register icons for both your application and its data file types. Include the following sizes: 16 x 16 pixel (16 color), 32 x 32 pixel (16 color), and 48 x 48 pixel (256 color).

 For more information about designing icons, see Chapter 13, "Visual Design."



## Registering Commands

Many of the commands found on icons, including Send To, Cut, Copy, Paste, Create Shortcut, Delete, Rename, and Properties, are provided by their container — that is, their containing folder or the desktop. But you must provide support for the icon's primary commands, also referred to as verbs, such as Open, Edit, Play, and Print. You can also register additional commands that apply to your file types, such as a What's This? command and even commands for other file types.

To add these commands, in the **HKEY\_CLASSES\_ROOT** key, you register a **shell** subkey and a subkey for each verb, and a **command** subkey for each menu command name.

### HKEY\_CLASSES\_ROOT

```
ApplicationIdentifier = Type Name
  shell [ = default verb [,verb2 [...]]
    verb [ = Menu Command Name]
      command = pathname [parameters]
```

You can also register a DDE command string for a DDE command.

### HKEY\_CLASSES\_ROOT

```
ApplicationIdentifier = Type Name
  shell [ = default verb [,verb2 [...]]
    verb [ = Menu Command Name]
      ddeexec = DDE command string
        Application = DDE Application Name
        Topic = DDE topic name
```

A verb is a language-independent name of the command. Applications may use it to invoke a specific command programmatically. The system defines Open, Print, Find, and Explore as standard verbs and automatically provides menu command names and appropriate access key assignments, localized in each international version of Windows. When you supply verbs other than these, provide menu command names localized for the specific version of Windows on which the application is installed. To assign a menu command name for a verb, make it the default value of the verb subkey.



The menu command names corresponding to the verbs for a file type are displayed to the user, either on a folder's File drop-down menu or pop-up menu for a file's icon. These appear at the top of the menu. You define the order of the menu commands by ordering the verbs in the value of the **shell** key. The first verb becomes the default command in the menu.

By default, capitalization follows how you enter format the menu command name value of the verb subkey. Although the system automatically capitalizes the standard commands (Open, Print, Explore, and Find), you can use the value of the menu command name to format the capitalization differently. Similarly, you use the menu command name value to set the access key for the menu command following normal menu conventions, prefixing the character in the name with an ampersand (&). Otherwise, the system sets the first letter of the command as the access key for that command.

To support user execution of a verb, provide the path for the application or a DDE command string. You can include command-line switches. For paths, include a %1 parameter. This parameter is an operational placeholder for whatever file the user selects.

For example, to register an Analyze command for an application that manages stock market information, the registry entries might look like the following.

```
HKEY_CLASSES_ROOT
  stockfile = Stock Data
    shell = analyze
      analyze = &Analyze
        command = C:\Program Files\Stock Analysis\Stock.exe /A
```

You may have different values for each command. You may assign one application to carry out the Open command and another to carry out the Print command, or use the same application for all commands.

## Enabling Printing

If your file types are printable, include a Print verb entry in the **shell** subkey under **HKEY\_CLASSES\_ROOT**, following the conventions described in the previous section. This will display the Print command on the pop-up menu for the icon and on the File menu of the folder in which the icon resides when the user selects the icon. When the user chooses the Print command, the system uses the registry entry to determine what application to run to print the file.

Also register a Print To registry entry for the file types your application supports. This entry enables dragging and dropping of a file onto a printer icon. Although a Print To command is not displayed on any menu, the printer includes Print Here as the default command on the pop-up menu displayed when the user drag and drops a file on the printer using button 2.

In both cases, print the file, preferably, without opening the application's primary window. One way to do this is to provide a command-line switch that runs the application for handling the printing operation only (for example, WordPad.exe /p). In addition, display some form of user feedback that indicates whether a printing process has been initiated and, if so, its progress. For example, this feedback could be a modeless message box that displays, "Printing page *m* of *n* on *printer name*" and a Cancel button. You may also include a progress indicator control.

## Registering OLE

Applications that support OLE use the registry as the primary means of defining class types, operations, and properties for data types supported by those applications. You store OLE registration information in the **HKEY\_CLASSES\_ROOT** key in subkeys under the **CLSID** subkey and in the class description's (Prog ID) subkey.




For more information about the specific registration entries for OLE, see the OLE documentation included in the Win32 SDK.

## Registering Shell Extensions

Your application can extend the functionality of the operational environment provided by the system, also known as the shell, in a number of ways. A shell extension enhances the system by providing additional ways to manipulate file objects, by simplifying the task of browsing through the file system, or by giving the user easier access to tools that manipulate objects in the file system.

Every shell extension requires a *handler*, special application code (32-bit OLE InProc server implemented as a dynamic-link library) that implements subordinate functions. The types of handlers you can provide include:

- Pop-up (context) menu handlers: these add menu items to the pop-up menu for a particular file type.
- Drag handlers: these allow you to support the OLE data transfer conventions for drag and drop operations of a specific file type.
- Drop handlers: these allow you to carry out some action when the user drops objects on a specific type of file.
- Nondefault drag and drop handlers: these are pop-up menu handlers that the system calls when the user drags and drops an object by using mouse button 2.
- Icon handlers: these can be used to add per-instance icons for file objects or to supply icons for all files of a specific type.
- Property sheet handlers: these add pages to a property sheet that the shell displays for a file object. The pages can be specific to a class of files or to a particular file object.
- Copy-hook handlers: these are called when a folder or printer object is about to be moved, copied, deleted, or renamed by the user. The handler can be used to allow or prevent the operation.

 Support for shell extensions may depend on the version of Windows installed. For more information about specific releases, see Appendix D, "Supporting Specific Versions of Windows."

You register the handler for a shell extension in the **HKEY\_CLASSES\_ROOT** key. The **CLSID** subkey contains a list of class identifier key values such as {00030000-0000-0000-C000-000000000046}. Each class identifier must also be a globally unique identifier.



For more information about creating handlers and class identifiers, see the OLE documentation included in the Win32 SDK.

You must also create a **shellex** subkey under the application's class identification entry in the **HKEY\_CLASSES\_ROOT** key.

```
HKEY_CLASSES_ROOT
  ApplicationIdentifier = Type Name
  Shell [ = default verb [, verb2 [...]]
  ...
  shellex
    HandlerType
      {CLSID identifier} = Handler Name
    ...
    HandlerType = {CLSID identifier}
```

The shell also uses several other special keys, such as **\***, **Folder**, **Drives**, and **Printers**, under **HKEY\_CLASSES\_ROOT**. You can use these keys to register extensions for system-supplied objects. For example, you may use the **\*** key to register handlers that the shell calls whenever it creates a pop-up menu or property sheet for a file object, as in the following example.

```
HKEY_CLASSES_ROOT
  * = *
  shellex
    ContextMenuHandlers
      {00000000-1111-2222-3333-0000000001}
    PropertySheetHandlers = SummaryInfo
      {00000000-1111-2222-3333-0000000002}
    IconHandler = {00000000-1111-2222-3333-0000000003}
```

The shell would use these handlers to add to the pop-up menus and property sheets of every file object. (The entries are intended only as examples, not literal entries.)



A pop-up menu handler may add commands to the pop-up menu of a file type, but it may not delete or modify existing menu commands. You can register multiple pop-up menu handlers for a file type. The order of the subkey entries determines the order of the items in the context menu. Handler-supplied menu items always follow registered command names.

Keep in mind that if you want to include a command on the pop-up menu of every file of a particular type, you do not need to create and register a pop-up menu handler. You can just use the normal means of registering commands for that type. Create a pop-up menu handler only when you want to provide a command only under specific conditions, such as the length of the file or its timestamp.


When registering an icon handler for providing per-instance icons for a file type, set the value for the **DefaultIcon** key to %1. This denotes that each file instance of this type can have a different icon.

## Supporting the Quick View Command

The system includes support for fast, read-only views of many file types when the user chooses the Quick View command from the file object's menu. This allows the user to view files without opening the application.

If your file type is not supported, you can install a file parser that translates your file type into a format the system file viewer can read. Although this approach allows you to easily support viewers for your data file types, it limits the interaction options for your file types to those provided by the system. Alternatively, you can create your own file viewer, using the system-supplied interfaces. You can also register a file viewer for a file type already registered.

You can also support the Quick View command for data objects stored within your application's interface, either by supplying a specific viewer for your data types or by writing the data to a temporary file and then executing a file viewer and passing the temporary file as a parameter.

 For more information about supporting the Quick View command and creating file viewers, see the documentation included in the Win32 SDK.



## Registering Sound Events

Your application can register specific events to which the user can assign sound files so that when those events are triggered, the assigned sound file is played. To register a sound event, create a key under the **HKEY\_CURRENT\_USER** key.

```
HKEY_CURRENT_USER  
  AppEvents  
    Event Labels  
      EventName = Event Name
```

Set the value for EventName to a human-readable name.

Registering a sound event only makes it available in Control Panel so the user can assign a sound file. Your application must provide the code to process that event.

## Installation

The following sections provide guidelines for installing your application's files. Applying these guidelines will help you reduce the clutter of irrelevant files when the user browses for a file. In addition, you'll reduce the redundancy of common files and make it easier for the user to update applications or the system software.

### Copying Files

When the user installs your software, avoid copying files into the Windows directory (folder) or its System subdirectory. Doing so clutters the directory and may degrade system performance. Instead, create a single directory, preferably using the application's name, in the Program Files directory (or the location that the user chooses). In this directory, place the executable file. For example, if a program is named My Application, create a My Application subdirectory and place My Application.exe in that directory.

To locate the Program Files directory, check the `ProgramFilesDir` value in the **CurrentVersion** subkey under **HKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows**. The actual directory may not literally be named Program Files. For example, in international versions of Microsoft Windows, the directory name is appropriately localized. For networks that do not support the Windows long filename conventions, MS-DOS names may be used instead.

In your application's directory, create a subdirectory named **System** and place all support files that the user does not directly access in it, such as dynamic-link libraries and Help files. For example, place a support file called `My Application.dll` in the subdirectory `Program Files\My Application\System`. Hide the support files and your application's **System** directory and register its location using a **Path** value in the **App Paths** subkey under **HKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows\CurrentVersion**. Although you may place support files in the same directory as your application, placing them in a subdirectory helps avoid confusing the user and makes files easier to manage.

Applications can share common support files to reduce the amount of disk space consumed by duplication. If some non-user-accessed files of your application are shared as systemwide components (such as Visual Basic's `Vbrun300.dll`), place them in the **System** subdirectory of the directory where the user installs Windows. The process for installing shared files includes these logical steps:

1. Before copying the file, verify that it is not already present.
2. If the file is already present, compare its date and size to determine whether it is the same version as the one you are installing. If it is, increment the usage count in its corresponding registry entry.
3. If the file you are installing is not more recent, do not overwrite the existing version.
4. If the file is not present, copy it to the directory.

If you store a new file in the **System** directory installed by Windows, register a corresponding entry in the **SharedDLL** subkey under the **HKEY\_LOCAL\_MACHINE** key.



The system provides support services in `Ver.dll` for assisting you to do version verification. For more information about this utility, see the documentation included in the Win32 SDK.

If a file is shared, but only among your applications, create a subdirectory using your application's name in the Common Files subdirectory of the Program Files subdirectory and place the file there. To locate the Common Files directory, check the CommonFilesDir value in the **CurrentVersion** subkey of **HKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows**. Alternatively, for "suite" style when multiple applications are bundled together, you can create a suite subdirectory in Program Files, where you place your executable files, and within that a System subdirectory with the support files shared only within the suite. In either case, register the path using the **Path** subkey under the **App Paths** subkey

When installing an updated version of the shared file, ensure that it is upwardly compatible before replacing the existing file. Alternatively, you can create a separate entry with a different filename (for example, Vbrun301.dll).

Name your executable file, dynamic-link libraries, and any other files that the user does not directly use, but that may be shared on a network, using conventional MS-DOS (8.3) names rather than long filenames. This will provide better support for users operating in environments where these files may need to be installed on network services that do not support the Windows long filename conventions.

Windows no longer requires Autoexec.bat and Config.sys files. Ensure that your application also does not require these files. Consider converting any MS-DOS device drivers to Windows virtual device drivers. The system supports dynamic loading of this type of device drivers, unlike MS-DOS device drivers which need to be loaded through Config.sys when starting the system. Similarly, because the registry allows you to register your application paths, your application does not require path information in Autoexec.bat.

In addition, do not make entries in Win.ini. Storing information in this file can make it difficult for the user to update or move your application. Also, avoid maintaining your application's own initialization file. Instead, use the registry. The registry provides conventions for storing most application and user settings. The registry provides greater flexibility allowing you to store information on a per machine or per user basis. It also supports accessing this information across a network.




Make certain you register the types supported by your application and the icons for these types along with your application's icons. In addition, register other application information, such as information required to enable printing.

## Providing Access to Your Application

To provide easy user access to your application, place a shortcut icon to the application in the Programs folder. You can determine the path for this folder in **Shell Folders** subkey under **HKEY\_CURRENT\_USER\Software\Microsoft\Windows\CurrentVersion\Explorer**. This adds the entry to the submenu of the Programs menu of the Start button. Avoid adding entries for every application you might include in your software; this quickly overloads the menu. Optionally, you can allow the user to choose which icons to place in the menu. Avoid using a folder as your entry in the Programs menu, because this creates a multilevel hierarchy. Including a single entry makes it easier and simpler for a user to access your application.

Also consider the layout of files you provide with your application. Folders in Windows 95 and later releases provide much greater flexibility for file organization than did the Windows Program Manager. In addition to the recommended structure for your main executable file and its support files, you may want to create special folders for documents, templates, conversion tools, or other files that the user accesses directly.

 You can create a “program group” entry in the Programs folder using the Windows 3.1 dynamic data exchange application programming interface (API). However, it is not recommended for applications installed with Microsoft Windows 95 and later releases, configured with the new shell user interface.

## Designing Your Installation Program

Your installation program should offer the user different installation options such as:

- **Typical Setup:** installation that proceeds with the common defaults set, copying only the most common files. Make this the default setup option.
- **Compact Setup:** installation of the minimum files necessary to operate your application. This option is best for situations where disk space must be conserved — for example, on laptop computers. You can optionally add a Portable setup option for additional functionality designed especially for configurations on laptops, portables, and portables used with docking stations.


- Custom Setup: installation for the experienced user. This option allows the user to choose where to copy files and which options or features to include. This can include options or components not available for compact or typical setup.
- CD-ROM Setup: installation from a CD-ROM. This option allows users to select what files to install from the CD and allows them to run the remaining files directly from the CD.
- Silent Setup: installation using a command-line switch. This allows your setup program to run with a batch file

In addition to these setup options, your installation program should be a well-designed, Windows-based application and follow the conventions detailed in this guide and in the following guidelines:

- Supply a common response to every option so that the user can step through the installation process by confirming the default settings (that is, by pressing the ENTER key).
- Tell users how much disk space they will need before proceeding with installation. In the custom setup option, adjust the figure as the user chooses to include or exclude certain options. If there is not sufficient disk space, let the user know, but also give the user the option to override.
- Offer the user the option to quit the installation before it is finished. Keep a log of the files copied and the settings made so the canceled installation can be cleaned up easily.
- Ask the user to insert a disk only once during the installation. Lay out your files on disk so that the user does not have to reinsert the same disk multiple times.
- Provide a visual prompt and an audio cue when the user needs to insert the next disk.
- Support installation from any location. Do not assume that installation must be done from a logical MS-DOS drive (such as drive A). Design your installation program to support any valid universal naming convention (UNC) path.
- Provide a progress indicator message box to inform the user how far they are through the installation process.



If you are creating your own installation program, consider using the wizard control. Using this control and following the guidelines for wizards will result in a consistent interface for users.

 For more information about designing wizards, see Chapter 12, “User Assistance.”


Naming your installation program Setup.exe or Install.exe (or localized equivalent) will allow the system to recognize the file. Place the file in the root directory of the disk the user inserts. This allows the system to automatically run your installation program when the user chooses the Install button in the Add/Remove Programs utility in Control Panel.

## Installing Fonts

When installing fonts with your application on a local system, determine whether the font is already present. If it is, rename your font file — for example, by appending a number to the end of its filename. After copying a font file, register the font in the **Fonts** subkey.

## Installing Your Application on a Network

If you create a client-server application so that multiple users access it from a network server, create separate installation programs: an installation program that allows the network administrator to prepare the server component of the application, and a client installation program that installs the client component files and sets up the settings to connect to the server. Design your client software so that an administrator can deploy it over the network and have it automatically configure itself when the user starts it.

 For more information about designing client-server applications, see Chapter 14, “Special Design Considerations.” Additional information can also be found in the documentation included in Win32 SDK.

Because Windows may itself be configured to be shared on a server, do not assume that your installation program can store information in the main Windows directory on the server. In addition, shared application files should not be stored in the “home” directory provided for the user.

Design your installation program to support UNC paths. Also, use UNC paths for any shortcut icons you install in the Start Menu folder.

## Uninstalling Your Application

The user may need to remove your application to recover disk space or to move the application to another location. To facilitate this, provide an uninstall program with your application that removes its files and settings. Remember to remove registry entries and shortcuts your application may have placed in the Start menu hierarchy. However, be careful when removing your application's directory structure not to delete any user files (unless you confirm their removal with the user).

Your uninstall program should follow the conventions detailed in this guide and in the following guidelines:

- Display a window that provides the user with information about the progress of the uninstall process. You can also provide an option to allow the program to uninstall “silently” — that is, without displaying any information so that it can be used in batch files.
- Display clear and helpful messages for any errors your uninstall program encounters during the uninstall process.
- When uninstalling an application, decrement the usage count in the registry for any shared component — for example, a dynamic-link library. If the result is zero, give the user the option to delete the shared component with the warning that other applications may use this file and will not work if it is missing.

Registering your uninstall program will display your application in the list of the Uninstall page of the Add/Remove Program utility included with Windows. To register your uninstall program, add entries for your application to the **Uninstall** subkey.

```

HKEY_LOCAL_MACHINE
  Software
    Microsoft
      Windows
        CurrentVersion
          Uninstall
            ApplicationName DisplayName = Application Name
            UninstallString = path [ switches ]
  
```

Both the **DisplayName** and **UninstallString** values must be supplied and be complete for your uninstall program to appear in the Add/Remove Program utility. The path you supply to **UninstallString** must be the complete command line used to carry out your uninstall program. The command line you supply should carry out the uninstall program directly rather than from a batch file or subprocess.

## Supporting AutoPlay

Windows supports the ability to automatically run a file when the user inserts removable media that support insertion notification, such as CD-ROM, PCMCIA hard disks, or flash ROM cards. To support this feature, include a file named `Autorun.inf` in the root directory of the removable media. In this file, include the filename of the file to run, using the following syntax.

```
[autorun]
open = filename
```

Unless you specify a path, the system looks for the file in the root of the inserted media. If you want to run a file located in a subdirectory, include a path relative to the root; include that path with the file as in the following example.

```
open = My Directory\My File.exe
```

Running the file from a subdirectory does not change the current directory setting. The command-line string you supply can also include parameters or switches.

Because the autoplay feature is intended to provide automatic operation, design the file you specify in the `Autorun.inf` file to provide visual feedback quickly to confirm the successful insertion of the media. Consider using a startup up window with a graphic or animated sequence. If the process you are automating requires a long load time or requires user input, offer the user the option to cancel the process.

Although you can use this feature to install an application, avoid writing files to the user's local disk without the user's confirmation. Even when you get the user's confirmation, minimize the file storage requirements, particularly for CD-ROM games or educational applications. Consuming a large amount of local file space defeats some of the benefits of the turnkey operation that the autoplay feature provides. Also, because a network administrator or the user can disable this feature, avoid depending on it for any required operations.

You can define the icon that the system displays for the media by including an entry in the Autorun.inf file that includes the filename (and optionally the path) including the icon using the following form.

```
icon = filename
```

The filename can specify an icon, a bitmap, an executable, or a dynamic-link library file. If the file contains more than one icon resource, specify the resource with a number after the filename — for example, My File.exe, 1. The numbering follows the same conventions as the registry. The default path for the file will be relative to the Autorun.inf file. If you want to specify an absolute path for an icon, use the following form.

```
defaulticon = path
```

The system automatically provides a pop-up menu for the icon and includes AutoPlay as the default command on that menu, so that double-clicking the icon will run the Open = line. You can include additional commands on the menu for the icon by adding entries for them in the Autorun.inf file, using the following form.

```
shell\verb\command = filename  
shell\verb = Menu Item Name
```

To define an access key assignment for the command, precede the character with an ampersand (&). For example, to add the command Read Me First to the menu of the icon, include the following in the Autorun.inf file.

```
shell\readme\command = Notepad.exe My Directory\Readme.txt  
shell\readme = Read &Me First
```



Although AutoPlay is typically the default menu item, you can define a different command to be the default by including the following line.

```
shell = verb
```

When the user double-clicks on the icon, the command associated with this entry will be carried out.

## System Naming Conventions


Windows provides support for filenames up to 255 characters long. Use the long filename when displaying the name of a file. Avoid displaying the filename extension unless the user chooses the option to display extensions or when the file type is not registered.

Because the system uses three-letter extensions to describe a file type, do not use extensions to distinguish different forms of the same file type. For example, if your application has a function that automatically backs up a file, name the backup file Backup of *filename.ext* (using its existing extension) or some reasonable equivalent, not *filename.bak*. The latter implies a change of the file's type. Similarly, do not use a Windows filename extension unless your file fits the type description.

Long filenames can include any character, except the following.

```
\\ : * ? < > | “
```

When your application automatically supplies a filename, use a name that communicates information about its creation. For example, files created by a particular application should use either the application-supplied type name or the short type name as a proposed name — for example, worksheet or document. When that file exists already in the target directory, add a number to the end of the proposed name — for example, Document (2). When adding numbers to the end of a proposed filename, use the first number of an ordinal sequence that does not conflict with an existing name in that directory.

 The system automatically formats a filename correctly if you use the **SHGetFileInfo** or **GetFileName** function. For more information about these functions, see the documentation included in the Win32 SDK.

When saving a file, make certain you preserve the creation date of the file. For simple applications that open and save a file, this happens automatically. However, more sophisticated applications may create temporary files, delete the original file, and rename the temporary file to the original filename. In this case, the application needs to copy the creation date as well from the old file to the new, using the standard system functions. Certain system file management functionality may depend on the correct creation date.

When you create a filename, the system automatically creates an MS-DOS filename (alias) for a file. The system displays both the long filename and the MS-DOS filename in the property sheet for the file.

When a file is copied, use the words “Copy of” as part of the generated filename — for example, “Copy of Sample” for a file named “Sample.” If the prefix “Copy of” is already assigned to a file, include a number in parentheses — for example, “Copy (2) of Sample”. You can apply the same naming scheme to links, except the prefix is “Link to” or “Shortcut to.”

It is also important to support UNC paths for identifying the location of files and folders. UNC paths and filenames have the following form.

```
\\Server\Share\Directory\Filename.ext
```

Using UNC names enables the user to directly browse the network and open files without having to make explicit network connections.

Wherever possible, display the full name of a file (without the extension). The number of characters you’ll be able to display depends somewhat on the font used and the context in which the name is displayed. In any case, supply enough characters such that the user can reasonably distinguish between names. Take into account common prefixes such as “Copy of” or “Shortcut to”. If you don’t display the full name, indicate that it has been truncated by appending an ellipsis to the end of the name.

You can use an ellipsis to abbreviate path names, in a displayable, but noneditable situation. In this case, include at least the first two entries of the beginning and the end of the path, using ellipses as notation for the names in between, as in the following example.

```
\\My Server\My Share\...\My Folder\My File
```

When using an icon to represent a network resource, label the icon with the name of the resource. If you need to show the network context rather than using a UNC path, label the resource using the following format.

*Resource Name on Computer Name*

## Taskbar Integration

The system provides support for integrating your application's interface with the taskbar. The following sections provide information on some of the capabilities and appropriate guidelines.

### Taskbar Window Buttons

When an application creates a primary window, the system automatically adds a taskbar button for that window and removes it when that window closes. For some specialized types of applications that run in the background, a primary window may not be necessary. In such cases, make certain you provide reasonable support for controlling the application using the commands available on the application's icon; it should not appear as an entry on the taskbar, however. Similarly, the secondary windows of an application should also not appear as a taskbar button.

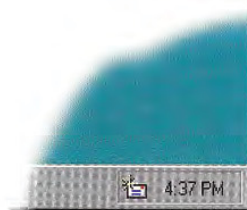
The taskbar window buttons support drag and drop, but not in the conventional way. When the user drags an object over a taskbar window button, the system automatically restores the window. The user can then drop the object in the window.



## Status Notification

The system allows you to add status or notification information to the taskbar. Because the taskbar is a shared resource, add information to it that is of a global nature only or that needs monitoring by the user while working with other applications.


Present status notification information in the form of a graphic supplied by your application, as shown in Figure 10.2.



**Figure 10.2** Status indicator in the taskbar

When adding a status indicator to the taskbar, also support the following interactions:

- Provide a pop-up window that displays further information or controls for the object represented by the status indicator when the user clicks with button 1. For example, the audio (speaker) status indicator displays a volume control. Use a pop-up window to supply for further information rather than a dialog box, because the user can dismiss the window by clicking elsewhere. Position the pop-up window near the status indicator so the user can navigate to it quickly and easily. Avoid displaying other types of secondary windows because they require explicit user interaction to dismiss them. If there is no information or control that applies, do not display anything.
- Display a pop-up menu for the object represented by the status indicator when the user clicks on the status indicator with button 2. On this menu, include commands that bring up property sheets or other windows related to the status indicator. For example, the audio status indicator provides commands that display the audio properties as well as the Volume Control mixer application.
- Carry out the default command defined in the pop-up menu for the status indicator when the user double-clicks.

 The **Shell\_NotifyIcon** function provides support for adding a status item in the taskbar. For more information about this function, see the documentation included in the Win32 SDK.



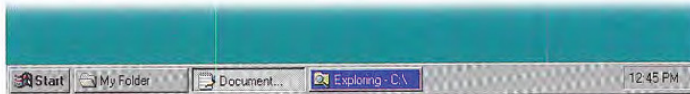
- Display a tooltip that indicates what the status indicator represents. For example, this could include the name of the indicator, a value, or both.
- Provide the user an option to not display the status indicator, preferably in the property sheet for the object displaying the status indicator. This allows the user to determine which indicator to include in this shared space. You may need to provide an alternate means of conveying this status information when the user turns off the status indicator.

## Message Notification

When your application's window is inactive but must display a message, rather than displaying a message box on top of the currently active window and switching the input focus, flash your application's title bar and taskbar window button to notify the user of the pending message. This avoids interfering with the user's current activity but lets the user know a message is waiting. When the user activates your application's window, the application can display a message box.


Use the system setting for the cursor blink rate for your flash rate. This allows the user to control the flash rate to a comfortable frequency.


Rather than flashing the button continually, you can flash the window button only a limited number of times (for example, three), then leave the button in the highlighted state, as shown in Figure 10.3. This lets the user know there is still a pending message.



**Figure 10.3** Flashing a taskbar button to notify a user of a pending message

This cooperative means of notification is preferable unless a message relates to the system integrity of the user's data, in which case your application may immediately display a system modal message box. In such cases, flush the input queue so that the user does not inadvertently select a choice in that message box.

 The **FlashWindow** function supports flashing your title bar and taskbar window button. For more information about this function, see the documentation included in the Win32 SDK.

 The **GetCaretBlinkTime** function provides access to the current cursor blink rate setting. For more information about this function, see the documentation included in the Win32 SDK.

## Application Desktop Toolbars

The system supports applications supplying their own desktop toolbars, also referred to as access bars or appbars, that operate similarly to the Windows taskbar. These may be docked to the edges of a screen and provide access to controls, such as buttons, for specific functions.

The system supports the same auto-hide behavior for application desktop toolbars as it does for the taskbar. This allows the desktop toolbar to only be visible when the user moves the pointer to the edge of the screen. The system also provides the “always on top” behavior used by the taskbar. When the user sets this property, the taskbar always appears on top (in the Z order) of any windows and also acts as a boundary for windows set to maximize to the display screen size.

Desktop toolbars can also be undocked and displayed as a palette window or redocked at a different edge of the screen. In the undocked, displayed as a palette window state, the toolbar no longer constrains other windows. However, if it supports the Always on Top property, it remains on top of other application windows.

Before designing a desktop toolbar, consider whether your application’s tasks really require one. Remember that a desktop toolbar will potentially affect the visible area for all applications. Only provide one for frequently used interfaces that can be applied across applications and always design it to be an optional interface, allowing the user to close it or otherwise configure it not to appear. You may also want to consider removing it when a specific application or applications are closed.

When creating your own desktop toolbar, model its behavior on the taskbar. Consider using the system’s notification of when the taskbar’s auto-hide or Always on Top property changes to apply a desktop toolbar you provide. If this does not fit your design, be certain to provide your own property sheet for setting these attributes for your desktop toolbar. Note that the system only supports auto-hide functionality for one desktop toolbar on each edge of the display. In addition, always provide a pop-up menu to access commands that apply to your desktop toolbar, such as Close, Move, Size, and Properties (but not the commands included on the desktop toolbar).



For more information on the recommended behavior for undocking and redocking toolbars, see Chapter 7, “Menus, Controls, and Toolbars.”

You can choose to display a desktop toolbar when the user runs a specific application, or by creating a separate application and including a shortcut icon to it in the system's Startup folder. Preferably set the initial size and position of your desktop toolbar so that it does not interfere with other desktop toolbars or the taskbar. However, the system does support multiple desktop toolbars to be docked along the same edge of the display screen. When docking on the same edge as the taskbar, the system places the taskbar on the outermost edge.

Your desktop toolbar can include any type of control. A desktop toolbar can also be a drag and drop target. Follow the recommendations outlined in this guide for supporting appropriate interaction.

## Full-Screen Display

Although the taskbar and application desktop toolbars normally constrain or clip windows displayed on the screen, you can define a window to the full extent of the display screen. Because this is not the typical form of interaction, only consider using full-screen display for very special circumstances, such as a slide presentation, and only when the user explicitly chooses a command for this purpose. Make certain you provide an easy way for the user to return to normal display viewing. For example, you can display an on-screen button when the user moves the pointer that restores the display when the user clicks it. In addition, keyboard interfaces, like ALT+TAB and ESC, should automatically restore the display.

Remember that desktop toolbars, including the taskbar, should support auto-hide options that allow the user to configure them to reduce their visual impact on the screen. Consider whether this auto-hide capability may be sufficient before designing your application to require a full-screen presentation. Advising the user to close or hide desktop toolbars may provide you with sufficient space without having to use the full display screen.



## Recycle Bin Integration



The Recycle Bin provides a repository for deleted files. If your application includes a facility for deleting files, support the Recycle Bin interface. You can also support deletion to the Recycle Bin for nonfile objects by first formatting the deleted data as a file by writing it to a temporary file and then calling the system functions that support the Recycle Bin.



The **SHFileOperation** function supports deletion using the Recycle Bin interface. For more information about this function, see the documentation included in the Win32 SDK.

## Control Panel Integration



The Windows Control Panel includes special objects that let users configure aspects of the system. Your application can add Control Panel objects or add property pages to the property sheets of existing Control Panel objects.

### Adding Control Panel Objects

You can create your own Control Panel objects. Most Control Panel objects supply only a single secondary window, typically a property sheet. Define your Control Panel object to represent a concrete object rather than an abstract idea.

Every Control Panel object is a dynamic-link library. To ensure that the dynamic-link library can be automatically loaded by the system, set the file's extension to .CPL and install it in the Windows System directory.



The system automatically caches information about Control Panel objects in order to provide quick user access, provided that the Control Panel object supports the correct system interfaces. For more information about developing Control Panel objects, see the documentation included in the Win32 SDK.

### Adding to the Passwords Object

The Passwords object in Control Panel supplies a property sheet that allows the user to set security options and manage passwords for all password-protected services in the system. The Passwords object also allows you to add the name of a password-protected service to the object's list of services and use the Windows login password for all password-protected services in the system.




When you add your service to the Passwords object, the name of the service appears in the Select Password dialog box that appears when the user chooses Change Other Passwords. The user can then change the password for the service by selecting the name and filling in the resulting dialog box. The name of your service also appears in the Change Windows Password dialog box; the name appears with a check box next to it. By setting the check box option, the user chooses to keep the password for the service identical to the Windows login password. Similarly, the user can disassociate the service from the Windows login password by toggling the check box setting off.

To add your service to the Passwords object, register your service under the **HKEY\_LOCAL\_MACHINE** key.

```
HKEY_LOCAL_MACHINE  
  System  
    CurrentControlSet  
      Control  
        PwdProvider  
          Provider Name Value Name = Value
```

You can also add a page to the property sheet of the Passwords object to support other security-related services that the user can set as property values. Add a property page if your application provides security-related functionality beyond simple activation and changing of passwords. To add a property page, follow the conventions for adding shell extensions.

 For more information about registering your password service, see the documentation included in the Win32 SDK.

## Plug and Play Support

Plug and Play is a feature of Windows that, with little or no user intervention, automatically installs and configures drivers when their corresponding hardware peripherals are plugged into a PC. This feature applies to peripherals designed according to the Plug and Play specification. Supporting and appropriately adapting to Plug and Play hardware change can make your application easier to use. Following are some examples of supporting Plug and Play:

- Resizing your windows and toolbars relevant to screen size changes.
- Prompting users to shut down and save their data when the system issues a low power warning.
- Warning users about open network files when undocking their computers.
- Saving and closing files appropriately when users eject or remove removable media or storage devices or when network connections are broken.

## System Settings and Notification

The system provides standard metrics and settings for user interface aspects, such as colors, fonts, border width, and drag rectangle (used to detect the start of a drag operation). The system also notifies running applications when its settings change. When your application starts up, query the system to set your application's user interface to match the system parameters to ensure visual and operational consistency. Also, design your application to adjust itself appropriately when the system notifies it of changes to these settings.



The **GetSystemMetrics**, **GetSysColor**, and **SystemParametersInfo** functions and the **WM\_SETTINGSCCHANGE** message are important to consider when supporting standard system settings. For more information about these system interfaces, see the documentation included in the Win32 SDK.

## Modeless Interaction

When designing your application, try to ensure that it is as interactive and nonmodal as possible. Here are some suggested ways of doing this:

- Use modeless secondary windows wherever possible.
- Segment processes, like printing, so you do not need to load the entire application to perform the operation.
- Make long processes run in the background, keeping the foreground interactive. For example, when something is printing, it should be possible to minimize the window even if the document cannot be altered. The multitasking support of Windows provides for defining separate processes, or *threads*, in the background.



For more information about threads, see the documentation included in the Win32 SDK.

# Working with OLE Embedded and OLE Linked Objects



Microsoft OLE provides a set of system interfaces that enables users to combine objects supported by different applications. This chapter outlines guidelines for the interface for OLE embedded and OLE linked objects; you can apply many of these guidelines to any implementation of containers and their components.

## The Interaction Model

As data becomes the major focus of interface design, its content is what occupies the user's attention, not the application managing it. In such a design, data is not limited to its native creation and editing environment; that is, the user is not limited to creating or editing data only within its associated application window. Instead, data can be transferred to other types of containers while maintaining its viewing and editing capability in the new container. Compound documents are a common example and illustration of the interaction between containers and their components, but they are not the only expression of this kind of object relationship that OLE can support.

Figure 11.1 shows an example of a compound document. The document includes word-processing text, tabular data from a spreadsheet, a sound recording, and pictures created in other applications.





# Classical CD Review

by Thomas D. Becker

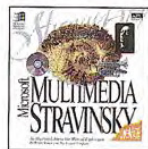
The introduction of the compact disc has had a far greater impact on the recording industry than anyone could have imagined, especially the manufacturers of vinyl long play (LP) albums. With the 1991 sales totals in, compact disc is clearly the preferred recording medium for American ears. In addition to audio compact discs, CD-ROMs are appearing on the market offering a multimedia experience of the classical repertoire. The Microsoft Composer Collection brings you the ability to enter the lives and minds of three astounding musical geniuses. That's because the Composer Collection contains three CD-ROM titles full of music, information, and entertainment. They are: Microsoft Multimedia Mozart, Microsoft Multimedia Stravinsky, and Microsoft Multimedia Beethoven. These works are reviewed below — be sure to check them out! —TDB

U.S. Compact Disc vs LP Sales (\$)			
	1983	1987	1991
CDs	6,345K	18,652K	32,657K
LPs	31,538K	26,571K	17,429K
<b>Total</b>	<b>37,883K</b>	<b>45,223K</b>	<b>50,086K</b>



Multimedia Mozart: The Dissonant Quartet  
The Voyager Company  
Microsoft

In the words of author and music scholar Robert Winter, the string quartet in the eighteenth century was regarded as one of the "most sublime forms of communication." The String Quartet in C Major is no exception. Discover the power and the beauty of this music with Microsoft Multimedia Mozart: *The Dissonant Quartet*, and enter the world in which Mozart created his most memorable masterpieces. Sit back and enjoy *The Dissonant Quartet* in its entirety, or browse around, exploring its themes and emotional dynamics in depth. View the entire piece in a single-screen overview with the *Pocket Audio Guide*.



Multimedia Stravinsky: The Rite of Spring  
The Voyager Company  
Microsoft

Multimedia Stravinsky: *The Rite of Spring* offers you an in-depth look at this controversial composition. Author Robert

Winter provides a fascinating commentary that follows the music, giving you greater understanding of the subtle dynamics of the instruments and powerful techniques of Stravinsky. You'll also have the opportunity to discover the ballet that accompanied *The Rite of Spring* in performance. Choreographed by Sergei Diaghilev, the ballet was as unusual for its time as the music. To whet your appetite, play this audio clip.



Multimedia Beethoven: The Ninth Symphony  
The Voyager Company  
Microsoft

Multimedia Beethoven: *The Ninth Symphony* is one of a series of engaging, informative, and interactive musical explorations from Microsoft. It enables you to examine Beethoven's world and life, and explore the form and beauty of one of his foremost compositions. You can compare musical themes, hear selected orchestral instruments, and see the symphonic score come alive. Multimedia Beethoven: *The Ninth Symphony* is an extraordinary opportunity to learn while you listen to one of the world's musical treasures. Explore this inspiring work at your own pace in *A Close Reading*. As you listen to a superb performance of Beethoven's

*The Audiophile Journal, June 1994* 12

Figure 11.1 A compound document

How was this music review created? First, a user created a document and typed the text, then moved, copied, or linked content from other documents. Data objects that, when moved or copied, retain their native, full-featured editing and operating capabilities in their new container are called *OLE embedded objects*.

A user can also link information. An *OLE linked object* represents or provides access to another object that is in another location in the same container or in a different, separate container.


Generally, containers support any level of nested OLE embedded and linked objects. For example, a user can embed a chart in a worksheet, which, in turn, can be embedded in a word-processing document. The model for interaction is consistent at each level of nesting.

## Creating OLE Embedded and OLE Linked Objects

OLE embedded and linked objects are the result of transferring existing objects or creating new objects of a particular type.

### Transferring Objects

Transferring objects into a document follows basic command and direct manipulation interaction methods. The following sections provide additional guidelines for these commands when you use them to create OLE embedded or linked objects.

 For more information about command transfer and direct manipulation transfer methods, see Chapter 5, "General Interaction Techniques."

### The Paste Command

As a general rule, using the Paste command should result in the most complete representation of a transferred object; that is, the object is embedded. However, containers that directly handle the transferred object can accept it optionally as native data instead of embedding it as a separate object, or as a partial or transformed form of the object if that is more appropriate for the destination container.

Use the format of the Paste command to indicate to the user how a transferred object is incorporated by a container. When the user copies a file object, if the container can embed the object, include the object's filename as a suffix to the Paste command. If the object is only a portion of a file, use the short type name — for example, Paste Worksheet or Paste Recording — as shown in Figure 11.2. A short type name can be derived from information stored in the registry. A Paste command with no name implies that the data will be pasted as native information.


 For more information about type names and the system registry, see Chapter 10, "Integrating with the System," and the OLE documentation included in the Microsoft Win32 Software Development Kit (SDK).

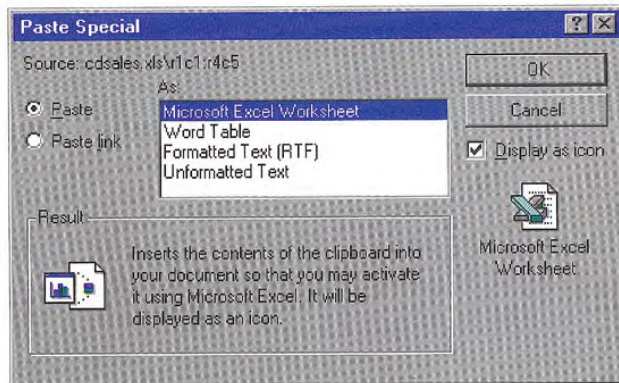



Figure 11.2 The Paste command with short type name

### The Paste Special Command

Supply the Paste Special command to give the user explicit control over pasting in the data as native information, an OLE embedded object, or an OLE linked object. The Paste Special command displays its associated dialog box, as shown in Figure 11.3. This dialog box includes a list box with the possible formats that the data can assume in the destination container.





 The Win32 SDK includes the Paste Special dialog box and other OLE-related dialog boxes that are described in this chapter.

**Figure 11.3** The Paste Special dialog box

In the formats listed in the Paste Special dialog box, include the object's full type name first, followed by other appropriate native data forms. When a linked object has been cut or copied, precede its object type by the word "Linked" in the format list. For example, if the user copies a linked Microsoft Excel worksheet, the Paste Special dialog box shows "Linked Microsoft Excel Worksheet" in the list of format options because it inserts an exact duplicate of the original linked worksheet. Native data formats begin with the destination application's name and can be expressed in the same terms the destination identifies in its own menus. The initially selected format in the list corresponds to the format that the Paste command uses. For example, if the Paste command is displayed as *Paste Object Filename* or *Paste Short Type Name* because the data to be embedded is a file or portion of a file, this is the format that is initially selected in the Paste Special list box.

To support creation of a linked object, the Paste Special dialog box includes a Paste Link option. Figure 11.4 shows this option.



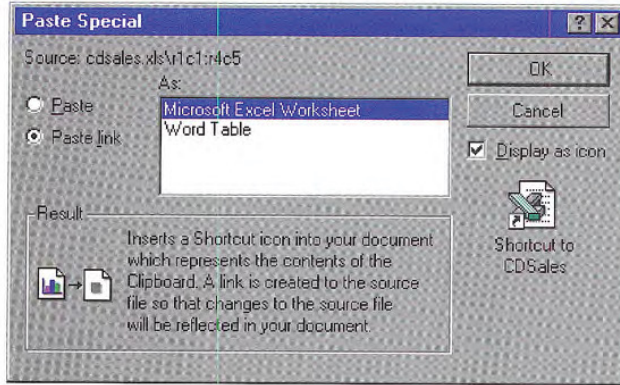


Figure 11.4 Paste Special dialog box with Paste Link option set

A Display As Icon check box allows the user to choose displaying the OLE embedded or linked object as an icon. At the bottom of the dialog box is a section that includes text and pictures that describe the result of the operation. Table 11.1 lists the descriptive text for use in the Paste Special dialog box.

Table 11.1 Descriptive Text for Paste Special Command

Function	Resulting text
Paste as an OLE embedded object.	“Inserts the contents of the Clipboard into your document so you that you may activate it using <i>CompanyName ApplicationName</i> .”
Paste as an OLE embedded object so that it appears as an icon.	“Inserts the contents of the Clipboard into your document so you that you may activate it using <i>CompanyName ApplicationName</i> application. It will be displayed as an icon.”
Paste as native data.	“Inserts the contents of the Clipboard into your document as <i>Native Type Name</i> . [ <i>Optional additional Help sentence</i> .]”
Paste as an OLE linked object.	“Inserts a picture of the contents of the Clipboard 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.”

*(Continued)*

<b>Function</b>	<b>Resulting text</b>
Paste as an OLE linked object so that it appears as a shortcut icon.	“Inserts a Shortcut icon into your document which represents the contents of the Clipboard. A link is created to the source file so that changes to the source file will be reflected in your document.”
Paste as linked native data.	“Inserts the contents of the Clipboard into your document as <i>Native Type Name</i> . A link is created to the source file so that changes to the source file will be reflected in your document.”


### The Paste Link, Paste Shortcut, and Create Shortcut Commands

If linking is a common function in your application, you can optionally include a command that optimizes this process. Use a Paste Link command to support creating a linked object or linked native data. When using the command to create a linked object, include the name of the object preceded by the word “to” — for example, “Paste Link to Latest Sales.” Omitting the name implies that the operation results in linked native data.

Use a Paste Shortcut command to support creation of a linked object that appears as a shortcut icon. You can also include a Create Shortcut command that creates a shortcut icon in the container. Apply these commands to containers where icons are commonly used.

### Direct Manipulation

You should also support direct manipulation interaction techniques, such as drag and drop, for creating OLE embedded or linked objects. When the user drags a selection into a container, the container application interprets the operation using information supplied by the source, such as the selection’s type and format, and by the destination container’s own context, such as the container’s type and its default transfer operation. For example, dragging a spreadsheet cell selection into a word-processing document can result in an OLE embedded table object. Dragging the same cell selection within the

 For more information about using direct manipulation for moving, copying, and linking objects, see Chapter 5, “General Interaction Techniques.”

spreadsheet, however, would likely result in simply transferring the data in the cells. Similarly, the destination container in which the user drops the selection can also determine whether the dragging operation results in an OLE linked object.

For nondefault OLE drag and drop, the container application displays a pop-up menu with appropriate transfer commands at the end of the drag. The choices may include multiple commands that transfer the data in a different format or presentation. For example, as shown in Figure 11.5, a container application could offer the following choices for creating links: Link Here, Link *Short Type Name* Here, and Create Shortcut Here, respectively resulting in a native data link, an OLE linked object displayed as content, and an OLE linked object displayed as an icon. The choices depend on what the container can support.

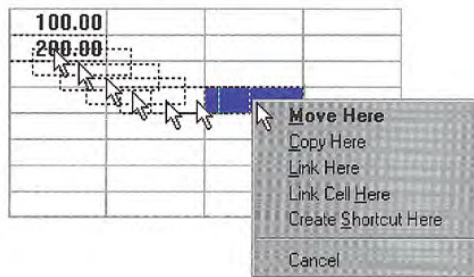


Figure 11.5 Containers can offer different OLE link options

The default appearance of a transferred object also depends on the destination container application. For most types of destinations, make the default command one that results in the data or content presentation of the object (or in the case of an OLE linked object, a representation of the content), rather than as an icon. If the user chooses Create Shortcut Here as the transfer operation, display the transferred object as an icon. If the object cannot be displayed as content — for example, because it does not support OLE — always display the object as an icon.



## Transfer of Data to Desktop

The system allows the user to transfer data selection within a file to the desktop or folders providing that the application supports the OLE transfer protocol. For move or copy operations — using the Cut, Copy, and Paste commands or direct manipulation — the transfer operation results in a file icon called a *scrap*. A link operation also results in a shortcut icon that represents a shortcut into a document.

When the user transfers a scrap into a container supported by your application, integrate it as if it were being transferred from its original source. For example, if the user transfers a selected range of cells from a spreadsheet to the desktop, it becomes a scrap. If the user transfers the resulting scrap into a word-processing document, the document should incorporate the scrap as if the cells were transferred directly from the spreadsheet. Similarly, if the user transfers the scrap back into the spreadsheet, the spreadsheet should integrate the cells as if they had been transferred within that spreadsheet. (Typically, internal transfers of native data within a container result in repositioning the data rather than transforming it.)

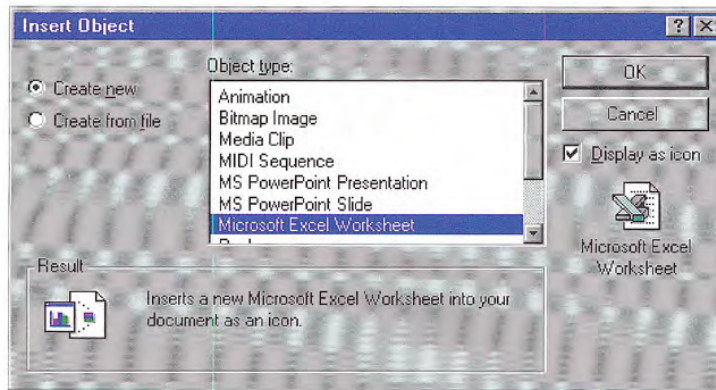
## Inserting New Objects

In addition to transferring objects, you can support user creation of OLE embedded or linked objects by generating a new object based on an existing object or object type and inserting the new object into the target container.

## The Insert Object Command


Include an Insert Object command on the menu responsible for creating or importing new objects into a container, such as an Insert menu. If no such menu exists, use the Edit menu. When the user selects this command, display the Insert Object dialog box, as shown in Figure 11.6. This dialog box allows the user to generate new objects based on their object type or an existing file.





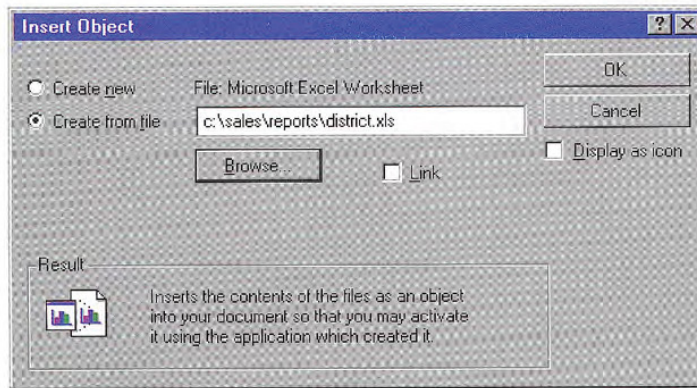
**Figure 11.6** The Insert Object dialog box

The type list is composed of the type names of registered types. When the user selects a type from the list box and chooses the OK button, an object of the selected type is created and embedded.

 For more information about type names and the registry, see Chapter 10, “Integrating with the System.”

The user can also create an OLE embedded or linked object from an existing file, using the Create From File and Link options. When the user sets these options and chooses the OK button, the result is the same as directly copying or linking the selected file.

When the user chooses the Create From File option button, the Object Type list is removed, and a text box and Browse button appear in its place, as shown in Figure 11.7. Ignore any selection formerly displayed in the Object Type list box (shown in Figure 11.6).




**Figure 11.7** Creating an OLE embedded object from an existing file

The text box initially includes the current directory as the selection. The user can edit the current directory path when specifying a file. As an alternative, the Browse button displays an Open dialog box that allows the user to navigate through the file system to select a file. Use the file's type to determine the type of the resulting OLE object.

Use the Link check box to support the creation of an OLE linked object to the file specified. The Insert Object dialog box displays this option only when the user chooses the Create From File option. This means that a user cannot insert an OLE linked object when choosing the Create New option button, because linked objects can be created only from existing files.

The Display As Icon check box in the Insert Object dialog box enables the user to specify whether to display the OLE object as an icon. When this option is set, the icon appears beneath the check box. An OLE linked object displayed as an icon is the equivalent of a shortcut icon. It should appear with the link symbol over the icon.

At the bottom of the Insert Object dialog box, text and pictures describe the final outcome of the insertion. Table 11.2 outlines the syntax of descriptive text to use within the Insert Object dialog box.


 If the user chooses a non-OLE file for insertion, it can be inserted only as an icon. The result is an OLE package. A *package* is an OLE encapsulation of a file so that it can be embedded in an OLE container. Because packages support limited editing and viewing capabilities, support OLE for all your object types so they will not be converted into packages.

**Table 11.2 Descriptive Text for Insert Object Dialog Box**

<b>Function</b>	<b>Resulting text</b>
Create a new OLE embedded object based on the selected type.	“Inserts a new <i>Type Name</i> into your document.”
Create a new OLE embedded object based on the selected type and display it as an icon.	“Inserts a new <i>Type Name</i> into your document as an icon.”
Create a new OLE embedded object based on a selected 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 a new OLE embedded object based on a selected file (copies the file) and display it as an 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 an OLE linked object that is linked to a selected file.	“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 an OLE linked object that is linked to a selected file and display it as a Shortcut icon.	“Inserts a Shortcut icon into your document which represents the file. The Shortcut icon will be linked to the original file, so that you can quickly open the original from inside your document.”

You can also use the context of the current selection in the container to determine the format of the newly created object and the effect of it being inserted into the container. For example, an inserted graph can automatically reflect the data in a selected table. Use the following guidelines to support predictable insertion:

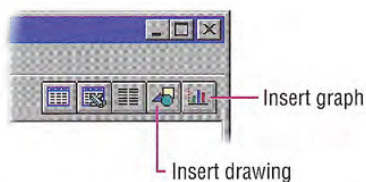
- If an inserted object is not based on the current selection, follow the same conventions as for a Paste command and add or replace the selection depending on the context. For example, in text or list contexts, where the selection represents a specific insertion location, replace the active selection. For nonordered or Z-ordered contexts, where the selection does not represent an explicit insertion point, add the object, using the destination's context to determine where to place the object.
- If the new object is automatically connected (linked) to the selection (for example, an inserted graph based on selected table data), insert the new object in addition to the selection and make the inserted object the new selection.

 For more information about the guidelines for inserting an object with a Paste command, see Chapter 5, "General Interaction Techniques."

After inserting an OLE embedded object, activate it for editing. However, if the user inserts an OLE linked object, do not activate the object.

### Other Techniques for Inserting Objects

The Insert Object command provides support for inserting all registered OLE objects. You can include additional commands tailored to provide access to common or frequently used object types. You can implement these as additional menu commands or as toolbar buttons or other controls. These buttons provide the same functionality as the Insert Object dialog box, but perform more efficiently. Figure 11.8 illustrates two examples. The drawing button inserts a new blank drawing object; the graph button creates a new graph that uses the data values from a currently selected table.



**Figure 11.8** Using toolbar buttons for creating new objects



## Displaying Objects

While a container can control whether to display an OLE embedded or linked object in its content or icon presentation, the container requests the object to display itself. In the content presentation, the object may be visually indistinguishable from native objects, as shown in Figure 11.9.

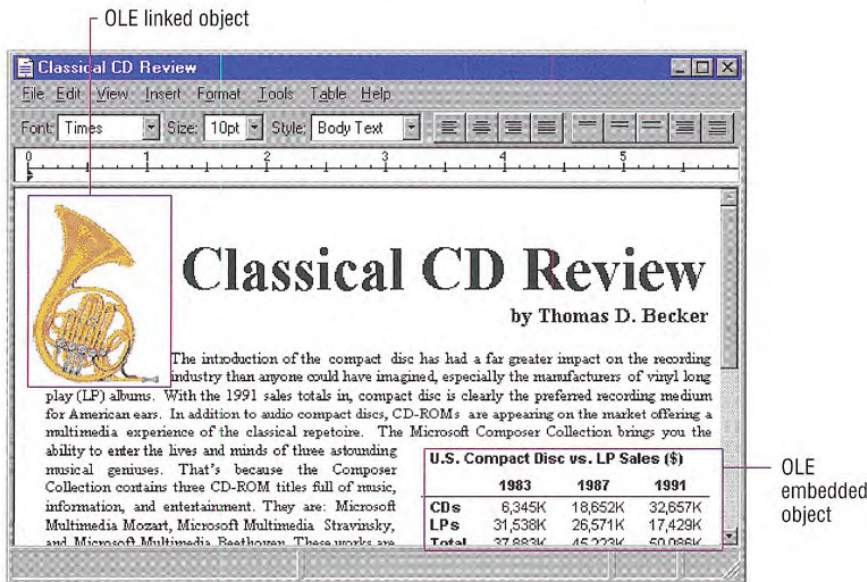



Figure 11.9 A compound document containing OLE objects

You may find it preferable to enable the user to visually identify OLE embedded or linked objects without interacting with them. To do so, you can include a Show Objects command that, when chosen, displays a solid border, one pixel wide, drawn in the window text

color around the extent of an OLE embedded object and a dotted border around OLE linked objects (shown in Figure 11.10). If the container application cannot guarantee that an OLE linked object is up-to-date with its source because of an unsuccessful automatic update or a manual link, the system should draw a dotted border using the system grayed text color to suggest that the OLE linked object is out of date. The border should be drawn around a container's first-level objects only, not objects nested below this level.

 The **GetSysColor** function provides the current settings for window text color (COLOR\_WINDOWTEXT) and grayed text color (COLOR\_GRAYTEXT). For more information about this function, see the documentation included in the Win32 SDK.

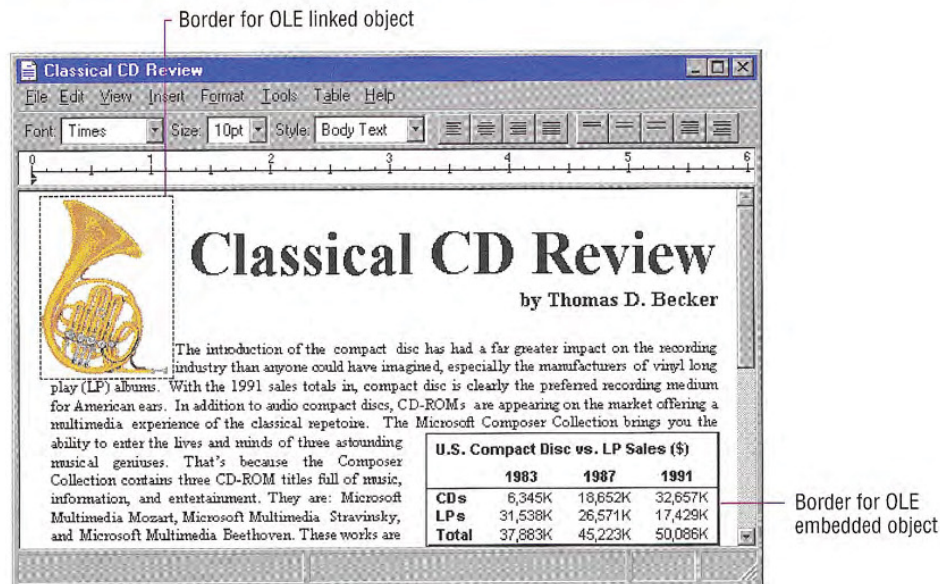


Figure 11.10 Identifying OLE objects using borders

If these border conventions are not adequate to distinguish OLE embedded and linked objects, you can optionally include additional distinctions; however, make them clearly distinct from the appearance for any standard visual states and distinguish OLE embedded from OLE linked objects.

Whenever the user creates an OLE linked or embedded object with the Display As Icon check box set, display the icon using the icon of its type, unless the user explicitly changes it. A linked icon also includes the shortcut graphic. If an icon is not registered in the registry for the object, use the system-generated icon.

An icon includes a label. When the user creates an OLE embedded object, define the icon's label to be one of the following, based on availability:

- The name of the object, if the object has an existing human-readable name such as a filename without its extension.
- The object's registered short type name (for example, Picture, Worksheet, and so on), if the object does not have a name.
- The object's registered full type name (for example, a bitmap image, a Microsoft Excel Worksheet), if the object has no name or registered short type name.
- "Document" if an object has no name, a short type name, or a registered type name.

When an OLE linked object is displayed as an icon, define the label using the source filename as it appears in the file system, preceded by the words "Shortcut to" — for example, "Shortcut to Annual Report." The path of the source is not included. Avoid displaying the filename extension unless the user chooses the system option to display extensions or the file type is not registered.

When the user creates an OLE object linked to only a portion of a document (file), follow the same conventions for labeling the shortcut icon. However, because a container can include multiple links to different portions of the same file, you may want to provide further identification to differentiate linked objects. You can do this by appending a portion of the end of the link path (moniker). For example, you may want to include everything from the end of the path up to the last or next to last occurrence of a link path delimiter. OLE applications should use the exclamation point (!) character for identifying a data range. However, the link path may include other types of delimiters. Be careful when deriving an identifier from the link path to format the additional information using only valid filename characters so that if the user transfers the shortcut icon to a folder or the desktop, the name can still be used.




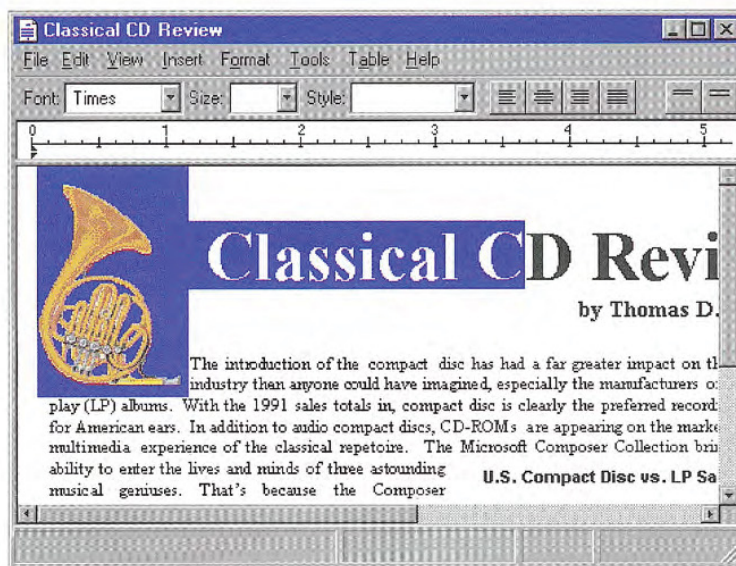
The system provides support to automatically format the name correctly if you use the **GetIconOfFile** function. For more information about this function, see the OLE documentation included in the Win32 SDK.



## Selecting Objects

An OLE embedded or linked object follows the selection behavior and appearance techniques supported by its container; the container application supplies the specific appearance of the object. For example, Figure 11.11 shows how the linked drawing of a horn is handled as part of a contiguous selection in the document.

 For information about selection, see Chapter 5, “General Interaction Techniques.” For information about selection appearance, see Chapter 13, “Visual Design.”



**Figure 11.11** An OLE linked object as part of a multiple selection

When the user individually selects the object, display the object with an appropriate selection appearance. For example, for the content view of an object, display it with handles, as shown in Figure 11.12. For OLE linked objects, overlay the content view’s lower left corner with the shortcut graphic. In addition, if your application’s window includes a status bar that displays messages, display an appropriate description of how to activate the object (see Table 11.3 later in this chapter).



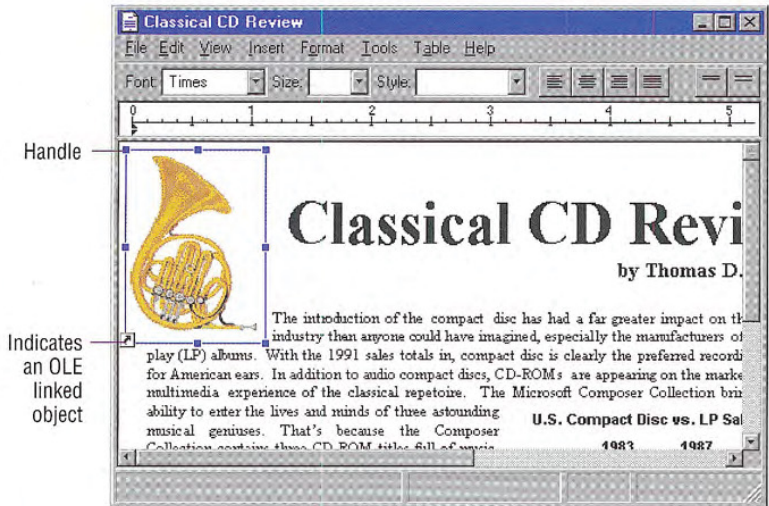


Figure 11.12 An individually selected OLE linked object

When the object is displayed as an icon, use the same selection appearance as for selected icons in folders and on the desktop, as shown in Figure 11.13.

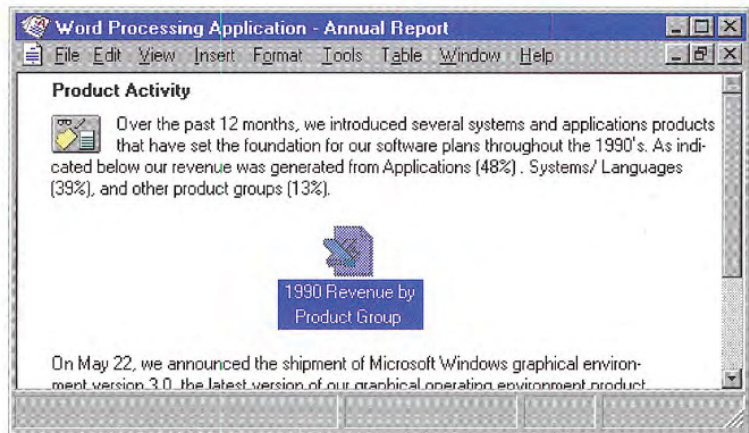



Figure 11.13 A selected OLE object displayed as an icon

## Accessing Commands for Selected Objects

A container application always displays the commands that can be applied to its objects. When the user selects an OLE embedded or linked object as part of the selection of native data in a container, enable commands that apply to the selection as a whole. When the user individually selects the object, enable only commands that apply specifically to the object. The container application retrieves these commands from what has been registered by the object's type in the registry and displays these commands in the menus that are supplied for the object. If your application includes a menu bar, include the selected object's commands on a submenu of the Edit menu, or as a separate menu on the menu bar. Use the name of the object as the text for the menu item. If you use the short type name as the name of the object, add the word "Object." For an OLE linked object, use the short type name, preceded by the word "Linked." Figure 11.14 shows these variations.

 You can also support operations based on the selection appearance. For example, you can support operations, such as resizing, using the handles you supply. When the user resizes a selected OLE object, however, scale the presentation of the object, because there is no method by which another operation, such as cropping, can be applied to the OLE object.



**Figure 11.14** Drop-down menus for selected OLE object

Define the first letter of the word "Object", or its localized equivalent, as the access character for keyboard users. When no object is selected, display the command with just the text, "Object", and disable it.

A container application should also provide a pop-up menu for a selected OLE object (shown in Figure 11.15), displayed using the standard interaction techniques for pop-up menus (clicking with mouse button 2). Include on this menu the commands that apply to the object as a whole as a unit of content, such as transfer commands and the object's registered commands. In the pop-up menu, display the object's registered commands as individual menu items rather than in a cascading menu. It is not necessary to include the object's name or the word "Object" as part of the menu item text.

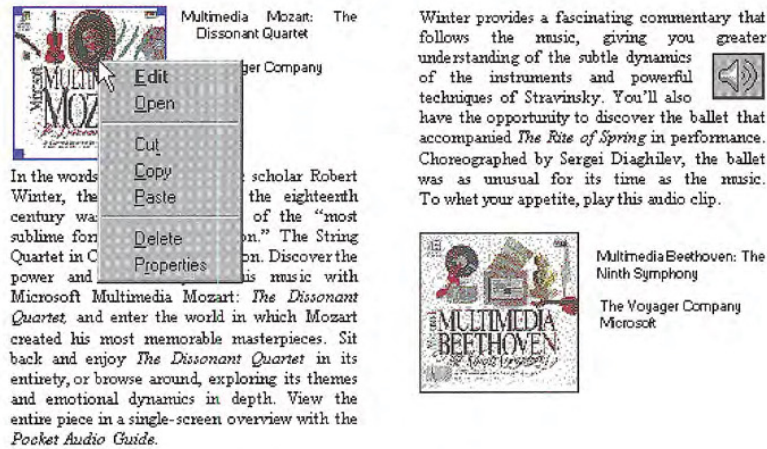


Figure 11.15 Pop-up menu for an OLE embedded picture

In the drop-down menu and the pop-up menu, include a Properties command. You can also include commands that depend on the state of the object. For example, a media object that uses Play and Rewind as operations disables Rewind when the object is at the beginning of the media object.

If an object's type is not registered, you still supply any commands that can be appropriately applied to the object as content, such as transfer commands, alignment commands, and an Edit and Properties command. When the user chooses the Edit command, display the system-supplied message box, as shown in Figure 11.41. This message box provides access to a dialog box that enables the user to choose from a list of applications that can operate on the type or convert the object's type.



## Activating Objects

Although selecting an object provides access to commands applicable to the object as a whole, it does not provide access to editing the content of the object. The object must be activated in order to provide user interaction with the internal content of the object. There are two basic models for activating objects: outside-in activation and inside-out activation.

### Outside-in Activation

*Outside-in activation* requires that the user choose an explicit activation command. Clicking, or some other selection operation, performed on an object that is already selected simply reselects that object and does not constitute an explicit action. The user activates the object by using a particular command such as Edit or Play, usually the object's default command. Shortcut actions that correspond to these commands, such as double-clicking or pressing a shortcut key, can also activate the object. Most OLE container applications employ this model because it allows the user to easily select objects and reduces the risk of inadvertently activating an object whose underlying code may take a significant amount of time to load and dismiss.

When supporting outside-in activation, display the standard pointer (northwest arrow) over an outside-in activated object within your container when the object is selected, but inactive. This indicates to the user that the outside-in object behaves as a single, opaque object. When the user activates the object, the object's application displays the appropriate pointer for its content. Use the registry to determine the object's activation command.

### Inside-out Activation

With *inside-out activation*, interaction with an object is direct; that is, the object is activated as the user moves the pointer over the extent of the object. From the user's perspective, inside-out objects are indistinguishable from native data because the content of the object



is directly interactive and no additional action is necessary. Use this method for the design of objects that benefit from direct interaction, or when activating the object has little effect on performance or use of system resources.

Inside-out activation requires closer cooperation between the container and the object. For example, when the user begins a selection within an inside-out object, the container must clear its own selection so that the behavior is consistent with normal selection interaction. An object supporting inside-out activation controls the appearance of the pointer as it moves over its extent and responds immediately to input. Therefore, to select the object as a whole, the user selects the border, or some other handle, provided by the object or its container. For example, the container application can support selection techniques, such as region selection that select the object.

Although the default behavior for an OLE embedded object is outside-in activation, you can store information in the registry that indicates that an object's type (application class) is capable of inside-out activation (`OLEMISC_INSIDEOUT`) and prefers inside-out behavior (`OLEMISC_ACTIVATEWHENVISIBLE`). You can set these values in a `MiscStatus` subkey, under the `CLSID` subkey of the `HKEY_CLASSES_ROOT` key.



For more information about how to access `OLEMISC_INSIDEOUT` and `OLEMISC_ACTIVATEWHENVISIBLE` and the `IOleObject::GetMiscStatus` function, see the OLE documentation included in the Win32 SDK.

## Container Control of Activation

The container application determines how to activate its component objects: either it allows the inside-out objects to handle events directly or it intercedes and only activates them upon an explicit action. This is true regardless of the capability or preference setting of the object. That is, even though an object may register inside-out activation, it can be treated by a particular container as outside-in. Use an activation style for your container that is most appropriate for its specific use and is in keeping with its own native style of activation so that objects can be easily assimilated.

Regardless of the activation capability of the object, a container should always activate its content objects of the same type consistently. Otherwise, the unpredictability of the interface is likely to impair its usability. Following are four potential container activation methods and when to use them.

Activation method	When to use
Outside-in throughout	This is the most common design for containers that often embed large OLE objects and deal with them as whole units. Because many available OLE objects are not yet inside-out capable, most compound document editors support outside-in throughout to preserve uniformity.
Inside-out throughout	Ultimately, OLE containers will blend embedded objects with native data so seamlessly that the distinction dissolves. Inside-out throughout containers will become more feasible as increasing numbers of OLE objects support inside-out activation.
Outside-in plus inside-out preferred objects	Some containers may use an outside-in model for large, foreign embeddings but also include some inside-out preferred objects as though they were native objects (by supporting <code>OLEMISC_ACTIVATE_WHENVISIBLE</code> ). For example, an OLE document might present form control objects as inside-out native data while activating larger spreadsheet and chart objects as outside-in.
Switch between inside-out throughout and outside-in throughout	Visual programming and forms layout design applications may include design and run modes. In this type of environment, a container typically holds an object that is capable of inside-out activation (if not preferable) and alternates between outside-in throughout when designing and inside-out throughout when running.


## OLE Visual Editing of OLE Embedded Objects

One of the most common uses for activating an object is editing its content in its current location. Supporting this type of in-place interaction is called *OLE visual editing*, because the user can edit the object within the visual context of its container.

Unless the container and the object both support inside-out activation, the user activates an embedded object for visual editing by selecting the object and choosing its Edit command, either from a drop-down or pop-up menu. You can also support shortcut techniques. For example, by making Edit the object's default operation, the user can double-click to activate the object for editing. Similarly, you can support pressing the ENTER key as a shortcut for activating the object.

When the user activates an OLE embedded object for visual editing, the user interface for its content becomes available and blended into its container application's interface. The object can display its frame *adornments*, such as row or column headers, handles, or scroll bars, outside the extent of the object and temporarily cover neighboring material. The object's application can also change the menu interface, which can range from adding menu items to existing drop-down menus to replacing entire drop-down menus. The object can also add toolbars, status bars, supplemental palette windows, and provide pop-up menus for selected content.

The degree of interface blending varies based on the nature of the OLE embedded object. Some OLE embedded objects may require extensive support and consequently result in dramatic changes to the container application's interface. Finer grain objects that emulate the native components of a container may have little or no need to make changes in the container's user interface. The container always determines the degree to which an OLE embedded object's interface can be blended with its own, regardless of the capability or preference of the OLE embedded object. A container application that provides its own interface for an OLE embedded object can suppress an OLE embedded object's own interface. Figure 11.16 shows how the interface might appear when its embedded worksheet is active.

 Although earlier versions of OLE user interface documentation suggested the ALT+ENTER key combination to activate an object if the ENTER key was already assigned, this key combination is now the recommended shortcut key for the Properties command. Instead, support the pop-up menu shortcut key. This enables the user to activate the object by selecting the command from the pop-up menu.



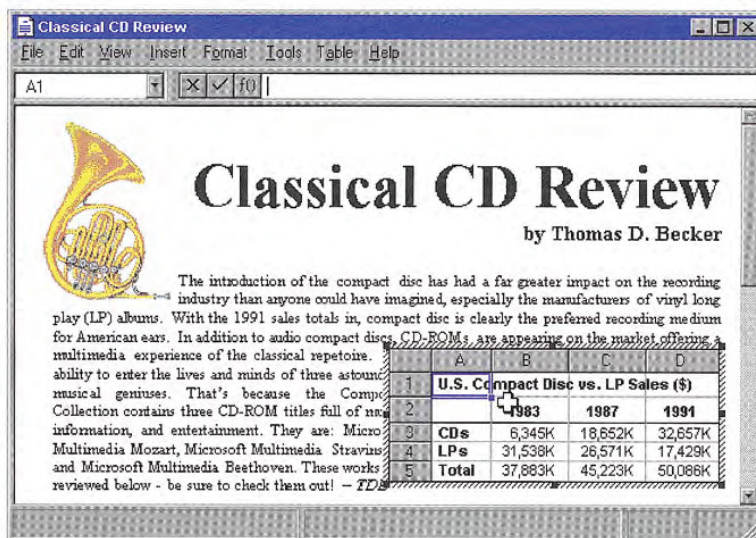


Figure 11.16 An embedded worksheet activated for OLE visual editing

When the user activates an OLE embedded object, avoid changing the view and position of the rest of the content in the window. Although it may seem reasonable to scroll the window and thereby preserve the content's position, doing so can disturb the user's focus, because the active object shifts down to accommodate a new toolbar and shifts back up when it is deactivated. An exception may be when the activation exposes an area in which the container has nothing to display. However, even in this situation, you may wish to render a visible region or filled area that corresponds to the background area outside the visible edge of the container so that activation keeps the presentation stable.

Activation does not affect the title bar. Always display the top-level container's name. For example, when the worksheet shown in Figure 11.16 is activated, the title bar continues to display the name of the document in which the worksheet is embedded and not the name of the worksheet. You can provide access to the name of the worksheet by supporting property sheets for your OLE embedded objects.



A container can contain multiply nested OLE embedded objects. However, only a single level is active at any one time. Figure 11.17 shows a document containing an active embedded worksheet with an embedded graph of its own. Clicking on the graph merely selects it as an object within the worksheet.

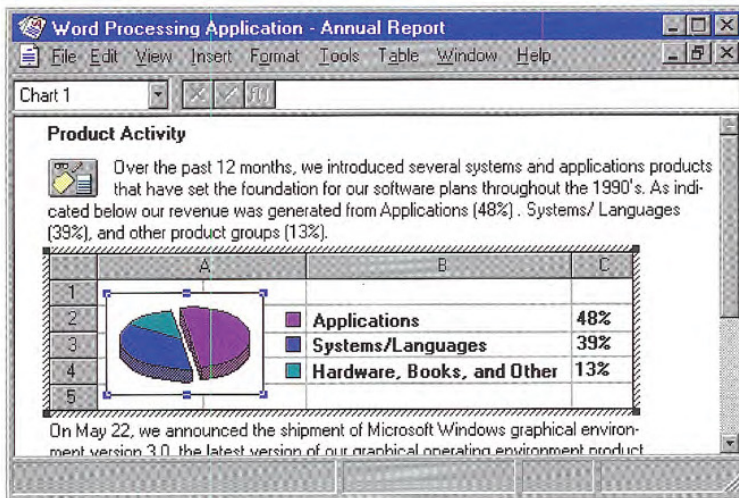


Figure 11.17 A selected graph within an active worksheet

Activating the embedded graph, for example, by choosing the graph's Edit command, activates the object for OLE visual editing, displaying the graph's menus in the document's menu bar. This is shown in Figure 11.18. At any given time, only the interface for the currently active object and the topmost container are presented; intervening parent objects do not remain visibly active.

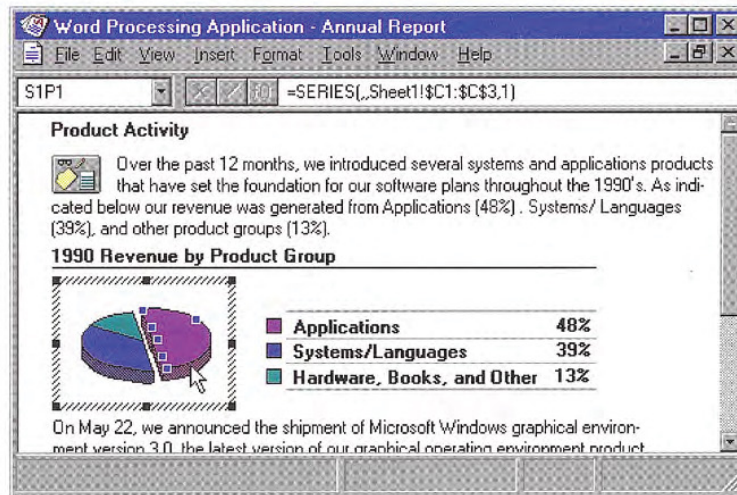



Figure 11.18 An active graph within a worksheet

An OLE embedded object should support OLE visual editing at any view magnification level because its container's view can be scaled arbitrarily. If an object cannot accommodate OLE visual editing in its container's current view scale, or if its container does not support OLE visual editing, open the object into a separate window for editing. For more information about opening OLE embedded objects, see the section, "Opening Embedded Objects," later in this chapter.

For any user interaction outside the extent of an active object, such as when the user selects or activates another object in the container, deactivate the current object and give the focus to the new object. This is also true for an object that is nested in the currently active object. An OLE embedded object application should also support user deactivation when the user presses the ESC key, after which it becomes the selected object of its container. If the object uses the ESC key at all times for its internal operation, the SHIFT+ESC key should deactivate the object.

Edits made to an active object become part of the container immediately and automatically, just like edits made to native data. Consequently, do not display an “Update changes?” message box when the object is deactivated. Remember that the user can abandon changes to the entire container, embedded or otherwise, if the topmost container includes an explicit command that prompts the user to save or discard changes to the container’s file.

While Edit is the most common command for activating an OLE embedded object for OLE visual editing, other commands can also create such activation. For example, when the user carries out a Play command on a video clip, you can display a set of commands that allow the user to control the clip (Rewind, Stop, and Fast Forward). In this case, the Play command provides a form of OLE visual editing.

 OLE embedded objects participate in the undo stack of the window in which they are activated. For more information about embedded objects and the undo stack, see the section, “Undo Operations for Active and Open Objects,” later in this chapter.

## The Active Hatched Border

If a container allows an OLE embedded object’s user interface to change its user interface, then the active object’s application displays a hatched border around itself to show the extent of the OLE visual editing context (shown in Figure 11.19). For example, if an active object places its menus in the topmost container’s menu bar, display the active hatched border. The object’s request to display its menus in the container’s menu bar must be granted by the container application. If the object’s menus do not appear in the menu bar (because the object did not require menus or the container refused its request for menu display), or the object is otherwise accommodated by the container’s user interface, you need not display the hatched border. The hatched pattern is made up of 45-degree diagonal lines.



	A	B	C	D
1	U.S. Compact Disc vs. LP Sales (\$)			
2		1983	1987	1991
3	CDs	6,345K	18,652K	32,657K
4	LPs	31,538K	26,571K	17,429K
5	Total	37,883K	45,223K	50,086K

**Figure 11.19** Hatched border around active OLE embedded objects

The active object takes on the appearance that is best suited for its own editing; for example, the object may display frame adornments, table gridlines, handles, and other editing aids. Because the hatched border is part of the object's territory, the active object defines the pointer that appears when the user moves the mouse over the border.

Clicking in the hatched pattern (and not on the handles) is interpreted by the object as clicking just inside the edge of the border of the active object. The hatched area is effectively a hot zone that prevents inadvertent deactivations and makes it easier to select the content of the embedded object.

## Menu Integration

As the user activates different objects, different commands need to be accessed in the window's user interface. The following classification of menus — primary container menu, workspace menu, and active object menus — separates the interface based on menu groupings. This classification enhances the usability of the interface by defining the interface changes as the user activates or deactivates different objects.



## Primary Container Menu

The topmost or primary container viewed in a primary window controls the work area of that window. If the primary container includes a menu bar, it supplies at least one menu that includes commands that apply to the primary container as an entire unit. For example, for document file objects, use a File menu for this purpose, as shown in Figure 11.20. This menu includes document and file level commands such as Open, Save, and Print. Always display the primary container menu in the menu bar at all times, regardless of which object is active.

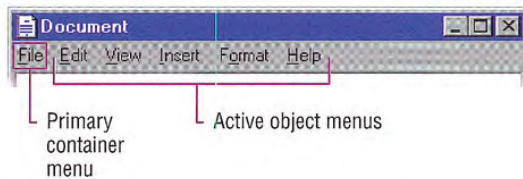


Figure 11.20 OLE visual editing menu layout

## Workspace Menu

An MDI-style application also includes a workspace menu (typically labeled "Window") on its menu bar that provides commands for managing the document windows displayed within it, as shown in Figure 11.21. Like the primary container menu, the workspace menu should always be displayed, independent of object activation.

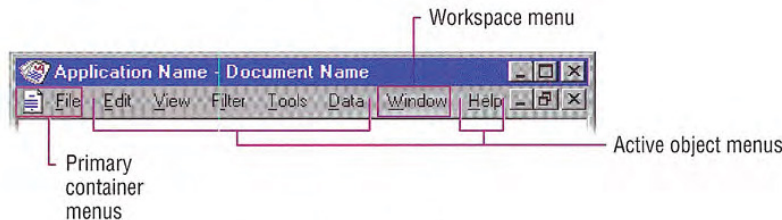


Figure 11.21 OLE visual editing menu layout for MDI

## Active Object Menus

Active objects can define menus that appear on the primary container's menu bar that operate on their content. Place commands for moving, deleting, searching and replacing, creating new items, applying tools, styles, and Help on these menus. As the name suggests, active object commands are executed by the currently active object and apply only within the extent of that object. If no embedded objects are active, but the window is active, the primary container should be considered the active object.

An active object's menus typically occupy the majority of the menu bar. Organize these menus following the same order and grouping that you display when the user opens the object into its own window. Avoid naming your active object menus File or Window, because primary containers often use those titles. Objects that use direct manipulation as their sole interface need not provide active object menus or alter the menu bar when activated.

The active object can display a View menu. However, when the object is active, include only commands that apply to the object. If the object's container requires its document or window-level "viewing" commands to be available while an object is active, place them on a menu that represents the primary container window's pop-up menu and on the Window menu — if present.

When designing the interface of selected objects within an active object, follow the same guidelines as that of a primary container and one of its selected OLE embedded objects; that is, the active object displays the commands of the selected object (as registered in the registry) either as submenus of its menus or as separate menus.

An active object also has the responsibility for defining and displaying pop-up menus for its content, with commands appropriate to apply to any selection within it. Figure 11.22 shows an example of a pop-up menu for a selection within an active bitmap image.



Figure 11.22 Example of pop-up menu for a selection in an active object

## Keyboard Interface Integration

In addition to integrating the menus, you must also integrate the access keys and shortcut keys used in these menus.

### Access Keys

The access keys assigned to the primary container's menu, an active object's menus, and MDI workspace menus should be unique. Following are guidelines for defining access keys for integrating these menu names:

- Use the first letter of the menu of the primary container as its access key character. Typically, this is "F" for File. Use "W" for a workspace's Window menu. Localized versions should use the appropriate equivalent.
- Use characters other than those assigned to the primary container and workspace menus for the menu titles of active OLE embedded objects. (If an OLE embedded object has previously existed as a standalone document, its corresponding application avoids these characters already.)




- Define unique access keys for an object's registered commands and avoid characters that are potential access keys for common container-supplied commands, such as Cut, Copy, Paste, Delete, and Properties.

Despite these guidelines, if the same access character is used more than once, pressing an ALT+*letter* combination cycles through each command, selecting the next match each time it is pressed. To carry out the command, the user must press the ENTER key when it is selected. This is standard system behavior for menus.

## Shortcut Keys

For primary containers and active objects, follow the shortcut key guidelines covered in this guide. In addition, avoid defining shortcut keys for active objects that are likely to be assigned to the container. For example, include the standard editing and transfer (Cut, Copy, and Paste) shortcut keys, but avoid File menu or system-assigned shortcut keys. There is no provision for registering shortcut keys for a selected object's commands.

If a container and an active object share a common shortcut key, the active object captures the event. That is, if the user activates an OLE embedded object, its application code directly processes the shortcut key. If the active object does not process the key event, it is available to the container, which has the option to process it or not. This applies to any level of nested OLE embedded objects. If there is duplication between shortcut keys, the user can always direct the key based on where the active focus is by activating that object. To direct a shortcut key to the container, the user deactivates an OLE embedded object — for example, by selecting in the container — but outside the OLE embedded object. Activation, not selection, of an OLE embedded object allows it to receive the keyboard events. The exception is inside-out activation, where activation results from selection.

 For more information about defining shortcut keys, see Chapter 4, "Input Basics," and Appendix B, "Keyboard Interface Summary."



## Toolbars, Frame Adornments, and Palette Windows

Integrating drop-down and pop-up menus is straightforward because they are confined within a particular area and follow standard conventions. Toolbars, frame adornments (as shown in Figure 11.23), and palette windows can be constructed less predictably, so it is best to follow a replacement strategy when integrating these elements for active objects. That is, toolbars, frame adornments, and palette windows are displayed and removed as entire sets rather than integrated at the individual control level—just like menu titles on the menu bar.

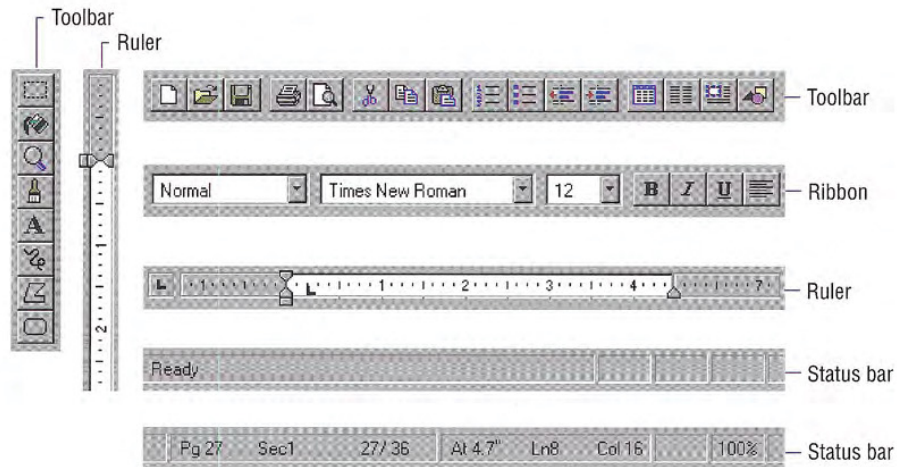


Figure 11.23 Examples of toolbars, status bars, and frame adornments

When the user activates an object, the object application requests a specific area from its container in which to post its tools. The container application determines whether to:

- Replace its tool (or tools) with the tools of the object, if the requested space is already occupied by a container tool.
- Add the object's tool (or tools), if a container tool does not occupy the requested space.
- Refuse to display the tool (or tools) at all. This is the least desirable method.

Toolbars, frame adornments, and palette windows are all basically the same interfaces — they differ 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 reside, and you determine their location by their scope. Figure 11.24 shows possible positions for interface controls.

<b>Location</b>	<b>Description</b>
Object frame	Place object-specific controls, such as a table header or a local coordinate ruler, directly adjacent to the object itself for tightly coupled interaction between the object and its interface. An object (such as a spreadsheet) can include scrollbars if its content extends beyond the boundaries of its frame.
Pane frame	Locate view-specific controls at the pane level. Rulers and viewing tools are common examples.
Document (primary container) window frame	Attach tools that apply to the entire document (or documents in the case of an MDI window) just inside any edge of its primary window frame. Popular examples include ribbons, drawing tools, and status lines.
Windowed	Display tools in a palette window — this allows the user to place them as desired. A palette window typically floats above the primary window and any other windows of which it is part.

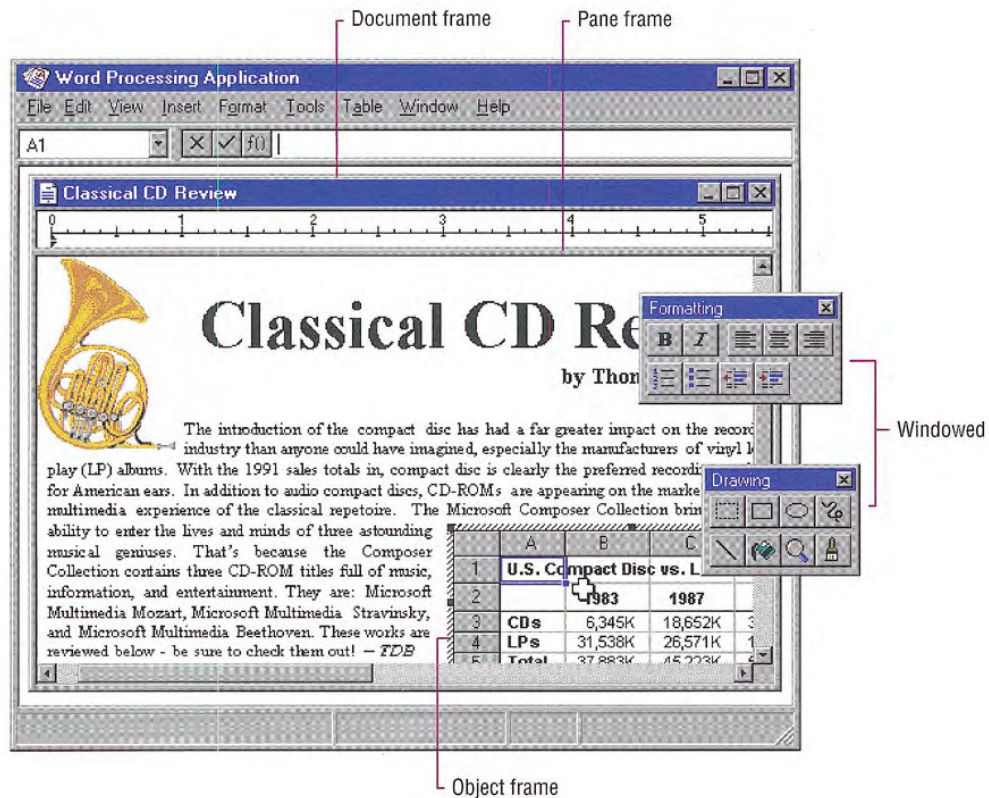



Figure 11.24 Possible locations for interface controls

When determining where to locate a tool area, avoid situations that cause the view to shift up and down as different-sized tool areas are displayed or removed as the user activates different objects. This can be disruptive to the user's task.

Because container tool areas can remain visible while an object is active, they are available to the user simply by interacting with them — this can reactivate the container application. The container determines whether to activate or leave the object active. If toolbar buttons of an active object represent a primary container or workspace commands, such as Save, Print, or Open, disable them.

 For more information about the negotiation protocols used for activation, see the OLE documentation included in the Win32 SDK.



As the user resizes or scrolls its container's area, an active object and its toolbar or frame adornments placed on the object frame are clipped, as is all container content. These interface control areas lie in the same plane as the object. Even when the object is clipped, the user can still edit the visible part of the object in place and while the visible frame adornments are operational.

Some container applications scroll at certain increments that may prevent portions of an OLE embedded object from being visually edited. For example, consider a large picture embedded in a worksheet cell. The worksheet scrolls vertically in complete row increments; 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, its bottom portion is clipped and consequently never viewed or edited in place. In cases like this, the user can open the picture into its own window for editing.

Window panes clip frame adornments of nested embedded objects, but not by the extent of any parent object. Objects at the very edge of their container's extent or boundary potentially display adornments that extend beyond the bounds of the container's defined area. In this case, if the container displays items that extend beyond the edge, display all the adornments; otherwise, clip the adornments at the edge of the container. Do not temporarily move the object within its container just to accommodate the appearance of an active embedded object's adornments. A pane-level control can potentially be clipped by the primary (or parent, in the case of MDI) window frame, and a primary window adornment or control is clipped by other primary windows.

## Opening OLE Embedded Objects

The previous sections have focused on OLE visual editing — editing an OLE embedded object in place; that is, its current location is within its container. Alternatively, the user can also open embedded objects into their own window. This gives the user the opportunity of seeing more of the object or seeing the object in a different view state. Support this operation by registering an Open command for the object. When the user chooses the Open command of an object, it opens it into a separate window for editing, as shown in Figure 11.25.



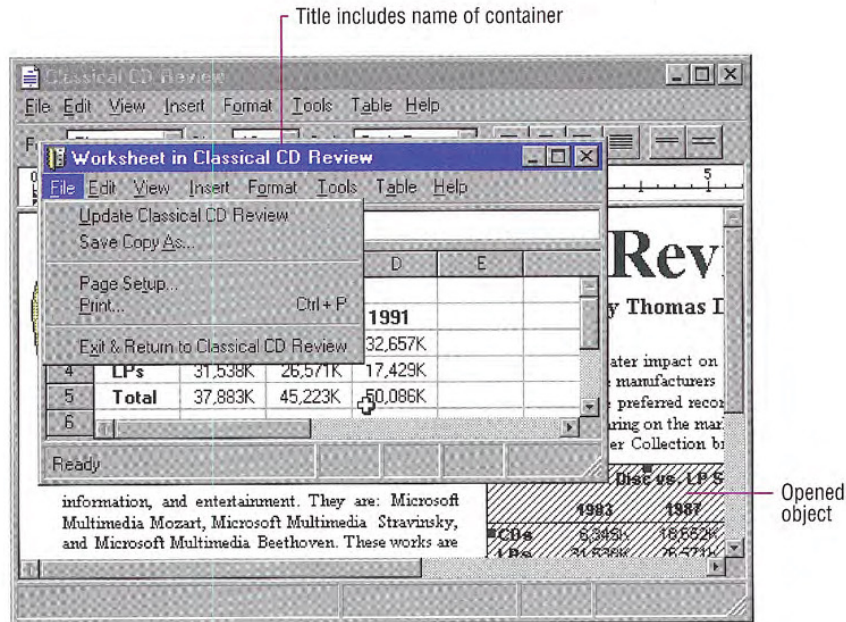



Figure 11.25 An opened OLE embedded worksheet

After opening an object, the container displays it masked with an “open” hatched (lines at a 45-degree angle) pattern that indicates the object is open in another window, as shown in Figure 11.26.

U.S. Compact Disc vs. LP Sales (\$)	1983	1987	1991
CDs	8,345K	18,652K	32,657K
LPs	31,538K	26,571K	17,429K
Total	37,883K	45,223K	50,086K

Figure 11.26 An opened OLE embedded object

Format the title text for the open object’s window as “*Object Name* in *Container Name*” (for example, “Sales Worksheet in Classical CD Review”). Including the container’s name emphasizes that the object in the container and the object in the open window are considered the same object.

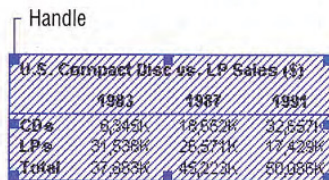
 This convention for the title bar text applies only when the user opens an OLE embedded object. When the user activates an OLE embedded object for visual editing, do not change the title bar text.

An open OLE embedded object represents an alternate window onto the same object within the container as opposed to a separate application that updates changes to the container document. Therefore, reflect edits immediately and automatically in the object in the window of its container. There is no need to display an update confirmation message upon exiting the open window. Nevertheless, you can still include an Update *Container Filename* command in the window of the open object to allow the user to request an update explicitly. This is useful if you cannot support frequent “real-time” image updates because of operational performance. In addition, when the user closes an open object’s window, automatically update its presentation in the container’s window. Provide a Close & Return To *Container Filename* or Exit & Return To *Container Filename* on the File menu replacing the Close or Exit command, as shown in Figure 11.25.

You can also include Import File and similar commands in the window of the open object. Treat importing a file into the window of the open embedded object the same as any change to the object.

If it has file operations, such as New or Open, remove these in the resulting window or replace them with commands, such as Import, to avoid severing the object’s connection with its container. The objective is to present a consistent conceptual model; the object in the opened window is always the same as the one in the container. You can replace the Save As command with a Save Copy As command that displays the Save As dialog box, but unlike Save As, Save Copy As does not make the copied file the active file.

When the user opens an object, it is the selected object in the container; however, the user can change the selection in the container afterwards. Like any selected OLE embedded object, the container supplies the appropriate selection appearance together with the open appearance, as shown in Figure 11.27.



U.S. Compact Disc vs. LP Sales (\$)			
	1983	1987	1991
CDs	5,345K	18,662K	32,657K
LPs	31,638K	26,571K	17,429K
Total	37,983K	45,233K	50,086K

Figure 11.27 A selected open OLE embedded object