

Welcome inside. This chapter begins the discussion of programming for Macintosh computers by describing the general organization of the *Macintosh system software*, a collection of routines that you'll use to simplify your development of Macintosh applications. The system software provides, among other things, routines that you can use to create and manage the essential parts of your application's user interface. This chapter illustrates the organization and content of the system software by dissecting a very simple sample application.

## Getting Started

Let's begin by looking at the source code for a simple application. Consider Listing 1-1.

**Listing 1-1** A simple Macintosh application

```
PROGRAM GreetMe;
VAR
    gWindow:    WindowPtr;           {pointer to a window record}
    gString:    Str255;              {the string to display}
    gRect:      Rect;                {the window's rectangle}
BEGIN
    InitGraf(@thePort);              {initialize QuickDraw}
    InitFonts;                        {initialize Font Manager}
    InitWindows;                      {initialize Window Manager}
    InitCursor;                      {initialize the cursor to an arrow}

                                {set the position of the window}
    SetRect(gRect, 100, 100, 400, 200);
    gString := 'Hello, world!';      {set the greeting to be displayed}

                                {create a window}
    gWindow := NewWindow(NIL, gRect, '', TRUE, dBoxProc, WindowPtr(-1),
                        FALSE, 0);
    SetPort(gWindow);                {set the current drawing port}
    WITH gWindow^.portRect DO        {set the position of the pen}
        MoveTo(((right - left) DIV 2) - (StringWidth(gString) DIV 2),
                (bottom - top) DIV 2);
    TextFont(systemFont);            {set the font}
    DrawString(gString);              {draw the string}

    REPEAT                           {loop until the mouse button is pressed}
    UNTIL Button;
END.
```

## Introduction

The application GreetMe defined by Listing 1-1 simply displays the window shown in Figure 1-1 and exits as soon as the user presses the mouse button.

**Figure 1-1** The window created by the simple application



This application is remarkably simple, but also quite revealing about some important aspects of Macintosh programming. Consider the call that creates the window in which the greeting is drawn:

```
gWindow := NewWindow(NIL, gRect, '', TRUE, dBoxProc,
                    WindowPtr(-1), FALSE, 0);
```

This call to the `NewWindow` function creates a window at the specified location in front of any existing windows on the screen. The `NewWindow` function is a good example of the kind of routines provided by the system software. These routines greatly simplify the creation of the standard “look and feel” of Macintosh applications. By using these routines, you can ensure that your application conforms as closely as possible to the standard Macintosh user interface and hence that users find your application easy to learn and use.

Let’s take a closer look at the call to `NewWindow`. The `NewWindow` function requires eight parameters, whose meanings are described in Table 1-1.

**Table 1-1** Parameters passed to `NewWindow` in Listing 1-1

Parameter	Meaning
<code>NIL</code>	The address of a window record, a data structure that contains information about the new window. Specifying <code>NIL</code> as the address of this structure instructs the system software to allocate that required storage itself.
<code>gRect</code>	The window’s bounding rectangle. This is the rectangle that encloses the new window. The values of the desired rectangle are specified by the previous call to <code>SetRect</code> , which defines the upper-left and lower-right corners of the rectangle.
<code>''</code>	The window’s title. The new window has no title bar, so this parameter is specified as the empty string.

**Table 1-1** Parameters passed to `NewWindow` in Listing 1-1 (continued)

Parameter	Meaning
<code>TRUE</code>	An indication of whether the new window should initially be visible or not. This parameter is set to <code>TRUE</code> to indicate that the window is indeed to be made visible.
<code>dBoxProc</code>	The type of window you want to create. The Macintosh user interface includes a great variety of window types for different purposes. For present purposes, the standard modal dialog box is appropriate. The constant <code>dBoxProc</code> identifies that type of window.
<code>WindowPtr(-1)</code>	The new window's initial plane (or layer) relative to any other existing windows. This parameter is a window pointer to the window behind which you want the new window to appear. The system software recognizes two special values here. If you pass <code>NIL</code> in this parameter, the new window appears <i>behind</i> all other windows. If you pass <code>-1</code> , the new window appears <i>in front of</i> all other windows. Because the <code>NewWindow</code> function expects a window pointer in this parameter, you need to typecast the special value <code>-1</code> as <code>WindowPtr(-1)</code> .
<code>FALSE</code>	An indication of whether the window has a close box or not. This parameter is set to <code>FALSE</code> to indicate that no close box is desired.
<code>0</code>	An application-specific reference number. This number is put into a particular field of the new window record, and can be useful to you if the window has specific data associated with it. Because there is no such data associated with this window, this parameter is set to <code>0</code> .

The `NewWindow` function returns a window pointer, which is the address in memory of a window record. The window record contains important information about the window (such as its current location on the screen and the current font and size of text that is to be drawn in the window). When you call a system software routine to perform some operation on a window, you'll typically pass a window pointer as a parameter to that routine. For example, in Listing 1-1, the window pointer is passed to the `SetPort` procedure to set the new window as the current drawing window.

**IMPORTANT**

You need to call `SetPort` before you do anything at all that affects the contents of a window, such as drawing graphics or text in the window, or even just erasing the contents of the window. ▲

Another notable element of Listing 1-1 is the `DrawString` procedure, which draws the specified string in the current font at the current drawing location. By default, the current drawing location in a new window is the upper-left corner. In this case, remaining at that location would make the greeting unreadable, because `DrawString` uses the vertical coordinate of the current point as the baseline of the text to be printed. Instead, `GreetMe` calls the `MoveTo` procedure to move the current pen location to a point that centers the greeting in the window:

## Introduction

```
WITH gWindow^.portRect DO           {set the position of the pen}
  MoveTo(((right - left) DIV 2) - (StringWidth(gString) DIV 2),
        (bottom - top) DIV 2);
```

The `MoveTo` procedure requires 2 parameters, the horizontal and vertical coordinates within the window of the new drawing position. The origin—point (0,0)—of a window is at its upper left corner. Horizontal coordinates increase as you move from left to right, and vertical coordinates increase as you move from top to bottom. The coordinates passed to `MoveTo` are calculated from the left, top, bottom, and right coordinates of the window (obtained from the `portRect` field of the window record).

## The Macintosh System Software

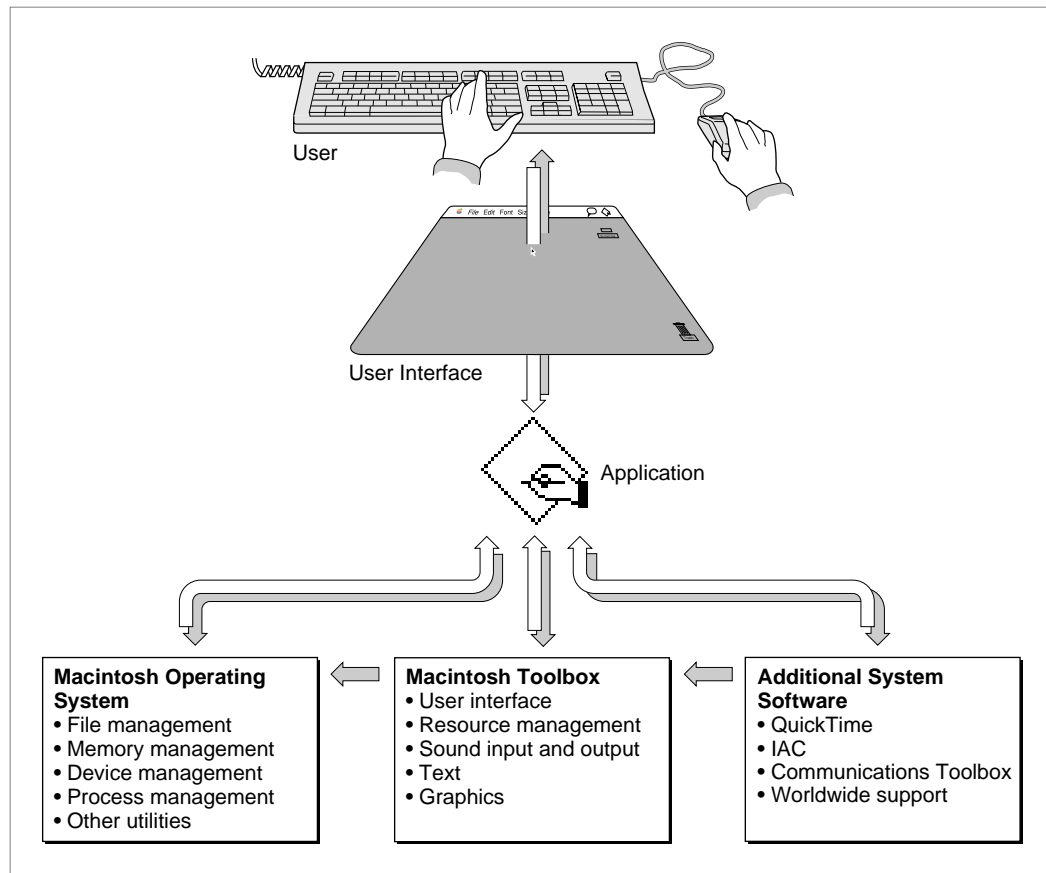
---

The richness of the Macintosh user interface is closely matched by the richness of the Macintosh system software routines. There are currently several thousand system software routines that, like `NewWindow`, are available to application developers for use in writing applications for the Macintosh operating system. Fortunately, you don't need to learn all of those routines before starting to develop applications for the Macintosh. The sample application defined in Listing 1-1 uses only a dozen or so system software routines. A typical application might directly call a few hundred of these routines.

The entire collection of system software routines is logically divided into functional groups—usually known as *managers*—that handle specific tasks or user interface elements. For example, the `NewWindow` routine belongs to the Window Manager, the part of the Macintosh system software that allows you to create, move, hide, resize, and otherwise manipulate windows. Similarly, the parts of the system software that allow you to create and manipulate menus belong to the Menu Manager.

Your application calls system software routines to create standard user interface elements and to coordinate its actions with other open applications. The main other application that your application needs to work with is the *Finder*, which is responsible for keeping track of files and managing the user's desktop. Usually, the user launches your application by double-clicking its icon (or one of its document's icons) in a Finder window. The Finder isn't really part of the Macintosh system software, but it is such an important piece of the Macintosh graphic user interface that it's sometimes difficult to tell where the Finder ends and the systems software begins. In fact, the system software provides a set of routines—known as the Finder Interface—that you can use to interact with the Finder.

As shown in Figure 1-2, most of the system software routines are part of either the Macintosh Operating System or the Macintosh Toolbox.

**Figure 1-2** Overview of the system software

This section describes the division of the Macintosh system software into its logical parts. Understanding this division of system software into managers and other units is essential to understanding Macintosh programming, as well as the general organization of *Inside Macintosh*.

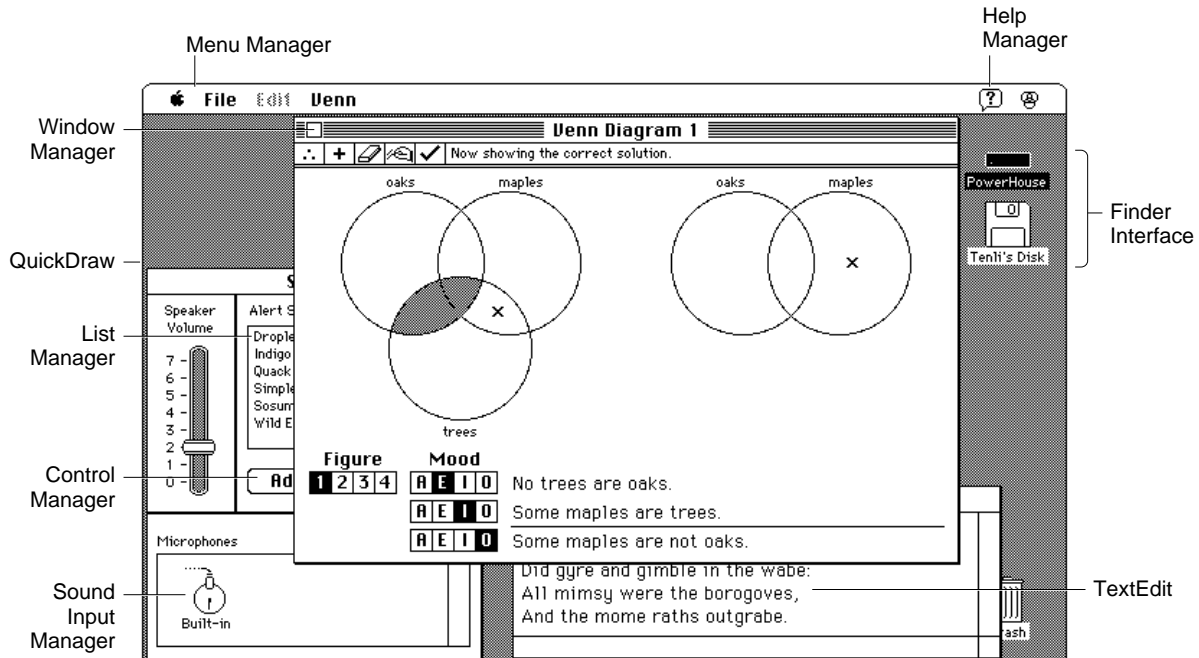
## The Macintosh Toolbox

The system software routines used in Listing 1-1 allow you to manage elements of the Macintosh user interface. These parts of the system software belong to the **Macintosh Toolbox** (sometimes also called the **Macintosh User Interface Toolbox**). By offering a common set of routines that every application can call to implement the user interface, the Toolbox not only ensures familiarity and consistency for the user, but also helps reduce your application's code size and development time. At the same time, the Toolbox offers a great deal of flexibility; your application can, whenever appropriate, use its own code instead of Toolbox routines, and it can define its own types of windows, menus, and controls. In general, however, you should use the Toolbox routines to maximize compatibility with present and future versions of the system software.

## Introduction

Figure 1-3 illustrates the main parts of the Macintosh Toolbox.

**Figure 1-3** Parts of the Macintosh Toolbox

**Note**

For historical reasons, some collections of system software routines are referred to as packages. One example is the Standard File Package (which allows you to present the standard file opening and saving dialog boxes). In general, the distinction between managers and packages is unimportant. Accordingly, the new *Inside Macintosh* has, whenever appropriate, adopted the practice of renaming packages as managers. For instance, the Disk Initialization Manager (described in the book *Inside Macintosh: Files*) was previously known as the Disk Initialization Package. ♦

Consider the first few lines of Listing 1-1 on page 3:

```
InitGraf(@thePort);           {initialize QuickDraw}
InitFonts;                     {initialize Font Manager}
InitWindows;                   {initialize Window Manager}
InitCursor;                     {initialize cursor to arrow}
```

## Introduction

These lines of code perform standard initialization of some essential Toolbox managers. You need to initialize these managers in order to set up the drawing environment for your application and to prepare parts of the Toolbox for further use. The `InitGraf` procedure initializes *QuickDraw*, the part of the Macintosh Toolbox that handles drawing and other graphics operations. Because the Macintosh user interface is largely a graphic user interface, QuickDraw routines are called by virtually all the other Toolbox managers. For example, the Window Manager calls QuickDraw to draw the window frame and any other required parts of a window (for instance, the title bar). For this reason, you need to initialize QuickDraw before you initialize the other main Toolbox Managers.

**Note**

QuickDraw gets its name from the fact that it's designed to perform basic graphics operations exceptionally fast. This is important for a user interface that relies so heavily on graphics. ♦

Your application will also call QuickDraw directly, usually to draw inside a window or to set up constructs (like rectangles) that you'll need when making other Toolbox calls. QuickDraw provides a rich array of routines that let you

- change, hide, and display the cursor
- manipulate the current drawing port
- set characteristics of the drawing pen
- draw text
- manage colors
- define rectangles, ovals, arcs, and other basic geometric shapes
- define arbitrarily shaped regions
- perform operations on shapes and regions

The essential thing to keep in mind is that if you can see something on the screen, then QuickDraw is lurking somewhere behind it, either directly (you drew it there) or indirectly (you called a Toolbox routine that called QuickDraw to draw it there).

The `InitFonts` procedure initializes the Font Manager, which supports the use of various character fonts when you draw text with QuickDraw. The `TextFont` routine sets the current font to that whose font number is passed as a parameter. `GreetMe` passes the special constant `systemFont`, which requests the font used by the system (for drawing menu titles and commands in menus, for example).

The `InitWindows` procedure initializes the Window Manager, and the `InitCursor` procedure (which belongs to QuickDraw) sets the cursor to the standard arrow cursor. Every application needs to call these routines before creating windows or handling any user actions.

## Introduction

Notice that Figure 1-3 depicts a number of other Toolbox managers that are not used by GreetMe. You'll encounter many of these as you progress through this book. For now, take a look at Table 1-2 for a brief description of the most commonly used Macintosh Toolbox managers.

**Table 1-2** The Macintosh Toolbox

Manager	Description
QuickDraw	Performs all screen display operations, including all drawing of graphics and text.
Window Manager	Allows you to create and manage windows of various types.
Dialog Manager	Allows you to create and manage dialog boxes, which are special kinds of windows. Typically you'll use dialog boxes to alert the user to unusual situations or to solicit information from the user.
Control Manager	Allows you to create and manage controls, such as buttons, radio buttons, checkboxes, pop-up menus, scroll bars, and application-defined controls.
Menu Manager	Allows you to create and manage your application's menu bar and the menus it contains. Also handles the drawing of menus and user actions within a menu.
Event Manager	Reports to your application events describing user actions and changes in the processing status of your application. Also allows you to communicate with other applications.
TextEdit	Provides simple text-formatting and text-editing capabilities, such as text input, selection, cutting, and pasting. Applications that are not primarily concerned with text processing can use TextEdit to handle most text manipulation.
Resource Manager	Allows your application to read and write resources. Any static data (such as menus, cursors, and windows) used by your application can usefully be stored as a resource. The system software provides a number of standard resources, and your application can define its own custom resources.
Finder Interface	Allows your application to interact with the Finder, the application that helps keep track of files and manages the user's desktop display.
Scrap Manager	Allows your application to support cutting and pasting of information among applications.
Standard File Package	Provides the standard dialog boxes that allow the user to select a file to open or a location and name for a file to be saved.
Help Manager	Allows your application to provide Balloon Help on-line assistance, information that describes the actions, behaviors, and properties of elements of your application.

**Table 1-2** The Macintosh Toolbox (continued)

Manager	Description
List Manager	Allows your application to create lists of items.
Sound Manager	Provides sound output capabilities.
Sound Input Manager	Provides sound input capabilities for Macintosh computers equipped with a sound input device such as a microphone.

## The Macintosh Operating System

The Macintosh Operating System provides routines that allow you to perform basic low-level tasks such as file input and output, memory management, and process and device control. The Macintosh Toolbox is a level above the Operating System and, as you've seen, provides routines that help you implement the standard Macintosh user interface for your application. The Toolbox calls the Operating System to do low-level operations, and you'll also need to call the Operating System directly yourself.

The Macintosh Toolbox allows you to create and manage parts of your application's user interface, and in some sense mediates your application and the user. By contrast, the Macintosh Operating System essentially mediates your application and the Macintosh hardware. For example, you'll read and write files not by reading data directly from the medium on which they are stored, but rather by calling appropriate File Manager routines. The File Manager locates the desired data within the logical hierarchical structure of files and directories that it manages; then it calls another part of the Operating System, the Device Manager, to read or write the data on the actual physical device. The File Manager and the Device Manager thereby insulate your application from the low-level details of interacting with the available data-storage hardware.

Similarly, the Memory Manager helps you allocate and dispose of memory within your application's logical address space. The Memory Manager takes care of mapping that logical address space onto the physical address space provided by the available RAM. It also helps manage your application's memory by moving allocated blocks of memory when necessary to create space for new blocks you want to allocate. Table 1-3 briefly describes the main parts of the Macintosh Operating System.

**Table 1-3** The Macintosh Operating System

Manager	Description
Process Manager	Handles the launching, scheduling, and termination of applications. Also provides information about open processes.
Memory Manager	Manages the dynamic allocation and releasing of memory in your application's memory partition.

*continued*

## Introduction

**Table 1-3** The Macintosh Operating System (continued)

Manager	Description
Virtual Memory Manager	Provides virtual memory services (the ability to have a logical address space that is larger than the total amount of available RAM).
File Manager	Provides access to the file system; allows applications to create, open, read, write, and close files.
Alias Manager	Helps you locate specified files, directories, or volumes.
Disk Initialization Manager	Manages the process of initializing disks.
Device Manager	Provides input from and output to hardware devices attached to the computer.
SCSI Manager	Controls the exchange of information between a Macintosh computer and peripheral devices attached through the Small Computer Standard Interface (SCSI).
Time Manager	Allows you to execute a routine periodically or after a specified time delay.
Vertical Retrace Manager	Allows you to synchronize the execution of a routine with the redrawing of the screen.
Shutdown Manager	Allows you to execute a routine while the computer is shutting down or restarting.

## Additional System Software Services

The Macintosh system software includes a number of other parts that don't historically belong to either the Macintosh Toolbox or the Macintosh Operating System. The system software provides an extremely powerful set of services you can use to handle text and to support the varying text-handling requirements of different languages and writing systems. Other system software components include the interapplication communications architecture, QuickTime, and the Communications Toolbox.

### Text Handling

Text handling on the Macintosh has two basic aspects that make it so powerful. First, it is fundamentally graphic; text is drawn as a sequence of graphic elements; therefore the full power and flexibility of the Macintosh graphic interface is available for drawing text in sophisticated ways.

Second, text handling is designed to function properly across multiple languages and writing systems. As you develop applications for worldwide markets, you need to consider differences in scripts, languages, and regions. The Macintosh system software presents one of the most flexible architectures for developing applications that can support more than one script.

A *script*, such as Roman, Kanji, or Arabic, is a writing system for a human language such as English, Japanese, or Arabic. Scripts have different characteristics; for example, they can differ in the direction in which their characters and lines run and in the number of characters in their character sets. The way in which you need to input, display, render, and edit text may change depending on the script in use.

A Macintosh *script system* is a set of system resources that support text input, manipulation, and display for a given writing system. The *Macintosh script management system* consists of system software managers and the WorldScript extensions, which together give your application the power to create and work with text of any script system. These are the essential text-handling managers:

- **QuickDraw** is the graphics manager of Macintosh system software. Your application makes QuickDraw calls to write text to the screen or to a printer. When QuickDraw draws text, it draws it according to the settings of the current window's graphics port record, which includes the location information and complete font information. QuickDraw can draw text of any script system. Figure 1-4 shows some of QuickDraw's text-drawing capabilities.

**Figure 1-4** A multiscript line of text drawn by QuickDraw

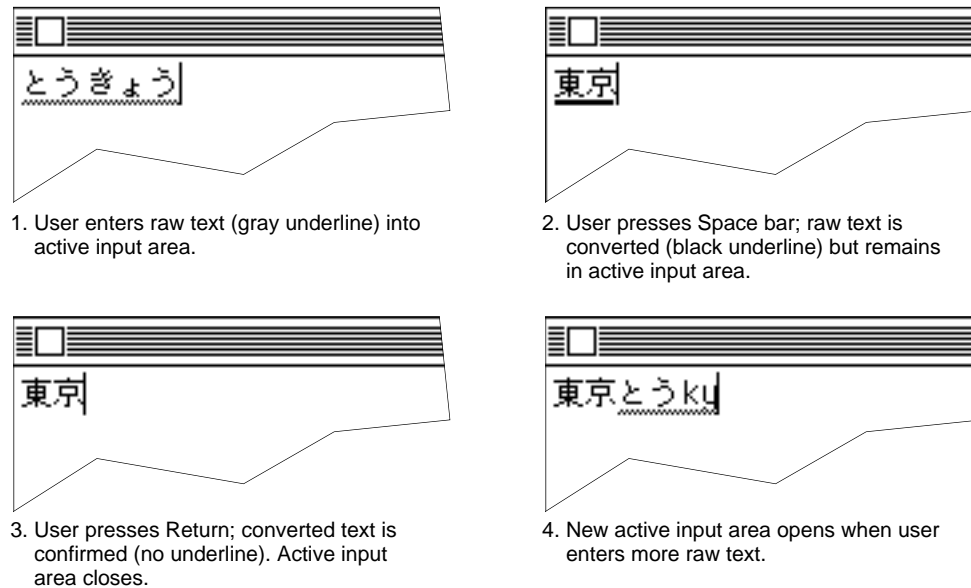


- The **Font Manager** supports QuickDraw by providing the fonts that QuickDraw needs, in the typefaces, sizes, and styles that QuickDraw requests. The Font Manager keeps track of all fonts available to an application, and supports fonts for all script systems.
- The **Text Utilities** are an integrated collection of routines for performing a variety of operations on text, ranging from sorting strings to formatting dates and times to finding word breaks. The Text Utilities work in conjunction with the Macintosh script management system and can take into account the differences in text handling among script systems. If you use these routines, you can handle text operations in a manner that is transportable to different parts of the world.
- The **Script Manager** is at the center of the Macintosh script management system. It initializes script systems, maintains important data structures, supports switching text input among different script systems, and provides several text-manipulation services.
- The **Text Services Manager** supports *text service components* such as input methods. If your application uses the Text Services Manager, it can support the special kinds of text input needed for 2-byte script systems such as Japanese, Chinese, and Korean.

## Introduction

Figure 1-5 shows how you can use the Text Services Manager to convert Japanese text.

**Figure 1-5** Input and conversion of Japanese text using the Text Services Manager



You can use the script management system to achieve any level of text-handling sophistication, from simple display of static text in one language to highly sophisticated multilanguage word processing and page layout. The simplest way to achieve basic worldwide flexibility in text handling is to use *TextEdit*, which provides simple text-handling capabilities for text of any script system, including multiscript text. TextEdit automatically handles text with more than one script, style, and direction. For example, TextEdit supports mixing English text (a left-to-right directional script) with Arabic text (a right-to-left directional script) in the same line (as you saw in Figure 1-4).

#### Note

For complete information on text handling, including multiscript text handling, see *Inside Macintosh: Text*. For information on individual script systems and how to localize your software for markets around the world, see *Guide to Macintosh Software Localization*. ♦

## Interapplication Communication

The *interapplication communications (IAC) architecture* provides a standard and extensible mechanism for communication among Macintosh applications. The IAC architecture includes these main parts:

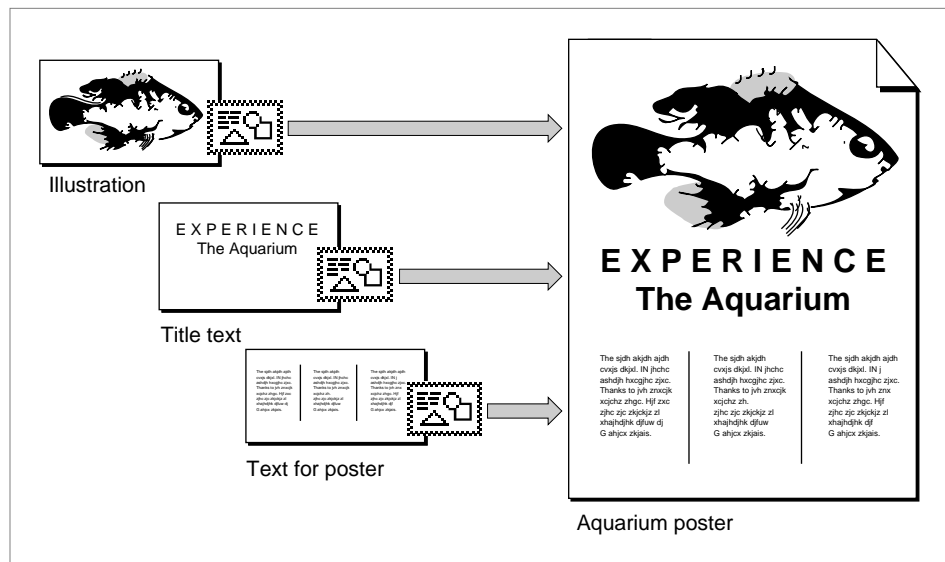
- The *Edition Manager* allows applications to automate copy and paste operations between applications, so that data can be shared dynamically.

- The *Apple Event Manager* allows applications to send and respond to Apple events.
- The *Event Manager* allows applications to send and respond to high-level events other than Apple events.
- The *Program-to-Program Communications (PPC) Toolbox* allows applications to exchange blocks of data with each other by reading and writing low-level message blocks. It also provides a standard user interface that allows a user working in one application to select another application with which to exchange data.

The parts of the IAC architecture depend upon each other in fairly straightforward ways. The Edition Manager uses the services of the Apple Event Manager to support dynamic data sharing. The Apple Event Manager, in turn, relies on the Event Manager to send Apple events as high-level events, and the Event Manager uses the services of the PPC Toolbox.

If you want your application to exchange data with another application, you'll probably use either the Edition Manager or the Apple Event Manager. The Edition Manager allows users to copy data from one application's document to another application's document, updating the information automatically when the data in the original document changes. Figure 1-6 shows how you can use the Edition Manager to create a poster whose elements (an illustration, a title, and some text) all originate in documents created by other applications. If, for example, the user changes the illustration in the original document, the copy of that illustration in the poster could be updated automatically.

**Figure 1-6** Sharing dynamic data with other applications



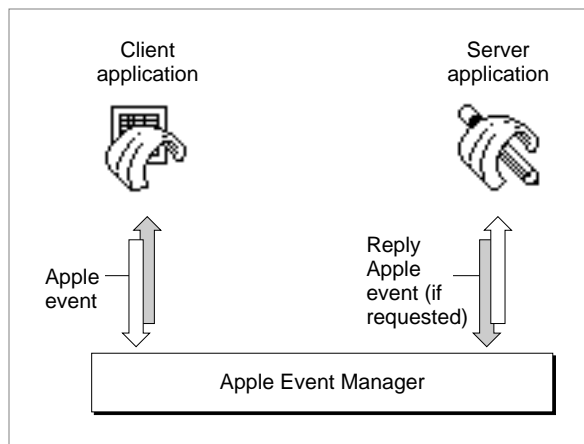
## Introduction

The Apple Event Manager allows you to send and receive Apple events, which are high-level events that conform to the Apple Event Interprocess Messaging Protocol. The *Apple Event Registry: Standard Suites* describes a standard vocabulary of Apple events that you can use to communicate with other open applications. Typically you use Apple events to request services and information from other applications, or to provide services and information in response to such requests.

Communication between two applications that support Apple events is initiated by a client application, which sends an Apple event to request a service or information. For example, a client application might request services such as printing specific files, checking the spelling of a list of words, or performing a numerical calculation; or it might request information, such as one customer's address or a list of names and addresses of all customers living in Ohio. The application providing the service or the requested information is called a server application. The client and server applications can reside on the same local computer or on remote computers connected to a network.

Figure 1-7 shows the relationships among a client application, the Apple Event Manager, and a server application. The client application uses Apple Event Manager routines to create and send the Apple event, and the server application uses Apple Event Manager routines to interpret the Apple event and respond appropriately. If the client application so requests, the server application sends back a reply Apple event.

**Figure 1-7** Sending and responding to Apple events



As you might imagine, there are many predefined kinds of Apple events, corresponding to the many services one application might request of another. Apple events are grouped into standard suites or groups of related events. Usually, you implement all the events in a given suite at the same time. The standard Apple event suites include the following:

## Introduction

- The *Required suite* consists of four basic Apple events that your application must support if it supports any Apple events at all. These events are Open Documents, Open Application, Print Documents, and Quit Application. The Finder uses these events for launching and terminating applications.
- The *Core suite* consists of the basic Apple events that nearly all applications use to communicate, including Get Data, Set Data, Move, Delete, and Save. You should support all the Apple events in the Core suite that make sense for your application.
- A *functional-area suite* consists of a group of Apple events that support a related functional area. One example of a functional area is the Text suite, which includes events related to text processing.

If an Apple event is one of these standard events, the client application can construct the event and the server application can interpret it according to the standard definition for that event. To ensure that your application can respond to Apple events sent by other applications, you should support the standard Apple events that are appropriate for your application.

**Note**

See the book *Inside Macintosh: Interapplication Communication* for complete details about the interapplication communications architecture. ♦

## QuickTime

---

*QuickTime* is a collection of managers and other system software components that allow your application to control time-based data. QuickTime allows you to integrate time-based data (such as video clips, animation sequences, sound sequences, or time-indexed scientific data) into your application and to let users manipulate it in the same easy, intuitive way that they manipulate other elements of the Macintosh user interface. With QuickTime, your application can allow users to display, edit, copy, and paste time-based data much as they do text and graphics.

A movie is a collection of one or more streams of data, called tracks. Each track represents a stream of data of a particular type, such as video, sound, still images, or animation. Depending on the way the tracks are defined, one or more tracks can be active at certain times while the movie is playing.

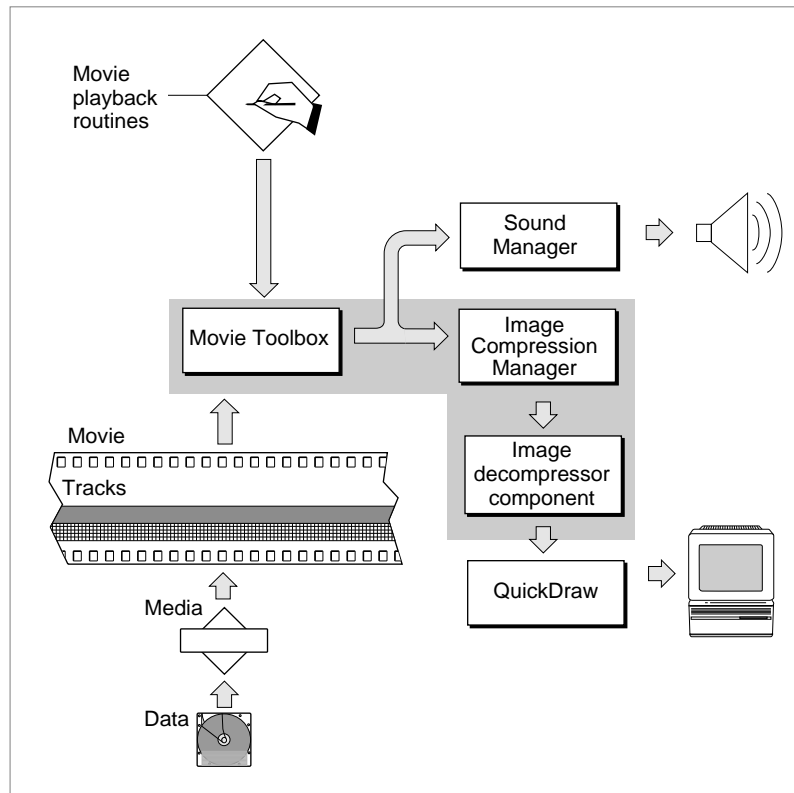
QuickTime consists mainly of these pieces:

- the Movie Toolbox
- the Image Compression Manager
- a set of predefined components

## Introduction

Many applications that incorporate QuickTime capabilities are interested only in playing movies. To do so, they call the Movie Toolbox, which provides routines that allow you to store, retrieve, and manipulate time-based data stored in QuickTime movies. Figure 1-8 illustrates the relationship between the various QuickTime managers and components.

**Figure 1-8** Playing a QuickTime movie

**Note**

See the books *Inside Macintosh: QuickTime* and *Inside Macintosh: QuickTime Components* for complete details about QuickTime. ♦

## Communications Toolbox

The **Communications Toolbox** is a collection of system software managers that you can use to provide your application with basic networking and communications services. You're likely to use the Communications Toolbox only if your application is specifically concerned with communication between computers. Examples of such applications include telecommunications packages and electronic bulletin board applications. By using the Communications Toolbox, you can insulate your application from the details of the actual physical connection between your computer and the remote computer.

## Introduction

The Communications Toolbox consists of four managers:

- The Connection Manager, which you can use to create and maintain a network connection.
- The Terminal Manager, which you can use to emulate a particular terminal during a network connection.
- The File Transfer Manager, which you can use to transfer files between your computer and the remote computer to which you are connected.
- The Communications Resource Manager, which you can use to register and keep track of communications resources.

**Note**

For complete information about the Communications Toolbox, see the book *Inside the Macintosh Communications Toolbox*. ♦

## System Software Routines

---

By now, you might be wondering how these various system software routines are made available to your application. In traditional programming environments, you gain access to such special routines by linking a subroutine library—which contains the actual executable code of those routines—to your application. The code of the special routine is contained in your application, just like the code of any application-defined routine.

One main drawback of such an approach is that it tends to result in very large applications. As you might imagine, the code comprising the thousands of system software routines takes up quite a bit of space. It would be impractical to link all that code, or whatever subset of it an application actually used, to each application.

Another important drawback of the traditional approach is the difficulty of revising system software routines to provide new capabilities or to fix bugs. You would need to obtain a new subroutine library and then rebuild your application so that the new code is included in it.

The original Macintosh system software circumvented these problems by adopting a fairly novel approach. The software routines that make up the Macintosh Toolbox and the Macintosh Operating System reside mainly in *read-only memory (ROM)*, provided by special chips contained in every Macintosh computer. When your application calls a Toolbox routine like `NewWindow`, the Operating System intercepts the call and executes the appropriate code contained in ROM.

This mechanism provides a simple way for the Operating System to substitute the code that is executed in response to a particular system software routine. Instead of executing the ROM-based code for some routine, the Operating System might choose to load some substitute code into the computer's *random-access memory (RAM)*; then, when your application calls the routine in question, the Operating System intercepts the call and executes that RAM-based code.

## Introduction

RAM-based code that substitutes for ROM-based code is called a *patch*. Patches are usually stored in the *System file*, located in the System Folder. The System file also contains collections of static data, known as resources, that applications can use to help present the standard Macintosh user interface.

The System file can also contain system software components that are not in a computer's ROM. To make one of these components available to your application, the Operating System simply loads it into RAM. This is like a patch, except that the new routines aren't replacing any existing ROM routines. Originally these sorts of RAM-based system software components were called *packages*; they were read into RAM only when some application called any one of the routines contained in them. However, because some of these packages have been included in later revisions of the ROM, the distinction between managers and packages has faded with time.

The current method for adding capabilities to the system software is to include the executable code of the new routines as a *system extension*. Extensions are stored in a special location (namely, in the Extensions folder in the System Folder) and are loaded into memory at system startup time. QuickTime, for example, is currently distributed as an extension.

When your application calls a system software routine, it doesn't matter, in general, whether the code that is executed in response resides in ROM, is a patch in RAM loaded from the System file, or is part of a RAM-based extension. It is, however, important that the appropriate code exist in at least one of these locations, because your application will crash if you attempt to call a routine that isn't defined anywhere. So, especially for code contained in extensions, you'll need to make sure that the code is present in the current operating environment before trying to call it. You can use the `Gestalt` function to determine whether a particular part of system software is available. For details on calling `Gestalt`, see the chapter "Gestalt Manager" in *Inside Macintosh: Operating System Utilities*.

There is one further twist in this picture that is worth mentioning. Some routines that are declared in your development system's header files are provided by the development system itself, not by the system software. These routines, known as *glue routines* (or just *glue*), are constructed by modifying available system software routines in some way. Consider the Memory Manager function `NewHandle`, which allocates a new relocatable block of memory. A call to `NewHandle` compiles into an executable instruction word. When that instruction is executed, the ROM code (or its RAM patch, if one exists) reads several of the bits in that word to determine exactly what to do. If, for instance, bit 9 of the instruction word is set, the ROM code allocates a block of the requested size and then clears all the bytes in that block to 0.

If you're programming in assembly language, you can set the bits of an instruction word directly. However, if you're programming in a high-level language like Pascal, you can't do that. Instead, you need to call a glue routine, in this case `NewHandleClear`, that takes care of calling `NewHandle` and setting the appropriate bits in the instruction word. Essentially, `NewHandleClear` is nothing but `NewHandle` together with some assembly-language code to set a bit in the instruction word. This translation is handled automatically by your development system at the time your application is compiled.

You'll encounter several other kinds of glue routines. Some glue routines translate high-level routines into low-level routines. Most of the high-level File Manager routines are of this variety. There is, for example, no code in ROM or the System file corresponding to the `FSpCreate` function. Instead, calling `FSpCreate` invokes some glue code that creates a parameter block, fills out some of the fields appropriately, and then passes that parameter block to the low-level function `PBHCreate`.

Some other glue routines are pure assembly-language instructions which don't call any system software routines. You might use glue like this to move a function result or other data from a register onto the stack.

You don't usually need to know whether a particular routine is implemented as glue code, except when you're doing low-level assembly-language debugging. For the time being, you can consider all the routines defined in *Inside Macintosh* as part of the Macintosh system software.

## The Sample Application

---

The remainder of this book illustrates how to write a Macintosh application by gradually dissecting the source code of a very simple sample application, called Venn Diagrammer. This application allows the user to use Venn diagrams as a method of determining whether a given syllogism is valid (that is, whether the conclusion must be true if both premises are true). This section briefly describes the operation of the Venn Diagrammer application.

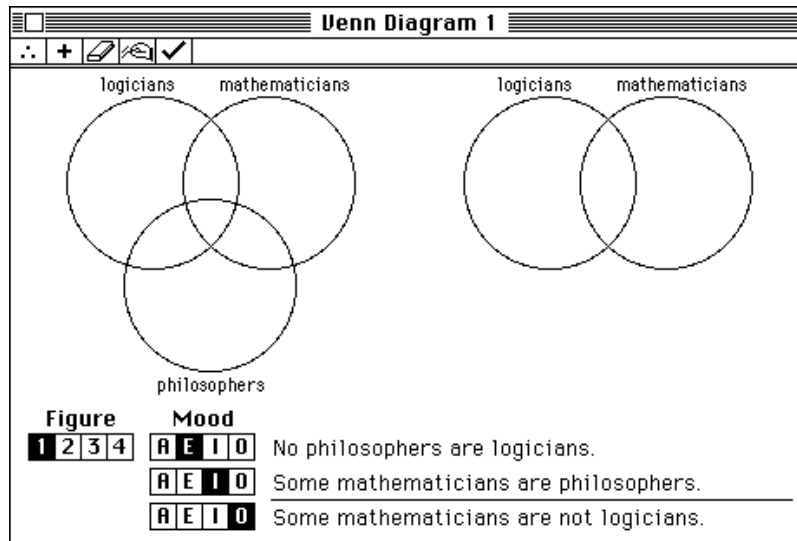
### IMPORTANT

The account of syllogisms and Venn diagrams given here is inadequate for a full understanding of these topics. Most programmers, however, have encountered Venn diagrams at some point in their lives. For a more complete account, consult a good textbook on introductory logic. ▲

## Introduction

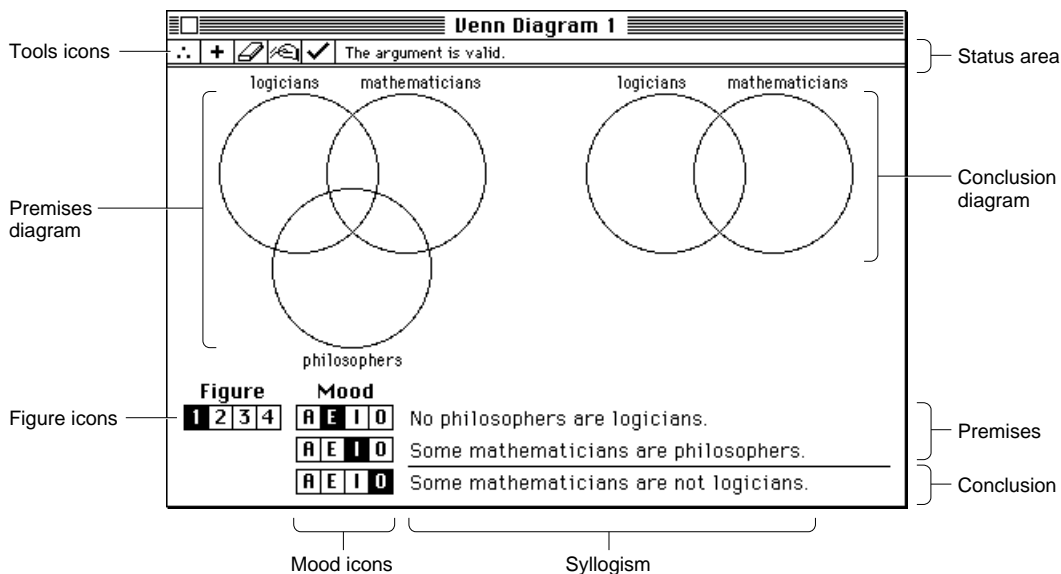
When the user launches the Venn Diagrammer application, it opens a Venn diagram window, shown in Figure 1-9.

**Figure 1-9** A typical Venn diagram window



This window contains a number of distinct parts, shown in Figure 1-10.

**Figure 1-10** The parts of a Venn diagram window



## Introduction

This window is designed to let the user select a syllogism and then assess the validity of the syllogism by appropriately modifying the Venn diagram (the five overlapping circles). The user graphs the information contained in the two premises in the three circles on the left and the information in the conclusion in the two circles on the right.

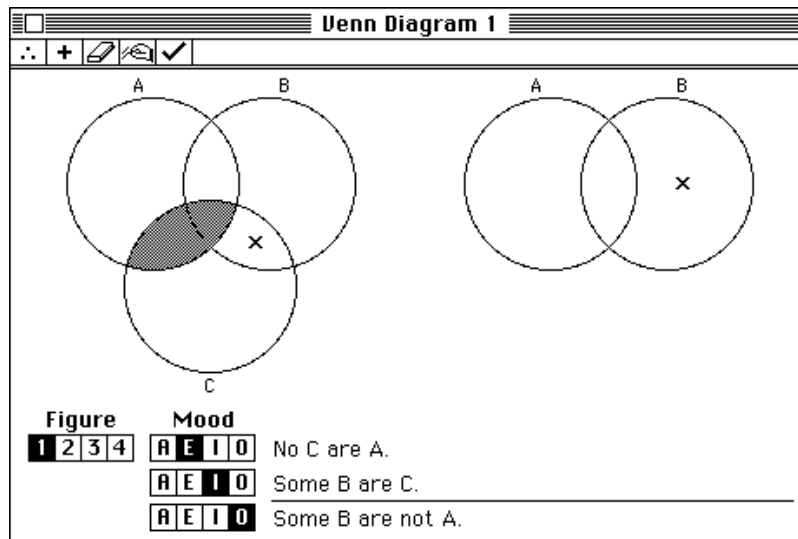
As you can see, a syllogism is an argument containing two premises and one conclusion. These three statements must each be of one of four specific forms, known as the statement's mood. The four moods are often designated by the letters A, E, I, and O, as follows:

- A All philosophers are logicians.
- E No philosophers are logicians.
- I Some philosophers are logicians.
- O Some philosophers are not logicians.

Syllogisms are further classified by figure, which determines the order of the terms in the two premises. A syllogism is completely determined by the three terms involved, the moods of the three statements, and the figure.

The user can graph the information in a syllogism by clicking in the overlapping regions in the circles. If a region is white, nothing is known about the region. If the region is shaded, it's known that there is nothing in that region (that is, the region is empty). Finally, if an X appears in the region, it's known that there is something in that region. A correctly graphed syllogism is shown in Figure 1-11.

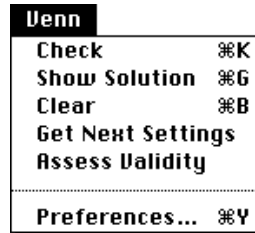
**Figure 1-11** A correctly constructed Venn diagram



## Introduction

At the top of the window, just below the title bar, are a set of tool icons and an empty status area. The tool icons allow the user to perform various operations on the diagram without having to move out of the window. For instance, clicking the tool in the middle (the eraser) clears the Venn diagram. These same operations can also be invoked using the Venn menu, as shown in Figure 1-12.

**Figure 1-12** The Venn menu



The Venn Diagrammer application displays information in the window's status area. For example, if the user clicks the leftmost tool icon (or chooses the Assess Validity menu command), the application determines whether the currently displayed syllogism is valid or invalid. If it's valid, the application displays the message "The argument is valid." in the status area; otherwise, it displays the message "The argument is invalid."

## Conventions for Sample Code

The sample code presented throughout this book follows a number of conventions to help you understand the code and to distinguish application-defined routines from system software routines. For the most part, the sample code listings presented throughout the *Inside Macintosh* suite of books follow these conventions as well.

- Constants defined by the Venn Diagrammer application begin with the letter *k*. For example, the number of tools in a Venn diagram window is specified by the constant `kNumTools`. There are, however, several exceptions to this rule:
  - Constants specifying resource IDs begin with the letter *r*. For example, the resource ID of the menu bar is specified by the constant `rMenuBar`.
  - Constants specifying menu resource IDs begin with the letter *m*. For example, the resource ID of the File menu is specified by the constant `mFile`.
  - Constants specifying menu commands begin with the letter *i*. For example, the number of the Quit command in the File menu is specified by the constant `iQuit`.
  - Constants specifying messages displayed to the user in a window's status area begin with the letter *e*. For example, the message "The argument is valid." is specified by the constant `eArgIsValid`.

## Introduction

- Application global variables have names beginning with the letter `g`. For example, the global variable that indicates whether the user wants to quit the application is called `gDone`. There are no exceptions to this rule.
- Application-defined routines have names beginning with either the prefix `Do` or the prefix `My`. For example, the routine that handles window updating is called `DoUpdate`. Similarly, the routine that returns a random number is called `MyRandom`. There is one exception to this rule:
  - Application-defined routines that return Boolean values have names beginning with the prefix `Is`. For example, the routine that determines whether a window is a dialog box is called `IsDialogWindow`. Several system software routines have similar-sounding names. (For instance, the Dialog Manager provides the `IsDialogEvent` routine.)
- Application-defined data structures and types have names beginning with the prefix `My`. For example, the structure that holds information about a document window is called `MyDocRec`. A pointer to a record of type `MyDocRec` is of type `MyDocRecPtr`.
- Routine parameters and local variables have names beginning with the prefix `my`. For example, many of the routines in the Venn Diagrammer application require a window pointer as one of the parameters; this parameter is usually called `myWindow`. This convention has, however, many exceptions.

**IMPORTANT**

These naming conventions are adopted in this book (and elsewhere in *Inside Macintosh*) solely for reasons of consistency and clarity. They might not be suitable for your purposes. ▲

It's worth mentioning in advance that Venn Diagrammer takes a minimalist approach to error-handling: it tries to detect any errors that might adversely affect its further processing and to work around those errors in such a way as to avoid those adverse effects. In fact, this strategy is far too simple for most applications. Your application should provide far more extensive error detection and reporting to the user. See "Handling Errors" beginning on page 176 for some further discussion of error-handling techniques.

