

Processes

Your application is usually only one of several applications that a user has open at one time. Your application must therefore share the available system resources such as the central processing unit (CPU) and the available random-access memory (RAM). The Macintosh Operating System uses a very simple and elegant method for your application to coordinate its actions with those of other open applications. The Process Manager sends events, through the Event Manager, to your application informing it of impending changes in your application's processing status. Your application needs to respond to those events in the appropriate way to ensure the smooth operation of all open applications.

This chapter describes what you need to do to ensure that your application operates smoothly in the Macintosh Operating System. It describes how your application is launched and how the Operating System controls access to the CPU and other system resources to create a cooperative multitasking environment in which your application and any other open applications execute. This environment is managed primarily by the Process Manager, which is responsible for launching processes, scheduling their use of the available system resources, and handling their termination. This chapter shows how to

- indicate the desired size of your application's memory partition
- suspend your application's execution when another application needs the CPU
- resume execution when your application regains control of the CPU
- terminate your application when the user quits or when a serious error occurs
- determine what software and hardware features are available on a particular machine

For a complete description of the cooperative multitasking environment, see the chapter "Process Manager" in *Inside Macintosh: Processes*. For a complete description of how to handle suspend and resume events, see the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

About Processes

The Macintosh Operating System, the Finder, and several other system software components work together to provide a ***multitasking environment*** in which a user can have multiple applications open at once and can switch between open applications as desired. To run in this environment, however, your application must follow certain rules governing its use of the available system resources. Because the smooth operation of all applications depends on their cooperation, this environment is known as a ***cooperative multitasking environment***.

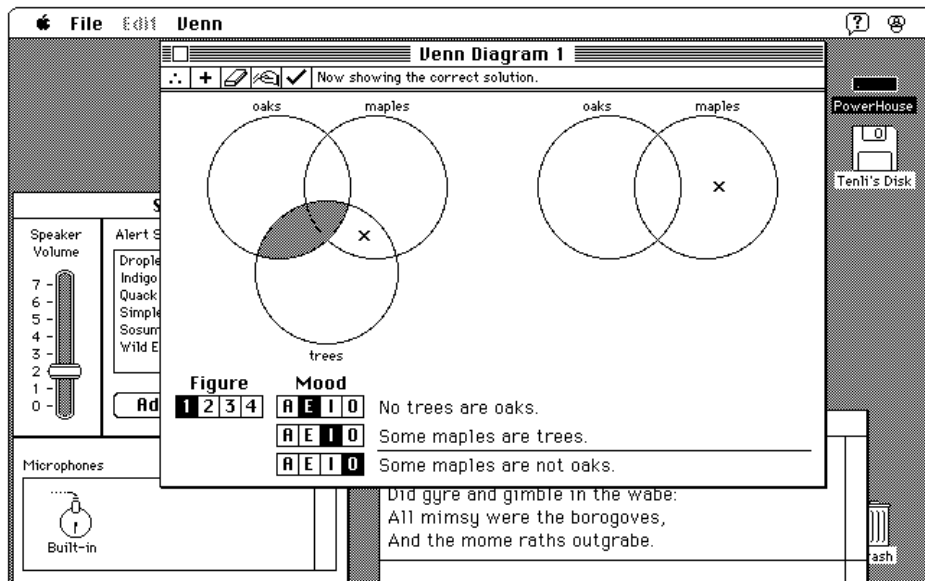
Note

The cooperative multitasking environment is available in system software versions 7.0 and later, and when the MultiFinder option is enabled in earlier system software versions. ♦

Processes

Although a number of documents and applications can be open at the same time, only one application is the active application. The *active application* is the application currently interacting with the user; its icon appears at the right side of the menu bar. The active application displays its menu bar and is responsible for highlighting the controls of its frontmost window. In Figure 9-1, Venn Diagrammer is the active application. Windows of other applications are visible on the desktop behind the frontmost window. Windows of other applications are visible on the desktop behind the frontmost window.

Figure 9-1 The desktop with several applications open



The Operating System schedules the processing of all applications and desk accessories, known collectively as *processes*. When a user opens an application, the Operating System loads the application code into memory and schedules the application to run at the next available opportunity, usually when the current process relinquishes the CPU. In most cases, the application runs immediately (or so it appears to the user).

When your application is first launched, it is the *foreground process*. Usually the foreground process has control of the CPU and other system resources, but it can agree to relinquish control of the CPU if there are no events (other than null events) pending for it. A process that is open but that isn't currently the foreground process is said to be a *background process*.

Processes

A background process can receive processing time when the foreground process makes an event call (that is, calls `WaitNextEvent` or `EventAvail`) and there are no events pending for that foreground process. The Process Manager sends a null event to the background process, thereby informing it that it is now the current process and can perform whatever background processing it desires. The background process should make an event call periodically in order to relinquish the CPU and ensure a timely return to foreground processing when necessary.

The CPU is available only to the current application, whether it is running in the foreground or the background. The application can be interrupted only by hardware interrupts, which are transparent to the application. However, to give processing time to background applications and to allow the user to interact with your application and others, you must periodically call the Event Manager's `WaitNextEvent` or `EventAvail` function to allow your application to relinquish control of the CPU for short periods. By using these event routines in your application, you allow the user to interact not only with your application but also with other applications.

The method by which the available processing time is distributed among multiple processes is known as *context switching* (or just *switching*). All switching occurs at a well-defined time, namely, when an application calls `WaitNextEvent`. When a context switch occurs, the Process Manager allocates processing time to a process other than the one that had been receiving processing time. Two types of context switching may occur: major and minor.

A *major switch* is a complete context switch: an application's windows are moved from the back to the front, or vice versa. In a major switch, two applications are involved, the one being switched to the foreground and the one being switched to the background. The Process Manager switches the A5 worlds of both applications, as well as the relevant low-memory environments. If those applications can handle suspend and resume events, they are so notified at the time that a major switch occurs.

A *minor switch* occurs when the Process Manager gives time to a background process without bringing the background process to the front. The two processes involved in a minor switch can be two background processes or a foreground process and a background process. As in a major switch, the Process Manager switches the A5 worlds and the low-memory environments of the two processes. However, the order of windows is not switched, and neither process receives either suspend or resume events.

When the frontmost window is an alert box or modal dialog box, major switching does not occur, although minor switching can. To determine whether major switching can occur, the Operating System checks (among other things) whether the window definition procedure of the frontmost window is `dBoxProc`, because the type `dBoxProc` is specifically reserved for alert boxes and modal dialog boxes. (If the frontmost window is a movable modal dialog box, major switching can still occur.)

Note

Your application can also be switched out if it calls a system software routine that internally makes an event call. For example, when your application calls `ModalDialog`, a minor switch can occur. ♦

Specifying Processing Options

To take full advantage of the cooperative multitasking environment provided by the Macintosh system software, you need to inform the Operating System about the processing capabilities and requirements of your application. You need to indicate, for example, the partition size your application needs in order to execute most effectively. You also need to indicate whether your application can do any processing while it is in the background. If it cannot do any background processing, there's no use in having the Process Manager give your application access to the CPU while it's in the background.

You specify these and other processing options to the Operating System by including in your application's resource fork a resource of type 'SIZE', known as its *size resource*. The size resource contains several long integers and many flag bits, which together give the Process Manager the information it needs to launch your application and control its processing.

IMPORTANT

Every application executing in system software version 7.0 and later, as well as every application executing in system software version 6.0 with MultiFinder, should contain a size resource. ▲

A 'SIZE' resource consists of a 16-bit flags field, followed by two 32-bit size fields. The flags field specifies operating characteristics of your application, and the size fields indicate the minimum and preferred partition sizes for your application. The *minimum partition size* is the actual limit below which your application will not run. The *preferred partition size* is the memory size at which your application can run most effectively. The Operating System attempts to secure this preferred amount of memory when your application is launched. If that amount of memory is unavailable, your application is placed into the largest contiguous block available, provided that it is larger than the specified minimum size.

Note

If the amount of available memory is between the minimum and the preferred sizes, the Finder displays a dialog box asking if the user wants to run the application using the amount of memory available. If your application does not have a 'SIZE' resource, it is assigned a default partition size of 512 KB, and the Process Manager uses a default value of FALSE for all specifications normally defined by constants in the flags field. ◆

When you define a 'SIZE' resource, you should give it a resource ID of -1. A user can modify the preferred size in the Finder's information window for your application. If the user does alter the partition size, the Operating System creates a new 'SIZE' resource having a resource ID of 0 in your application's resource fork. At application launch time, the Process Manager looks for a 'SIZE' resource with ID 0; if this resource is not found, the Process Manager uses your original 'SIZE' resource (with ID -1). This new 'SIZE' resource is also created when the user modifies any of the other settings in the resource.

Listing 9-1 shows the Rez input for a sample 'SIZE' resource.

Listing 9-1 The Rez input for a sample 'SIZE' resource

```
resource 'SIZE' (-1) {
    reserved,                /*reserved*/
    acceptSuspendResumeEvents, /*accepts suspend and resume events*/
    reserved,                /*reserved*/
    cannotBackground,        /*can't use background null events*/
    doesActivateOnFGSwitch,   /*activates own windows in */
                                /* response to OS events*/
    backgroundAndForeground,  /*application has a user interface*/
    dontGetFrontClicks,      /*don't return mouse events */
                                /* in front window on resume*/
    ignoreAppDiedEvents,      /*doesn't want app-died events*/
    is32BitCompatible,        /*works with 24- or 32-bit addr*/
    notHighLevelEventAware,   /*can't use high-level events*/
    onlyLocalHLEvents,        /*can't use remote high-level events*/
    notStationeryAware,       /*can't use stationery documents*/
    dontUseTextEditServices,  /*can't use inline input services*/
    reserved,                /*reserved*/
    reserved,                /*reserved*/
    reserved,                /*reserved*/
    kPrefSize * 1024,         /*preferred memory size*/
    kMinSize * 1024          /*minimum memory size*/
};
```

The 'SIZE' resource specification in Listing 9-1 indicates, among other things, that the application accepts suspend and resume events, does no processing in the background, activates or deactivates any windows as necessary in response to operating-system events, has a user interface, and doesn't want to receive any mouse event associated with a resume event that was caused by the user clicking in the application's front window. In this example, the Rez input file must define values for the constants `kPrefSize` and `kMinSize`; for example, if `kPrefSize` is set to 50, the preferred partition size is 50 KB.

Note

See the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for a more complete description of the 'SIZE' resource. ♦

The numbers you specify as your application's preferred and minimum partition sizes depend on the particular memory requirements of your application. Your application's memory requirements depend in turn on the size of your application's A5 world, heap, and stack. (See the chapter "Memory" earlier in this book for details about these areas of your application's partition.)

Processes

You can usually make a fairly reliable estimate of the size of your application's A5 world by determining the size of your application's global variables and its jump table (whose size you can determine by looking at the size of your compiled application's 'CODE' resource with ID 0). You can also make a good guess about the size of your application's static heap objects—objects that are always present during the execution of your application (for example, code segments, Toolbox data structures for window records, and so on).

It's a little bit more work to determine the amount of space you'll need to reserve for dynamic heap objects. These include objects created on a per-document basis (which may vary in size proportionally with the document itself) and objects required for specific commands or functions. Perhaps the best advice to follow in determining your application's minimum and preferred partition sizes is to experiment with reasonable values and make sure that there is always enough memory to meet reasonable requests from the user. You can also use tools such as MacsBug's heap-exploring commands to help empirically determine your application's dynamic memory requirements.

Handling Suspend and Resume Events

Your application receives suspend and resume events as a result of changes in its processing status. When your application is in the foreground and the Process Manager wants to switch it into the background, the Process Manager sends it a *suspend event*. This is a signal to your application to prepare to be switched out. Your application isn't actually switched out immediately. Instead, the Process Manager gives your application a chance to handle the suspend event. Your application is switched out at the *next* event call it makes. Similarly, the application that is about to be switched into the foreground is sent a *resume event* once it's actually switched. The resume event is a signal to that application that it can resume normal foreground processing.

Upon receiving a suspend event, your application should deactivate the front window, remove the highlighting from any selections, and hide any floating windows. Your application should also convert any private scrap into the global scrap, if necessary. If your application shows a window that displays the Clipboard contents, you should hide this window also, because the user might change the contents of the Clipboard before returning to your application. Your application can also do anything else necessary to get ready for a major switch. Then your application should call `WaitNextEvent` to relinquish the processor and allow the Operating System to schedule other processes for execution.

Processes

Upon receiving a resume event, your application should activate the front window and restore any windows to the state the user left them in at the time of the previous suspend event. For example, your application should show scroll bars, restore any selections that were previously in effect, and show any floating windows. Your application should copy the contents of the Clipboard and convert the data back to its private scrap, if necessary. If your application shows a window that displays the Clipboard contents, you can update the contents of the window after reading in the scrap. Your application can then resume interacting with the user.

Responding to a suspend or resume event usually involves activating or deactivating windows. If you set the `acceptSuspendResumeEvents` flag and the `doesActivateOnFGSwitch` flag in your application's 'SIZE' resource, your application is responsible for activating or deactivating its windows when it handles suspend and resume events.

Listing 9-2 defines the routine called by the Venn Diagrammer application to handle operating-system events.

Listing 9-2 Handling operating-system events

```
PROCEDURE DoOSEvent (myEvent: EventRecord);
    VAR
        myWindow: WindowPtr;
    BEGIN
        CASE BSR(myEvent.message, 24) OF
            mouseMovedMessage:
                BEGIN
                    DoIdle(myEvent);           {right now, do nothing}
                END;
            suspendResumeMessage:
                BEGIN
                    myWindow := FrontWindow;
                    IF (BAnd(myEvent.message, resumeFlag) <> 0) THEN
                        DoActivate(myWindow, activeFlag)      {activate window}
                    ELSE
                        DoActivate(myWindow, 1 - activeFlag); {deactivate window}
                    END;
                OTHERWISE
                    ;
                END;
        END;
    END;
```

Processes

The procedure `DoOSEvent` is called by the main event loop (Listing 4-4 on page 77) whenever the `what` field of an event record contains the constant `osEvt`. You need to inspect the `message` field of that event record to determine what kind of operating-system event you've received. Table 9-1 shows the information contained in the bits of the `message` field.

Table 9-1 The bits in the `message` field of an operating-system event record

Bit	Contents
0	0 if a suspend event 1 if a resume event
1	0 if Clipboard conversion is not required 1 if Clipboard conversion is required
2–23	Reserved
24–31	<code>suspendResumeMessage</code> if a suspend or resume event <code>mouseMovedMessage</code> if a mouse-moved event

As you can see, you need to inspect bits 24–31 to determine what kind of operating-system event you've received. Those eight bits contain one of two constants:

```
CONST
    suspendResumeMessage    = $01;      {suspend or resume event}
    mouseMovedMessage       = $FA;      {mouse-moved event}
```

If the event is a suspend or resume event, you then need to examine bit 0 to determine whether that event is a suspend or resume event. (Bits 0 and 1 are meaningful only if bits 24–31 indicate that the event is a suspend or resume event.) You can use the `resumeFlag` constant to determine whether the event is a suspend or resume event. If the event is a resume event, you can use the `convertClipboardFlag` constant to determine whether Clipboard conversion from the Clipboard to your application's scrap is required.

```
CONST
    resumeFlag              = 1;  {resume event}
    convertClipboardFlag    = 2;  {Clipboard conversion required}
```

The procedure `DoOSEvent` defined in Listing 9-2 first checks what kind of event it has received. If the event is a mouse-moved event, `DoOSEvent` ignores the event, treating it like a null event. If the event is a suspend or resume event, `DoOSEvent` then activates or deactivates the front window, depending on whether the event is a resume or a suspend event.

Note

Because the Venn Diagrammer application doesn't support cutting or pasting, it doesn't need to worry about converting the Clipboard. ♦

Handling Null Events

Recall that the Event Manager sends your application a null event when there are no other events to report. The `WaitNextEvent` function reports a null event by returning a function result of `FALSE` and by setting the `what` field of the event record to `nullEvt`.

When your application receives a null event, it can perform idle processing. Your application should do only minimal processing in response to a null event, so that other processes can use the CPU and so that the foreground process (or your application, when it is in the foreground) can respond promptly to the user. For example, if your application is in the foreground when it receives a null event, you can make the insertion point blink in the active window (if your application supports text entry).

If your application receives a null event in the background, it can perform tasks or do other processing while in the background. However, your application should not perform any tasks that would slow down the responsiveness of the foreground process. Your application also should not interact with the user if it is in the background.

Note

Remember that your application receives null events while it is in the background only if you've set the `canBackground` flag in your application's 'SIZE' resource. If you don't want your application to receive null events when it is in the background, you should set the `cannotBackground` flag. ♦

The Venn Diagrammer application uses null events in a somewhat interesting way. Whenever the application receives a null event, it calls the application-defined procedure `DoIdle`, which checks to see whether the user wants it to automatically adjust the Venn diagram and whether the diagram might need adjusting. If both of these are true, then `DoIdle` calls the application-defined procedure `DoVennIdle` to perform the automatic adjustment. The `DoIdle` procedure is defined in Listing 9-3.

Listing 9-3 Handling null events

```
PROCEDURE DoIdle (myEvent: EventRecord);
    VAR
        myWindow:    WindowPtr;
        myHandle:    MyDocRecHnd;
BEGIN
    myWindow := FrontWindow;
    IF IsAppWindow(myWindow) THEN
        IF gAutoAdjust THEN
```

Processes

```

BEGIN
    myHandle := MyDocRecHnd(GetWRefCon(myWindow));
    IF myHandle^.needsAdjusting THEN
        DoVennIdle(myWindow);
    END;
END;

```

The document record contains the field `needsAdjusting`, which is set to `TRUE` each time the user clicks anywhere within the Venn diagram circles. If the user's preference is for automatic diagram adjustment, then `DoIdle` calls the application-defined procedure `DoVennIdle` to adjust the diagram. Figure 9-2 shows the state of a diagram needing adjustment, and Figure 9-3 shows the same diagram after `DoVennIdle` has adjusted the diagram.

Note

The `DoVennIdle` procedure is not defined in this book. In addition to determining whether and how to adjust the diagram, `DoVennIdle` resets the `needsAdjusting` field of the document record to `FALSE`. ♦

Figure 9-2 A Venn diagram before automatic adjusting

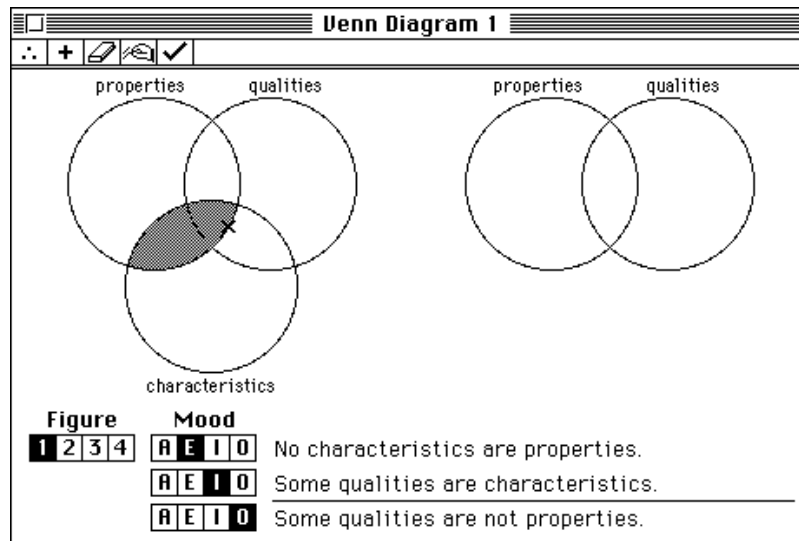
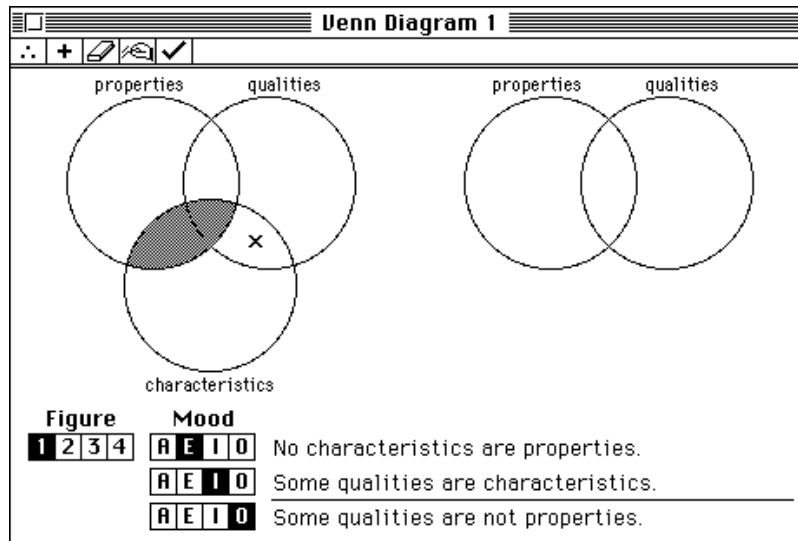


Figure 9-3 A Venn diagram after automatic adjusting

Quitting an Application

Eventually the user will quit your application, usually by choosing Quit from the File menu (or by pressing the usual keyboard equivalent, Command-Q). At that time, you should close all windows, release any memory you still are holding, and exit your main event loop. Listing 9-4 shows the `DoQuit` routine called by the Venn Diagrammer application when the user chooses Quit from the File menu.

Listing 9-4 Quitting your application

```
PROCEDURE DoQuit;
  VAR
    myWindow: WindowPtr;
BEGIN
  myWindow := FrontWindow;           {close all windows}
  WHILE myWindow <> NIL DO
  BEGIN
    DoUpdate(myWindow);               {force redrawing window}
    DoCloseWindow(myWindow);
    myWindow := FrontWindow;
  END;
END;
```

Processes

```

END;
gDone := TRUE;                                {set flag to exit main event loop}
END;

```

The `DoQuit` procedure simply closes all windows belonging to the application and then sets the application global variable `gDone` to indicate that the user has finished using the application. Recall that the main event loop (Listing 4-4 on page 77) terminates when `gDone` is `TRUE`.

Note

The Process Manager automatically deallocates your application partition and closes all windows when your application terminates. As a result, the Venn Diagrammer application could simply have set `gDone` to `TRUE` in response to the Quit command. However, `DoQuit` illustrates how to close all windows because your version of `DoCloseWindow` might need to prompt the user to save any unsaved data in document windows currently on the desktop. ♦

Handling Errors

Occasionally, a system software routine might be unable to perform the service you've requested of it. You might, for instance, pass `GetResource` a resource specification that doesn't apply to any resource in any of the open resource files. Or, the user might have opened so many document windows that there simply isn't enough space in your application's heap to open another one. In these situations, you need to determine that an error has occurred and react to it in some appropriate manner.

The system software has several ways of informing your application that a requested service is not possible. Many functions return a result code that indicates whether the function completed successfully, and if not, what the reason for failure was. These functions return a result of type `OSErr`. Here's an example:

```

myResult := FindFolder(kOnSystemDisk, kPreferencesFolderType,
                      kDontCreateFolder, myVRefNum, myDirID);
IF myResult = noErr THEN
    ...
ELSE
    ...;

```

Other routines—mainly procedures and functions that return other types of results—don't return a result code directly. To find out whether these kinds of routines were successful, you need to call an additional system software routine. For example, some Resource Manager procedures don't directly indicate if the resource operation was successful or not. To find that out, you can call the `ResError` function. The `DoSavePrefs` routine (defined in Listing 3-6 on page 66) uses this strategy to update a preferences resource:

Processes

```

RmveResource(myHandle);
IF ResError = noErr THEN
    AddResource(myPrefData, kPrefResType, kPrefResID, myName);
IF ResError = noErr THEN
    WriteResource(myPrefData);

```

Similarly, the Resource Manager routine `Get1Resource` returns a handle to the specified resource data. If for some reason the resource cannot be opened, the function returns a handle whose value is `NIL`. You can inspect the returned value to determine whether it's safe to proceed.

```

myHandle := Get1Resource(kPrefResType, kPrefResID);
IF myHandle <> NIL THEN
    ...;

```

You could also call `ResError` to determine if `Get1Resource` succeeded. In other words, the following lines are equivalent to the preceding ones:

```

myHandle := Get1Resource(kPrefResType, kPrefResID);
IF ResError <> noErr THEN
    ...;

```

The Memory Manager provides the `MemError` function, which works much as `ResError` does. For Memory Manager functions that return a value, you can either inspect the returned value or call `MemError` to determine if the function completed successfully.

This book has used a fairly simple strategy for detecting and reacting to the normal kinds of problems. When calling a function that returns a pointer or handle, Venn Diagrammer checks that the value of that pointer or handle isn't `NIL`. If it is `NIL`, Venn Diagrammer usually just skips any code that uses that pointer or handle.

IMPORTANT

Venn Diagrammer's error-handling strategy is far too simple for most applications, and it runs afoul of good human interface principles. For example, if the `DoCreateWindow` function (defined in Listing 6-6 on page 117) cannot allocate the memory it needs, it exits and returns a `NIL` window pointer to the calling routine. The net result is that no new window is created, in spite of the user's desire to create one. At the very least, `DoCreateWindow` should inform the user that a new window could not be created because sufficient memory was not available. ▲

Occasionally, an application might run into some more serious problem during its execution that renders further processing impossible or undesirable. For example, if the Venn Diagrammer application isn't able to allocate enough memory for the data structure it uses to maintain information about a document window's geometry, there's no point in continuing to run, because the application won't be able to draw anything in any document windows. In that case, the application should gracefully terminate its own execution. (See Listing 5-3 on page 95.)

Processes

To do this, the Venn Diagrammer application defines the `DoBadError` procedure and calls it whenever there is a problem serious enough to warrant such drastic action. The `DoBadError` procedure is defined in Listing 9-5.

Listing 9-5 Handling serious errors

```
PROCEDURE DoBadError (myError: Integer);
VAR
    myItem:      Integer;
    myMessage:   Str255;
BEGIN
    SetCursor(arrow);                {set arrow cursor}
    GetIndString(myMessage, kErrorStrings, myError);
    ParamText(myMessage, '', '', '');
    myItem := Alert(rErrorAlert, NIL); {display message}
    ExitToShell;                     {terminate execution}
END;
```

The application passes `DoBadError` an index into a resource of type 'STR#' that contains messages indicating the types of serious errors. First `DoBadError` sets the cursor to the standard arrow cursor (this step is necessary only if your application ever changes the cursor). Then `DoBadError` retrieves the appropriate message from the application's resource fork and calls the Dialog Manager routine `ParamText` to substitute the message into the alert box text. After that, `DoBadError` displays the alert box by calling the Dialog Manager routine `Alert`. (See Figure 7-2 on page 134 for an example of this alert box.) Finally, `DoBadError` calls the Process Manager procedure `ExitToShell` to terminate the application immediately.

Checking the Operating Environment

Calling `ExitToShell` is the preferred way to terminate your application if for some reason you don't want to return to your main event loop. You might also want to call `DoBadError` to terminate your application before you even get to the main event loop. This might happen if your application requires system software routines that aren't available in all operating environments. In general, if your application uses any system software routines that aren't available in all operating environments, you need to make sure that they are available in the current environment. Otherwise, your application will crash.

For example, the Venn Diagrammer application uses the `FindFolder` function to find the Preferences folder containing the application's preferences file (see Listing 3-3 on page 62). Because `FindFolder` was introduced in system software version 7.0, Venn Diagrammer will crash if it calls `FindFolder` when running in an earlier system software version.

Processes

To avoid crashing in environments that don't support the `FindFolder` function, the Venn Diagrammer application makes sure that the function is available before calling it. It calls the `Gestalt` function to see if `FindFolder` is present, as shown in Listing 9-6.

Listing 9-6 Checking that `FindFolder` is present

```
FUNCTION IsFindFolder: Boolean;
VAR
    myResult:   OSErr;
    myFeature:  LongInt;
BEGIN
    IsFindFolder := FALSE;           {assume it's not available}
    myResult := Gestalt(gestaltFindFolderAttr, myFeature);
    IF myResult = noErr THEN
        IsFindFolder := BTST(myFeature, gestaltFindFolderPresent);
END;
```

The `Gestalt` function is part of the Gestalt Manager, which you can use to determine what software and hardware features are available in the current operating environment. When passed the `gestaltFindFolderAttr` selector code, the `Gestalt` function fills in the long integer passed in its second parameter (`myFeature`) with a bit field that encodes information about the features of the `FindFolder` function. Currently only one bit is defined, specified using the constant `gestaltFindFolderPresent`. If that bit is set, then `FindFolder` is present in the operating environment. The Venn Diagrammer application calls `IsFindFolder` as follows (see Listing 3-3 on page 62):

```
IF IsFindFolder THEN
    myResult := FindFolder(kOnSystemDisk, kPreferencesFolderType,
                          kDontCreateFolder, myVRefNum, myDirID);
```

Note

For complete details about using the `Gestalt` function to determine the features of the current operating environment, see the chapter “Gestalt Manager” in *Inside Macintosh: Operating System Utilities*. ♦

If `FindFolder` function isn't available, Venn Diagrammer looks in the *default directory* instead of in the Preferences folder for the user's preferences file. This isn't the best strategy possible, but it's good enough for a simple application like Venn Diagrammer. More generally, however, you need to decide what the base system software requirements of your application are and how you want to react if necessary services aren't available. In some cases, working around a problem isn't so easy. In those cases, informing the user that your software won't run in the current system configuration and then exiting is probably the right thing to do.

A second way to determine the availability of a particular system software routine is to test directly for the existence of the routine by inspecting its trap number (a number that identifies each system software routine), using the technique illustrated in Listing 9-7.

Processes

You should use this method to test for the existence of routines not included in managers about which Gestalt can report.

Listing 9-7 Determining whether a trap is available

```

FUNCTION NumToolboxTraps: Integer;
BEGIN
    IF NGetTrapAddress(_InitGraf, ToolTrap) =
        NGetTrapAddress($AA6E, ToolTrap) THEN
        NumToolboxTraps := $200
    ELSE
        NumToolboxTraps := $400;
END;

FUNCTION GetTrapType (theTrap: Integer): TrapType;
CONST
    TrapMask = $0800;
BEGIN
    IF BAND(theTrap, TrapMask) > 0 THEN
        GetTrapType := ToolTrap
    ELSE
        GetTrapType := OSTrap;
END;

FUNCTION TrapAvailable (theTrap: Integer): Boolean;
VAR
    tType:      TrapType;
BEGIN
    tType := GetTrapType(theTrap);
    IF tType = ToolTrap THEN
        BEGIN
            theTrap := BAND(theTrap, $07FF);
            IF theTrap >= NumToolboxTraps THEN
                theTrap := _Unimplemented;
            END;
            TrapAvailable := NGetTrapAddress(theTrap, tType) <>
                NGetTrapAddress(_Unimplemented, ToolTrap);
        END;
END;

```

Processes

Listing 9-8 shows how to use the `TrapAvailable` function defined in Listing 9-7 to determine whether the `WaitNextEvent` function is available.

Listing 9-8 Checking for the availability of the `WaitNextEvent` function

```
FUNCTION WNEAvailable: Boolean;
CONST
    _WaitNextEvent    = $A860;    {trap number of WaitNextEvent}
BEGIN
    WNEAvailable := TrapAvailable(_WaitNextEvent);
END;
```

The `NumToolboxTraps` function relies on the fact that the `InitGraf` trap (trap number \$A86E) is always implemented. If the trap dispatch table is large enough (that is, has more than \$200 entries), then \$AA6E always points to either `_Unimplemented` or something else, but never to `InitGraf`. As a result, you can check the size of the trap dispatch table by checking to see if the address of trap \$A86E is the same as \$AA6E.

After receiving the information about the size of the dispatch table, the `TrapAvailable` function first checks to see if the trap to be tested has a trap number greater than the total number of traps available on the machine. If so, it sets the `theTrap` variable to `_Unimplemented` before testing it against the `_Unimplemented` trap. See the discussion of the trap dispatch table utilities in *Inside Macintosh: Operating System Utilities* for complete details on trap numbers and the trap dispatch table.

IMPORTANT

There's one final twist in this story. Your software development system might provide glue routines that mimic the operation of some system software routines, thereby allowing you to call them in earlier system software versions. (For instance, MPW versions 3.2 and later provide glue that allows you to call `FindFolder` in system software versions prior to 7.0.) However, you cannot in general use `Gestalt` or the technique shown in Listing 9-7 to test for the availability of routines provided as glue. Instead, you'll need to consult the documentation for your development system to find out what glue routines it provides. ▲

