INSIDE MACINTOSH

Processes

Addison-Wesley Publishing Company

Reading, Massachusetts Menlo Park, California New York
Don Mills, Ontario Wokingham, England Amsterdam Bonn
Sydney Singapore Tokyo Madrid San Juan
Paris Seoul Milan Mexico City Taipei

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

ISBN 0-201-63241-1
1 2 3 4 5 6 7 8 9-MU-9695949392
First Printing, August 1992

Contents

Figures and Listings

About This Book

This book describes the parts of the Macintosh Operating System that allow you to manage processes and tasks. It includes introductory material about managing processes on Macintosh computers as well as a complete technical reference to the Process Manager, the Time Manager, the Vertical Retrace Manager, and other process-related services provided by the system software.

If you are new to programming on the Macintosh Operating System, you should begin with the chapter "Introduction to Processes and Tasks." This chapter provides a general introduction to process and task management on Macintosh computers. It describes 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 run. 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 also describes how your application can install tasks that are executed asynchronously from your application, usually in response to interrupts. You can

- execute a task after a certain amount of time has elapsed
- execute a task repetitively
- notify the user while your application is in the background
- execute a task between screen refreshes
- execute a routine as part of the shutdown or restart process

Once you are familiar with basic process and task management on Macintosh computers, you might need to look at the chapter "Process Manager." It describes how you can get information about open processes and, if necessary, launch processes and desk accessories. You can also use the Process Manager to alter the processing status of an application or to terminate your application.

If you want a task to be executed after some specified amount of time has elapsed, you can use the Time Manager to schedule that task for later execution. The task can reschedule itself, so you can use the Time Manager to execute a routine repetitively. You can also use the Time Manager to calculate elapsed times and to synchronize events in the Macintosh computer. See the chapter "Time Manager" for details.

The Vertical Retrace Manager allows you to schedule a task for execution during vertical retrace interrupts. Like Time Manager tasks, vertical retrace tasks can reschedule themselves so that they are executed repetitively. In general, you should use the Vertical Retrace Manager to handle repetitive

tasks that need to be synchronized with the redrawing of the screen and the Time Manager to handle those tasks that don't. See the chapter "Vertical Retrace Manager" for details.

You can use the Notification Manager to inform users of significant occurrences in applications that are running in the background or in software that is largely invisible to the user. This software includes device drivers, vertical blanking (VBL) tasks, Time Manager tasks, completion routines, and desk accessories that operate behind the scenes. See the chapter "Notification Manager" for complete details.

You should read the chapter "Segment Manager" for information about how the Operating System manages the loading and unloading of your application's code segments into and out of memory. By dividing your application's executable code into segments, you allow it to run in a memory partition that is smaller than the total size of the application itself.

The final chapter in this book, "Shutdown Manager," shows how you can install procedures that are executed as part of the final stages of shutting down or restarting a Macintosh computer.

Format of a Typical Chapter

Almost all chapters in this book follow a standard structure. For example, the chapter "Process Manager" contains these sections:

- "About the Process Manager." This section provides an overview of the features provided by the Process Manager.
- "Using the Process Manager." This section describes the tasks you can accomplish using the Process Manager. It describes how to use the most common routines, gives related user interface information, provides code samples, and supplies additional information.
- "Process Manager Reference." This section provides a complete reference to the Process Manager by describing the constants, data structures, and routines that it uses. Each routine description also follows a standard format, which gives the routine declaration and description of every parameter of the routine. Some routine descriptions also give additional descriptive information, such as assembly-language information or result codes.
- "Summary of the Process Manager." This section provides the Process Manager's Pascal interface, as well as the C interface, for the constants, data structures, routines, and result codes associated with the Process Manager. It also includes relevant assembly-language interface information.

Some chapters contain additional main sections that provide more detailed discussions of certain topics. For example, in the chapter "Shutdown Manager," the section "The Shutdown Process" describes the process that the Shutdown Manager procedures perform to shut down or restart the system.

Conventions Used in This Book

Inside Macintosh uses various conventions to present information. Words that require special treatment appear in specific fonts or font styles. Certain information, such as parameter blocks, use special formats so that you can scan them quickly.

Special Fonts

All code listings, reserved words, and the names of actual data structures, constants, fields, parameters, and routines are shown in Courier (`this is Courier`).

Words that appear in **boldface** are key terms or concepts and are defined in the Glossary.

Types of Notes

There are several types of notes used in this book.

Note

A note like this contains information that is interesting but possibly not essential to an understanding of the main text. (An example appears on page 1-4.) u

IMPORTANT

A note like this contains information that is essential for an understanding of the main text. (An example appears on page 2-16.) s

S **WARNING**

Warnings like this indicate potential problems that you should be aware of as you design your application. Failure to heed these warnings could result in system crashes or loss of data. (An example appears on page 1-12.) s

Assembly-Language Information

Inside Macintosh provides information about the registers for specific routines like this:

Registers on entry

A0 Contents of register A0 on entry

Registers on exit

D0 Contents of register D0 on exit

In addition, *Inside Macintosh* presents information about the fields of a parameter block in this format:

Parameter block

↔	`inAndOut`	`Integer`	Input/output parameter.
←	`output1`	`Ptr`	Output parameter.
→	`input1`	`Ptr`	Input parameter.

The arrow in the far left column indicates whether the field is an input parameter, output parameter, or both. You must supply values for all input parameters and input/output parameters. The routine returns values in output parameters and input/output parameters.

The second column shows the field name as defined in the MPW Pascal interface files; the third column indicates the Pascal data type of that field. The fourth column provides a brief description of the use of the field. For a complete description of each field, see the discussion that follows the parameter block or the description of the parameter block in the reference section of the chapter.

Development Environment

The system software routines described in this book are available using Pascal, C, or assembly-language interfaces. How you access these routines depends on the development environment you are using. This book shows system software routines in their Pascal interface using the Macintosh Programmer's Workshop (MPW).

All code listings in this book are shown in Pascal. They show methods of using various routines and illustrate techniques for accomplishing particular tasks. All code listings have been compiled and, in most cases, tested. However, Apple Computer does not intend that you use these code samples in your application.

This book occasionally uses *SurfWriter* as the name of a sample application for illustrative purposes; this is not an actual product of Apple Computer, Inc.

APDA, Apple's source for developer tools, offers worldwide access to a broad range of programming products, resources, and information for anyone developing on Apple platforms. You'll find the most current versions of Apple and third-party development tools, debuggers, compilers, languages, and technical references for all Apple platforms. To establish an APDA account, obtain additional ordering information, or find out about site licensing and developer training programs, contact

APDA
Apple Computer, Inc.
20525 Mariani Avenue, M/S 33-G
Cupertino, CA 95014-6299

Telephone: 800-282-2732 (United States)
800-637-0029 (Canada)
800-562-3910 (elsewhere in the world)

Fax: 408-562-3971

Telex: 171-576

If you provide commercial products and services, call 408-974-4897 for information on the developer support programs available from Apple.

For information on registering signatures, file types, Apple events, and other technical information, contact

Macintosh Developer Technical Support
Apple Computer, Inc.
20525 Mariani Avenue, M/S 75-3T
Cupertino, CA 95014-6299

CHAPTER 1

Introduction to Processes and Tasks

Contents

This chapter is a general introduction to process and task management on Macintosh computers. It describes 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 also describes how you can use the services provided by the Time Manager, the Vertical Retrace Manager, and other parts of the Macintosh Operating System to schedule tasks for execution outside the time provided to your application by the Process Manager. Usually these tasks are executed in response to an interrupt.

You should read this chapter for an overview of how the Process Manager schedules applications and loads them into memory. You also need to read this chapter if you install any tasks that execute at interrupt time, which are subject to a number of important restrictions.

To use this chapter, you need to be familiar with how your application uses memory, as described in the chapter "Introduction to Memory Management" in *Inside Macintosh: Memory*. You should also be familiar with how your application receives events, as discussed in the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

This chapter begins with a general discussion of processes and tasks. Then it describes in detail the operation of the Process Manager in launching and scheduling processes. This chapter ends with a description of installing tasks that execute at interrupt time. For a more complete discussion of these topics, see the remaining chapters in this book.

The Cooperative Multitasking Environment

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.

For example, your application should include a `'SIZE'` resource that specifies how large a memory partition it should be allocated at application launch time. If that much memory is available when your application is launched, the Process Manager allocates it and sets up your application partition. Similarly, your application should periodically make an event call to allow the Operating System the opportunity to schedule other applications for execution. Because the smooth operation of all applications depends on their cooperation, this environment is known as a **cooperative multitasking environment.**

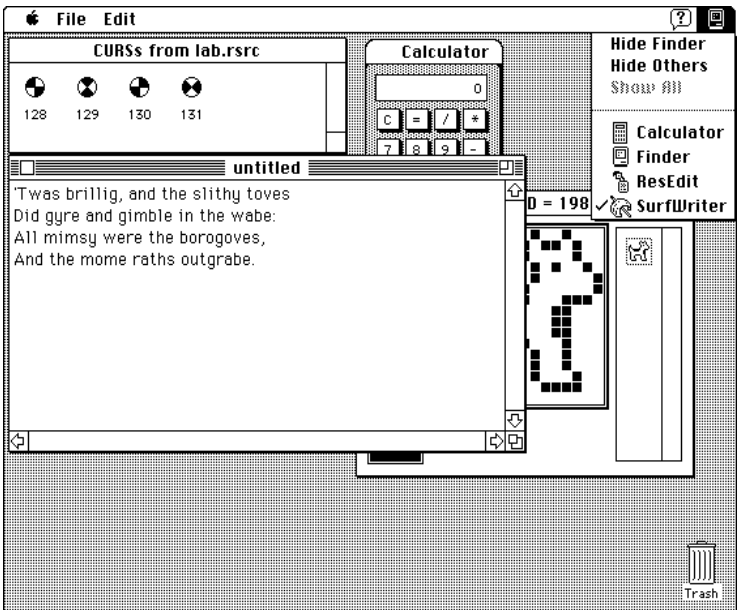
Note
The cooperative multitasking environment is available in system software versions 7.0 and later, and when the MultiFinder option is enabled in earlier system software versions. u

The Operating System schedules the processing of all applications and desk accessories. When a user opens a document or 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 or application relinquishes the CPU. In most cases, the application runs immediately (or so it appears to the user).

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.

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 in 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 1-1, SurfWriter is the active application. Windows of other applications are visible on the desktop behind the frontmost window.

Figure 1-1 The desktop with several applications open



Most processing in the cooperative multitasking environment is done by applications or desk accessories. Occasionally, you might need to install a task to be executed in response to an interrupt. In general, however, it is best to avoid installing interrupt tasks if at all possible. Interrupt tasks must be small and fast, and they are subject to a number of limitations that do not apply to applications. The Operating System itself is heavily interrupt-driven, and you can severely impair the responsiveness of the computer by installing too many tasks or tasks that take too long to complete.

About Processes

The Process Manager manages the scheduling of processes. A **process** is an open application or, in some cases, an open desk accessory. (Desk accessories that are opened in the context of an application are not considered processes.) The number of processes is limited only by available memory.

The Process Manager maintains information about each process—for example, the current state of the process, the address and size of its partition, its type, its creator, a copy of all process-specific system global variables, information about its `'SIZE'` resource, and a process serial number. This process information is referred to as the **context** of a process. The Process Manager assigns a **process serial number** to identify each process. A process serial number identifies a particular instance of an application; this number is unique during a single boot of the local machine.

The **foreground process** is the one currently interacting with the user; it appears to the user as the active application. The foreground process displays its menu bar, and its windows are in front of the windows of all other applications.

A **background process** is a process that isn't currently interacting with the user. At any given time a process is either in the foreground or the background; a process can switch between the two states at well-defined times.

The foreground process has first priority for accessing the CPU. Other processes can access the CPU only when the foreground process yields time to them. There is only one foreground process at any one time. However, multiple processes can exist in the background.

An application that is in the background can get CPU time but can't interact with the user while it is in the background. (However, the user can bring the application to the foreground—for example, by clicking in one of the application's windows.) Any application that has the `canBackground` flag set in its `'SIZE'` resource is eligible to obtain access to the CPU when it is in the background.

Applications can be designed without a user interface; these are called **background-only applications.** A background-only application does not call the Window Manager `InitWindows` routine and is identified by having the `onlyBackground` flag set in its `'SIZE'` resource. Background-only applications do not display windows or a menu bar and are not listed in the Application menu.

Background-only applications and applications that can run in the background should be designed to relinquish the CPU often enough so that the foreground process can perform its work and respond to the user.

Once an application is running, in either the foreground or the background, the CPU is available only to that application. That application can be interrupted only by hardware interrupts, which are transparent to the scheduling of the application. However, the application that is running must periodically relinquish control of the CPU. This yielding of the CPU allows background applications access to processing time and lets users interact with the foreground application or switch to another application.

Your application can relinquish control of the CPU each time you call the Event Manager functions `WaitNextEvent` or `EventAvail`. If, at that time, there are no events pending for your application, the Process Manager may schedule other processes for execution. (You can also call the `GetNextEvent` function; however, you should use `WaitNextEvent` to provide greater support for cooperative multitasking.)

Process Creation

When a user first opens your application, the Process Manager creates a partition for it. A **partition** is a contiguous block of memory that the Process Manager allocates for your application's use. The partition is divided into specific areas: application heap, A5 world, and stack. The **application heap** contains the application's `'CODE'` segment 1, data structures, resources, and other code segments as needed. The **A5 world** contains the application's QuickDraw global variables, its application global variables, and its jump table, all of which are accessed through the A5 register. The application **jump table** contains one entry for every externally referenced routine in every code segment of your application. The application **stack** is used to store temporary variables. (See the chapter "Introduction to Memory Management" in *Inside Macintosh: Memory* for more complete details on these areas of your application's partition.)

When you create an application, you specify in its `'SIZE'` resource how much memory you want the Process Manager to allocate for your application's partition. You specify two values: the preferred amount of memory to allocate and the minimum amount of memory to allocate. When a user opens your application from the Finder, the Process Manager first attempts to allocate a partition of the preferred size. If your application cannot be launched in the preferred amount of memory, the Finder might display a dialog box giving the user the option of opening the application using less than the preferred size. The Finder will not launch your application if the minimum amount of memory specified for your application is not available.

After the Process Manager creates a partition for your application, the Process Manager loads your code into memory and sets up the stack, heap, and A5 world (including the jump table) for your application. If the user selects one or more files to open or print, the Finder sets up information your application can use to determine which files to open or print.

The Process Manager assigns the application a process serial number, records its context, and returns control to the launching application (usually the Finder). The Process

Manager typically transfers control to the new application after the launching application makes a subsequent call to `WaitNextEvent` or `EventAvail`.

The next section describes how your application can allow other applications to receive CPU time and how the Process Manager schedules CPU time among processes.

Process Scheduling

Your application can yield control of the CPU to other processes only at very specific times, namely when you call the Event Manager functions `WaitNextEvent` or `EventAvail`. Whenever your application calls one of these functions, the Process Manager checks the status of your process and takes the opportunity to schedule other processes.

Note

Your application can also yield processing time to other processes as a result of calling other Toolbox routines containing internal calls to `WaitNextEvent` or `EventAvail`. For example, your application can yield the CPU to other processes as a result of calling either of the Apple Event Manager functions `AESend` or `AEInteractWithUser`. See the chapter "Apple Event Manager" in *Inside Macintosh: Interapplication Communication* for information on using these two functions. u

In general, your application continues to receive processing time as long as any events are pending for it. When your application is the foreground process, it yields time to other processes in these situations: when the user wants to switch to another application or when no events are pending for your application. Your application can also choose to yield processing time to other processes when it is performing a lengthy operation.

A **major switch** occurs when the Process Manager switches the context of the foreground process with the context of a background process (including the A5 worlds and application-specific system global variables) and brings the background process to the front, sending the previous foreground process to the background.

When your application is the foreground process and the user elects to work with another application (by clicking in a window of another application, for example), the Process Manager sends your application a **suspend event** if the `acceptSuspendResumeEvents` bit is set in your application's `'SIZE'` resource. When your application receives a suspend event, it should prepare to suspend foreground processing, allowing the user to switch to the other application. For example, in response to the suspend event, your application should remove the highlighting from the controls of its frontmost window and take any other necessary actions. Your application is actually suspended the next time it calls `WaitNextEvent` or `EventAvail`.

After your application receives the suspend event and calls `WaitNextEvent` or `EventAvail`, the Process Manager saves the context of your process, restores the context of the process to which the user is switching, and sends a **resume event** to that process (if the `acceptSuspendResumeEvents` bit is set in its `'SIZE'` resource). In response to a resume event, your application should resume processing and start

interacting with the user. For example, your application should highlight the controls of its frontmost window.

A major switch also occurs when the user hides the active application (by choosing the Hide command in the Application menu). In general, a major switch cannot occur when a modal dialog box is the frontmost window. However, a major switch can occur when a movable modal dialog box is the frontmost window.

A **minor switch** occurs when the Process Manager switches the context of a process to give time to a background process without bringing the background process to the front. For example, a minor switch occurs when no events are pending in the event queue of the foreground process. In this situation, processes running in the background have an opportunity to execute when the foreground process calls `WaitNextEvent` or `EventAvail`. (If the foreground process has one or more events pending in the event queue, then the next event is returned and the foreground process again has sole access to the CPU.)

When an application is switched out in this way, the Process Manager saves the context of the current process, restores the context of the next background process scheduled to run, and sends the background process an event. At this time, the background process can receive either update, null, or high-level events.

A background process should not perform any task that significantly limits the ability of the foreground process to respond quickly to the user. A background process should call `WaitNextEvent` often enough to let the foreground process be responsive to the user. Upon receiving an update event, the background process should update only the content of its windows. Upon receiving a null event, the background process can use the CPU to perform tasks that do not require significant amounts of processing time.

The next time the background process calls `WaitNextEvent` or `EventAvail`, the Process Manager saves the context of the background process and restores the context of the foreground process (if the foreground process is not waiting for a specified amount of time to expire before being scheduled again). The foreground process is then scheduled to execute. If no events are pending for the foreground process and it is waiting for a specified amount of time to expire, the Process Manager schedules the next background process to run. The Process Manager continues to manage the scheduling of processes in this manner.

Drivers and vertical blanking (VBL) tasks installed in the system heap are scheduled regardless of which application is currently executing. Drivers installed in an application's heap are not scheduled to run when the application is not executing. See the section "Task Scheduling," beginning on page 1-11, for more information about the scheduling of interrupt tasks.

Note

See the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for specific information on how your application can handle suspend and resume events and how your application can take advantage of the cooperative multitasking environment. u

Whenever your application calls `WaitNextEvent` or `EventAvail`, the Process Manager checks the status of your process and takes the opportunity to schedule other processes. Using the `WaitNextEvent` function, you can control when your process is eligible to be switched out.

The `sleep` parameter of the `WaitNextEvent` function specifies a length of time, in ticks, during which the application relinquishes the CPU if no events are pending. For example, if you specify a nonzero value in the `sleep` parameter and no events are pending in your application's event queue when you call `WaitNextEvent`, the Process Manager saves the context of your process and schedules other processes until an event becomes available or the time expires. Once the specified time expires or an event becomes available for your application, your process becomes eligible to run. At this time, the Process Manager schedules your process to run at the next available chance. (You can also call the Process Manager's `WakeUpProcess` function to make a process eligible to run before the time in the `sleep` parameter expires.) If the time specified by `sleep` expires and no events are pending for your application, the Process Manager sends your application a null event.

In general, you should specify a value greater than 0 in the `sleep` parameter so that those applications that need processing time can get it. If your application performs any periodic task, then the frequency of the task usually determines what value you specify in the `sleep` parameter. The less frequent the task, the higher the value of the `sleep` parameter. A reasonable value for the `sleep` parameter is 60.

About Tasks

An **interrupt** is a form of **exception,** an error or special condition detected by the microprocessor in the course of program execution. In particular, an interrupt is an exception that is signaled to the processor by a device. You cannot predict what your application will be doing when an interrupt task is executed. Interrupts can occur not only between different statements that your application executes but also in the middle of a single call that your application makes. For example, your application might invoke a Toolbox trap, and the microprocessor could receive an interrupt in the middle of the execution of the corresponding Toolbox routine.

Interrupts are usually sent by a device to notify the microprocessor of a change in the condition of the device. Routines that are executed as a result of an interrupt are known as **interrupt tasks.** For example, an interrupt might cause execution of an interrupt task that checks regularly for a change in the position of the mouse and updates the position of the cursor to reflect any change.

Your application can initiate interrupt tasks of its own. For example, you could write an interrupt task that repeatedly spins the cursor or increments a global variable. However, even application-generated interrupt tasks do not occur at predictable points in your application's execution. Applications can schedule tasks to be performed at regular time intervals, such as 100 times per second, or in response to conditions in hardware devices. Tasks scheduled at regular time intervals are actually executed in response to hardware

devices that perceive that requested time intervals have elapsed. The actual execution of tasks is independent of the flow of application code.

Task Creation

Many interrupt tasks are handled by system software and are transparent to your application. However, your application can use any of several facilities to install its own interrupt tasks that are executed not at regular points in the flow of its code but at intervals determined by hardware devices.

- The Time Manager allows you to schedule periodic tasks and tasks to be executed after a certain amount of time has elapsed. You can, for example, use the Time Manager to compute elapsed times with great precision.
- The Vertical Retrace Manager allows you to schedule tasks to be executed between retraces of a video screen. Tasks that you schedule with the Vertical Retrace Manager can reset themselves, just like Time Manager tasks. Although the Vertical Retrace Manager lacks the great precision of the Time Manager, it is available on all Macintosh models.
- The Notification Manager allows both processes in the background and interrupt tasks to alert the user. For example, your application might need to inform the user that some error has occurred, rendering further background processing impossible. You can pass to the Notification Manager's installation routine a pointer to a response procedure to be executed as the final stage of notification.
- The Device Manager allows device drivers for slot cards to install interrupts. If you are writing slot-interrupt tasks, you might also wish to use the Deferred Task Manager, which allows you to defer lengthy interrupt tasks that might prevent other interrupt tasks from executing.

All of these managers need to maintain information about multiple interrupt tasks that might have been installed. To hold such information, the Operating System uses data structures known as **operating-system queues.** For more information on the structure of such queues, see the chapter "Queue Utilities" in *Inside Macintosh: Operating System Utilities.*

When an interrupt causes the microprocessor to suspend normal execution, the processor uses the stack to save the address of the next instruction and the processor's internal status. In this way, when the microprocessor completes execution of interrupt tasks, it can resume the current process where it left off.

After storing these values on the stack, the microprocessor executes an **interrupt handler** to deal with the interrupt. The addresses of all of the interrupt handlers, called **interrupt vectors,** are stored in a vector table in low memory. For example, if the interrupt is a vertical retrace interrupt, the microprocessor examines the value of the Vertical Retrace Manager's interrupt vector and executes the interrupt handler whose code starts at the address referenced by that value. The vertical retrace interrupt handler might then execute one or more vertical blanking tasks. When an interrupt task is executed, the interrupt is said to be **serviced.**

Each type of interrupt has an **interrupt priority level,** which defines how important it is that an interrupt be serviced. The microprocessor also maintains a **processor priority** that limits which interrupts will be serviced. When a device generates an interrupt whose interrupt priority level is higher than the processor priority level, the processor priority level is raised to the interrupt priority level, the interrupt is serviced, and the processor priority level is lowered to its previous level. When a device generates an interrupt whose interrupt priority level is lower than or equal to the processor priority level, the interrupt is ignored; interrupts of levels lower than the processor priority are said to be **disabled** when higher-level interrupts are executing. This scheme ensures that relatively important interrupts are not themselves interrupted by less important interrupts.

If you are writing a typical application, you do not need to worry about interrupts themselves or the low-level details associated with them. Your application installs interrupt tasks and ordinarily does not need to worry about the interrupts that cause them to execute. For more information on interrupts themselves, see the chapter "Device Manager" in *Inside Macintosh: Devices* and the chapter "Deferred Task Manager" in this book.

Task Scheduling

As previously indicated, your interrupt tasks are executed in response to an interrupt. Because the execution of an interrupt task is not tied to the normal execution of your application, that task might continue to be executed even when your application is not itself executing. For example, all Time Manager tasks installed by your application continue to be executed as scheduled, whether or not your application is still the current application.

If it doesn't make sense to continue executing a particular Time Manager task when your application is no longer receiving processing time, you need to disable the execution of that task whenever your application is switched out and then reenable the task when your application regains control of the CPU. To disable a Time Manager task, you can remove its entry from the Time Manager queue. To reenable it, reinstall its entry in the queue.

In some cases, the Operating System automatically disables some of your application's interrupt tasks when your application is switched out. All VBL tasks installed by the Vertical Retrace Manager routine `VInstall` (which are known as system-based VBL tasks) are disabled whenever the installing application loses control of the CPU, if the address of the task is in the application partition. If you want to continue executing a system-based VBL task when your application is switched out, you must make sure that the address of the task is in the system partition. See the chapter "Vertical Retrace Manager" in this book for details on how to accomplish this.

Note

A VBL task installed by the routine `SlotVInstall` (known as a slot-based VBL task) is always executed as scheduled, regardless of the task's address. u

When an interrupt task is executed, the Operating System does not always restore the installing application's context. As a result, you might not be able to read any application-specific system global variables from within the task. In addition, the task will not have access to any application-installed patches (which are part of its context). If your interrupt task depends on any part of your application's context, it should call the Process Manager function `GetCurrentProcess` to make sure that your process is currently in control of the CPU and hence that its context is valid.

Note

Your interrupt code must also avoid calling traps that access application-specific system global variables, unless you determine that your application's context is valid. In general, however, there is no way to determine whether a trap accesses system global variables. u

Even if your application's context is not valid, you can still access some information in your application's partition if you suitably set up and restore the A5 register within your interrupt task. Your application global variables and your application's jump table are both accessed via an address in the microprocessor's A5 register. If you need to read or write any of your application's global variables or call routines in another segment, you must set up the A5 register with your application's value of the `CurrentA5` global variable. Because you cannot in general inspect `CurrentA5` at interrupt time, you need to read its value at noninterrupt time and pass the value to your interrupt routine. See the chapters "Time Manager" and "Vertical Retrace Manager" in this book for illustrations of a technique you can use for this purpose. For more information on how to set the A5 register properly, see the chapter "Memory Management Utilities" in *Inside Macintosh: Memory*.

If you do call routines in another code segment at interrupt time, you must make sure that the segment is already loaded in memory. Otherwise, the Operating System will call the Segment Manager to load the segment into memory, which could cause memory to be allocated.

s **WARNING**

Interrupt tasks should never directly or indirectly cause memory to be allocated, moved, or purged, because the heap might be in an inconsistent state when the task is executed. s

For this same reason, your interrupt tasks must never depend on the validity of handles that are not locked. The interrupt task might be called in the middle of a memory-allocation request, during which time the Memory Manager might be moving an unlocked block in the heap. If you must access relocatable blocks of heap memory within an interrupt task, make sure to lock those blocks before installing the task.

If virtual memory is available in the current operating environment, you also need to make certain that your interrupt tasks do not attempt to read information in a page of memory that might not be resident in physical RAM. Otherwise, the Operating System will attempt to read the affected pages of memory into physical RAM, which is likely to cause the system to crash. To be safe, you should hold all data and code accessed at interrupt time in physical memory. For details, see the chapter "Virtual Memory Manager" in *Inside Macintosh: Memory*.

Task Guidelines

This section summarizes the guidelines to follow if your application installs tasks that are executed at interrupt time.

- Make your interrupt task as short as possible. A good strategy is to have the interrupt task modify a global variable from which your application can determine what noninterrupt processing to perform. If this strategy is not sufficient, you can use the Deferred Task Manager to defer lengthy interrupt tasks until all interrupts are reenabled.
- If you modify your application's global variables from within an interrupt task or call routines in another code segment, make sure to set up and restore the A5 register. The chapters "Time Manager" and "Vertical Retrace Manager" in this book contain examples of this technique.
- Don't call any routines that cause memory to be moved or compacted, either directly or indirectly.
- Don't use any handles that are not locked.
- Make sure that the code segment containing the interrupt task is loaded, locked, and unpurgeable. Never unload a code segment containing an active interrupt task.
- Do not allocate parameter blocks or task records as local variables of routines that might return before the interrupt task is completed.
- Do not make synchronous calls in an interrupt task.
- Minimize the amount of stack space your task uses. Remember that some interrupt tasks execute at times when your application is not the current application; as a result, you might not be able to predict how much stack space is available to your task.
- Preserve all microprocessor registers other than A0–A3 and D0–D3. Most compilers for high-level languages automatically generate code that does this.

CHAPTER 2

Process Manager

Contents

This chapter describes the Process Manager, the part of the Macintosh Operating System that provides a cooperative multitasking environment. The Process Manager controls access to shared resources and manages the scheduling and execution of applications. The Finder uses the Process Manager to launch your application when the user opens either your application or a document created by your application. This chapter discusses how your application can control its execution and get information—for example, the number of free bytes in the application's heap—about itself or any other open application.

Although earlier versions of system software provide process management, the Process Manager is available to your application only in system software version 7.0 and later. The Process Manager provides a cooperative multitasking environment, similar to the features provided by the MultiFinder option in earlier versions of system software. You can use the `Gestalt` function to find out if the Process Manager routines are available and to see which features of the `Launch` function are available.

You should read the chapter "Introduction to Processes and Tasks" in this book for an overview of how the Process Manager schedules applications and loads them into memory. If your application needs to launch other applications, you need to read this chapter for information on the high-level function that lets your application launch other applications and the routines you can use to get information about open applications.

To use this chapter, you need to be familiar with how your application uses memory, as described in the chapter "Introduction to Memory Management" in *Inside Macintosh: Memory*. You should also be familiar with how your application receives events, as discussed in the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

This chapter provides a brief description of the Process Manager and then shows how you can

- control the execution of your application
- get information about your application
- launch other applications or desk accessories
- get information about applications launched by your application
- generate a list of all open applications and information about each one
- terminate the execution of your application

About the Process Manager

The Process Manager schedules the processing of all applications and desk accessories. It allows multiple applications to share CPU time and other resources. Applications share the available memory and access to the CPU. Several applications can be open (loaded into memory) at once, but only one uses the CPU at any one time.

Note
For a complete description of how the Process Manager schedules applications and desk accessories for execution, see the chapter "Introduction to Processes and Tasks" in this book. u

The Process Manager also provides a number of routines that allow you to control the execution of processes and to get information about processes, including your own. You can use the Process Manager routines to

- control the execution of your application
- get information about processes
- launch other applications
- launch desk accessories

The Process Manager assigns a process serial number to each open application (or desk accessory, if it is not opened in the context of an application). The process serial number is unique to each process on the local computer and is valid for a single boot of the computer. You can use the process serial number to specify a particular process for most Process Manager routines.

When a user opens or prints a file from the Finder, it uses the Process Manager to launch the application that created the file. The Finder sets up the information from which your application can determine which files to open or print. The Finder information includes a list of files to open or print.

In system software version 7.0 and later, applications that support high-level events (that is, that have the `isHighLevelEventAware` flag set in the `'SIZE'` resource) receive the Finder information through Apple events. The chapter "Apple Event Manager" in *Inside Macintosh: Interapplication Communication* describes how your application processes Apple events to open or print files.

Applications that do not support high-level events can call the `CountAppFiles`, `GetAppFiles`, and `ClrAppFiles` routines or the `GetAppParms` routine to get the Finder information. See the chapter "Introduction to File Management" in *Inside Macintosh: Files* for information on these routines.

Using the Process Manager

This section shows how you can use the Process Manager to

- obtain information about open processes
- launch applications and desk accessories
- terminate your application

Getting Information About Other Processes

You can call the `GetNextProcess`, `GetFrontProcess`, or `GetCurrentProcess` functions to get the process serial number of a process. The `GetCurrentProcess` function returns the process serial number of the process currently executing, called the **current process.** This is the process whose A5 world is currently valid; this process can be in the background or foreground. The `GetFrontProcess` function returns the process serial number of the foreground process. For example, if your process is running in the background, you can use `GetFrontProcess` to determine which process is in the foreground.

The Process Manager maintains a list of all open processes. You can specify the process serial number of a process currently in the list and call `GetNextProcess` to get the process serial number of the next process in the list. The interpretation of the value of a process serial number and of the order of the list of processes is internal to the Process Manager.

When specifying a particular process, use only a process serial number returned by a high-level event or a Process Manager routine, or constants defined by the Process Manager. You can use these constants to specify special processes:

```
CONST
   kNoProcess        = 0;        {process doesn't exist}
   kSystemProcess    = 1;        {process belongs to OS}
   kCurrentProcess   = 2;        {the current process}
```

In all Process Manager routines, the constant `kNoProcess` refers to a process that doesn't exist, the constant `kSystemProcess` refers to a process belonging to the Operating System, and the constant `kCurrentProcess` refers to the current process.

To begin enumerating a list of processes, call the `GetNextProcess` function and specify the constant `kNoProcess` as the parameter. In response, `GetNextProcess` returns the process serial number of the first process in the list. You can use the returned process serial number to get the process serial number of the next process in the list. When the `GetNextProcess` function reaches the end of the list, it returns the constant `kNoProcess` and the result code `procNotFound`.

You can also use a process serial number to specify a target application when your application sends a high-level event. See the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for information on how to use a process serial number when your application sends a high-level event.

You can call the `GetProcessInformation` function to obtain information about any process, including your own. For example, for a specified process, you can find

- the application's name as it appears in the Application menu
- the type and signature of the application
- the number of bytes in the application partition
- the number of free bytes in the application heap
- the application that launched the application

The `GetProcessInformation` function returns information in a process information record, which is defined by the `ProcessInfoRec` data type.

```
TYPE ProcessInfoRec =
   RECORD
      processInfoLength:   LongInt;       {length of process info record}
      processName:         StringPtr;     {name of this process}
      processNumber:       ProcessSerialNumber;
                                          {psn of this process}
      processType:         LongInt;       {file type of application file}
      processSignature:    OSType;        {signature of application file}
      processMode:         LongInt;       {'SIZE' resource flags}
      processLocation:     Ptr;           {address of partition}
      processSize:         LongInt;       {partition size}
      processFreeMem:      LongInt;       {free bytes in heap}
      processLauncher:     ProcessSerialNumber;
                                          {process that launched this one}
      processLaunchDate:   LongInt;       {time when launched}
      processActiveTime:   LongInt;       {accumulated CPU time}
      processAppSpec:      FSSpecPtr;     {location of the file}
   END;
```

You specify the values for three fields of the process information record: `processInfoLength`, `processName`, and `processAppSpec`. You must either set the `processName` and `processAppSpec` fields to `NIL` or set these fields to point to memory that you have allocated for them. The `GetProcessInformation` function returns information in all other fields of the process information record. See "Process Information Record" on page 2-16 for a complete description of the fields of this record.

Listing 2-1 shows how you can use the `GetNextProcess` function with the `GetProcessInformation` function to search the process list for a specific process.

Listing 2-1 Searching for a specific process

```
FUNCTION FindAProcess (signature: OSType;
                       VAR process: ProcessSerialNumber;
                       VAR InfoRec: ProcessInfoRec;
                       myFSSpecPtr: FSSpecPtr;
                       myName: Str31): Boolean;
BEGIN
   FindAProcess := FALSE;              {assume FALSE}
   process.highLongOfPSN := 0;
   process.lowLongOfPSN := kNoProcess; {start at the beginning}

   InfoRec.processInfoLength := sizeof(ProcessInfoRec);
```

```
   InfoRec.processName := myName;
   InfoRec.processAppSpec := myFSSpecPtr;

   WHILE (GetNextProcess(process) = noErr) DO
   BEGIN
      IF GetProcessInformation(process, InfoRec) = noErr THEN
      BEGIN
         IF (InfoRec.processType = LongInt('APPL')) AND
            (InfoRec.processSignature = signature) THEN
         BEGIN                          {found the process}
            FindAProcess := TRUE;
            Exit(FindAProcess);
         END;
      END;
   END; {WHILE}
END;
```

The code in Listing 2-1 searches the process list for the application with the specified signature. For example, you might want to find a specific process so that you can send a high-level event to it.

Launching Other Applications

You can launch other applications by calling the high-level `LaunchApplication` function. This function lets your application control various options associated with launching an application. For example, you can

- allow the application to be launched in a partition smaller than the preferred size but greater than the minimum size, or allow it to be launched only in a partition of the preferred size
- launch an application without terminating your own application, bring the launched application to the front, and get information about the launched application
- request that your application be notified if any application that it has launched terminates

Earlier versions of system software used a shorter parameter block as a parameter to the `_Launch` trap macro. The `_Launch` trap macro still supports the use of this parameter block. Applications using the `LaunchApplication` function should use the new launch parameter block (of type `LaunchParamBlockRec`). Use the `Gestalt` function and specify the selector `gestaltOSAttr` to determine which launch features are available.

Most applications don't need to launch other applications. However, if your application includes a desk accessory or another application, you might use either the high-level `LaunchApplication` function to launch an application or the `LaunchDeskAccessory` function to launch a desk accessory. For example, if you have implemented a spelling checker as a separate application, you might use the

LaunchApplication function to open the spelling checker when the user chooses Check Spelling from one of your application's menus.

You specify a launch parameter block as a parameter to the LaunchApplication function. In this launch parameter block, you can specify the filename of the application to launch, specify whether to allow launching only in a partition of the preferred size or to allow launching in a smaller partition, and set various other options—for example, whether your application should continue or terminate after it launches the specified application.

The LaunchApplication function launches the application from the specified file and returns the process serial number, preferred partition size, and minimum partition size if the application is successfully launched.

Note that if you launch another application without terminating your application, the launched application does not actually begin executing until you make a subsequent call to WaitNextEvent or EventAvail.

The launch parameter block is defined by the LaunchParamBlockRec data type.

```
TYPE LaunchParamBlockRec =
   RECORD
      reserved1:              LongInt;             {reserved}
      reserved2:              Integer;             {reserved}
      launchBlockID:          Integer;             {extended block}
      launchEPBLength:        LongInt;             {length of block}
      launchFileFlags:        Integer;             {app's Finder flags}
      launchControlFlags:     LaunchFlags;         {launch options}
      launchAppSpec:          FSSpecPtr;           {location of app's file}
      launchProcessSN:        ProcessSerialNumber; {returned psn}
      launchPreferredSize:    LongInt;             {returned pref size}
      launchMinimumSize:      LongInt;             {returned min size}
      launchAvailableSize:    LongInt;             {returned avail size}
      launchAppParameters:    AppParametersPtr;    {high-level event}
   END;
```

In the launchBlockID field, specify the constant extendedBlock to identify the parameter block and to indicate that you are using the fields following it in the launch parameter block.

```
CONST
   extendedBlock              = $4C43;    {extended block}
```

In the launchEPBLength field, specify the constant extendedBlockLen to indicate the length of the remaining fields in the launch parameter block (that is, the length of the fields following the launchEPBLength field). For compatibility, you should always specify the length value in this field.

```
CONST
   extendedBlockLen            = sizeof(LaunchParamBlockRec) - 12;
```

The launchFileFlags field contains the Finder flags for the application file. (See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for a description of the Finder flags.) The LaunchApplication function sets this field for you if you set the bit defined by the launchNoFileFlags constant in the launchControlFlags field. Otherwise, you must get the Finder flags from the application file and set this field yourself (by using the File Manager routine FSpGetFInfo, for example).

In the launchControlFlags field, you specify various options that control how the specified application is launched. See the section "Launch Options" on page 2-15 for information on the launch control flags.

You specify the application to launch in the launchAppSpec field of the launch parameter block. In this field, you specify a pointer to a file system specification record (FSSpec). See the chapter "File Manager" in *Inside Macintosh: Files* for a complete description of the file system specification record.

The LaunchApplication function sets the initial default directory of the application to the parent directory of the application file.

If it successfully launches the application, LaunchApplication returns, in the launchProcessSN field, a process serial number. You can use this number in Process Manager routines to refer to this application.

The LaunchApplication function returns the launchPreferredSize and launchMinimumSize fields of the launch parameter block. The values of these fields are based on their corresponding values in the 'SIZE' resource. These values may be greater than those specified in the application's 'SIZE' resource because the returned sizes include any adjustments to the size of the application's stack. See the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for information on how the size of the application stack is adjusted. Values are always returned in these fields whether or not the launch was successful. These values are 0 if an error occurred—for example, if the application file could not be found.

The LaunchApplication function returns a value in the launchAvailableSize field only when the memFullErr result code is returned. This value indicates the largest partition size currently available for allocation.

The launchAppParameters field specifies the first high-level event sent to an application. If you set this field to NIL, the LaunchApplication function automatically creates and sends an Open Application event to the launched application. (See the chapter "Apple Event Manager" in *Inside Macintosh: Interapplication Communication* for a description of this event.) To send a particular high-level event to

the launched application, you can specify a pointer to an application parameters record. The application parameters record is defined by the data type `AppParameters`.

```
TYPE AppParameters =
   RECORD
      theMsgEvent:      EventRecord;       {event (high-level)}
      eventRefCon:      LongInt;           {reference constant}
      messageLength:    LongInt;           {length of buffer}
      messageBuffer:    ARRAY [0..0] OF SignedByte;
   END;
```

You specify the high-level event in the fields `theMsgEvent`, `eventRefCon`, `messageLength`, and `messageBuffer`.

Listing 2-2 demonstrates how you can use the `LaunchApplication` function.

Listing 2-2 Launching an application

```
PROCEDURE LaunchAnApplication (mySFReply: StandardFileReply);
VAR
   myLaunchParams:      LaunchParamBlockRec;
   launchedProcessSN:   ProcessSerialNumber;
   launchErr:           OSErr;
   prefSize:            LongInt;
   minSize:             LongInt;
   availSize:           LongInt;
BEGIN
   WITH myLaunchParams DO
   BEGIN
      launchBlockID := extendedBlock;
      launchEPBLength := extendedBlockLen;
      launchFileFlags := 0;
      launchControlFlags := launchContinue + launchNoFileFlags;
      launchAppSpec := @mySFReply.sfFile;
      launchAppParameters := NIL;
   END;
   launchErr := LaunchApplication(@myLaunchParams);

   prefsize := myLaunchParams.launchPreferredSize;
   minsize := myLaunchParams.launchMinimumSize;
   IF launchErr = noErr THEN
      launchedProcessSN := myLaunchParams.launchProcessSN
   ELSE IF launchErr = memFullErr THEN
      availSize := myLaunchParams.launchAvailableSize
```

```
   ELSE
      DoError(launchErr);
END;
```

Listing 2-2 indicates which application file to launch by using a file system specification record (perhaps returned by the `StandardGetFile` routine) and specifying, in the `launchAppSpec` field, a pointer to this record. The `launchControlFlags` field indicates that `LaunchApplication` should extract the Finder flags from the application file, launch the application in a partition of the preferred size, bring the launched application to the front, and not terminate the current process.

By default, `LaunchApplication` brings the launched application to the front and sends the foreground application to the background. If you don't want to bring an application to the front when it is first launched, set the `launchDontSwitch` flag in the `launchControlFlags` field of the launch parameter block.

In addition, if you want your application to continue to run after it launches another application, you must set the `launchContinue` flag in the `launchControlFlags` field of the launch parameter block. For a complete description of the available launch control options, see "Launch Options" on page 2-15.

If you want your application to be notified about the termination of an application it has launched, set the `acceptAppDiedEvents` flag in your `'SIZE'` resource. If you set this flag and an application launched by your application terminates, your application receives an Application Died Apple event (`'aevt' 'obit'`). See "Terminating an Application" on page 2-11 for more information on the Application Died event.

Launching Desk Accessories

In system software version 7.0 and later, the Process Manager launches a desk accessory in its own partition when that desk accessory is opened, giving it a process serial number and an entry in the process list. The Process Manager puts the name of the desk accessory in the list of open applications in the Application menu and also gives the active desk accessory its own About menu item in the Apple menu containing the name of the desk accessory. This makes desk accessories more consistent with the user interface of small applications.

Although you can use the `LaunchDeskAccessory` function to launch desk accessories, you should use it only when your application needs to launch a desk accessory for some reason other than the user's choosing a desk accessory from the Apple menu. Beginning in system software version 7.0, the Apple menu can contain any Finder object that the user decides to add to the menu. When the user chooses any such user-added item from the Apple menu, your application should respond by calling the `OpenDeskAcc` function instead.

Terminating an Application

The Process Manager automatically terminates a process when the process either exits its main routine or encounters a fatal error condition (such as an attempt to divide by 0).

When a process terminates, the Process Manager takes care of any required cleanup operations; these include removing the process from the list of open processes and releasing the memory occupied by the application partition (as well as any temporary memory the process still holds). If necessary, the Process Manager sends an Application Died event to the process that launched the one about to terminate.

Your application can also terminate itself directly by calling the `ExitToShell` procedure. In general, you need to call `ExitToShell` only if you want to terminate your application without having it return from its main routine. This might be useful when your initialization code detects that some essential system capability is not available (for instance, when the computer running a stereo sound-editing application does not support stereo sound playback). Listing 2-3 shows one way to exit gracefully in this situation.

Listing 2-3 Terminating an application

```
PROCEDURE CheckForStereoSound;
VAR
   myErr:      OSErr;           {result code from Gestalt}
   myFeature:  LongInt;         {features bit flags from Gestalt}
   myString:   Str255;          {text of alert message}
   myItem:     Integer;         {item returned by StopAlert}
CONST
   kAlertBoxID    = 128;        {resource ID of alert template}
   kAlertStrings  = 128;        {resource ID of alert strings}
   kNoStereoAlert = 5;          {index of No Stereo alert text}
BEGIN
   myErr := Gestalt(gestaltSoundAttr, myFeature);
   IF myErr = noErr THEN
      IF BTst(myFeature, gestaltStereoCapability) = FALSE THEN
         BEGIN
            GetIndString(myString, kAlertStrings, kNoStereoAlert);
            ParamText(myString, '', '', '');
            myItem := StopAlert(kAlertBoxID, NIL);
            ExitToShell;        {exit the application}
         END
   ELSE
      DoError(myErr);
END;
```

The procedure `CheckForStereoSound` defined in Listing 2-3 checks whether the computer supports stereo sound playback. If not, `CheckForStereoSound` notifies the user by displaying an alert box and terminates the application by calling `ExitToShell`.

Note
The `ExitToShell` procedure is the only means of terminating a process. It is always called during process termination, whether by your application itself, the Process Manager, or some other process. u

If your application launches another application that terminates, either normally or as the result of an error, the Process Manager can notify your application by sending it an Application Died event. To request this notification, you must set the `acceptAppDied` flag in your application's `'SIZE'` resource. (For a complete description of the `'SIZE'` resource, see the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.)

Application Died—inform that an application has terminated

Event ID	`kAEApplicationDied`
Required parameters	
Keyword	`keyErrorNumber`
Descriptor type	`typeLongInteger`
Data	A sign-extended `OSErr` value. A value of `noErr` indicates normal termination; any other value indicates that the application terminated because of an error.
Keyword	`keyProcessSerialNumber`
Descriptor type	`typeProcessSerialNumber`
Data	The process serial number of the application that terminated.
Requested action	None. This Apple event is sent only to provide information.

The Process Manager gets the value of the `keyErrorNumber` parameter from the system global variable `DSErrCode`. This value can be set either by the application before it terminates or by the Operating System (when the application terminates as the result of a hardware exception or other problem).

Process Manager Reference

This section describes the constants, data structures, and routines that are specific to the Process Manager.

Constants

You can use Process Manager constants to get information about the attributes of the Process Manager, identify certain special processes, and specify launch options.

Gestalt Selector and Response Bits

You can determine if the Process Manager is available and find out which features of the launch routine are available by calling the `Gestalt` function with the selector `gestaltOSAttr`.

```
CONST
   gestaltOSAttr              = 'os  ';{O/S attributes}
```

The `Gestalt` function returns information by setting or clearing bits in the `response` parameter. The following constants define the bits currently used:

```
CONST
   gestaltLaunchCanReturn     = 1;  {can return from launch}
   gestaltLaunchFullFileSpec  = 2;  {LaunchApplication available}
   gestaltLaunchControl       = 3;  {Process Manager is available}
```

Constant descriptions

`gestaltLaunchCanReturn`
Set if the `_Launch` trap macro can return to the caller. The `_Launch` trap macro in system software version 7.0 (and in earlier versions running MultiFinder) gives your application the option to continue running after it launches another application. In earlier versions of system software not running MultiFinder, the `_Launch` trap macro forces the launching application to quit.

`gestaltLaunchFullFileSpec`
Set if the `launchControlFlags` field supports control flags in addition to the `launchContinue` flag, and if the `_Launch` trap can process the `launchAppSpec`, `launchProcessSN`, `launchPreferredSize`, `launchMinimumSize`, `launchAvailableSize`, and `launchAppParameters` fields in the launch parameter block.

`gestaltLaunchControl`
Set if the Process Manager is available.

Process-Identification Constants

The Process Manager provides three constants that can be used instead of a process serial number to identify a process:

```
CONST
   kNoProcess        = 0;        {process doesn't exist}
   kSystemProcess    = 1;        {process belongs to OS}
   kCurrentProcess   = 2;        {the current process}
```

Constant descriptions

`kNoProcess` Identifies a process that doesn't exist.

`kSystemProcess`
Identifies a process that belongs to the Operating System.

`kCurrentProcess`
Identifies the current process.

Launch Options

When you use the `LaunchApplication` function, you specify the launch options in the `launchControlFlags` field of the launch parameter block. These are the constants you can specify in the `launchControlFlags` field:

```
CONST
   launchContinue          = $4000;
   launchNoFileFlags       = $0800;
   launchUseMinimum        = $0400;
   launchDontSwitch        = $0200;
   launchInhibitDaemon     = $0080;
```

Constant descriptions

`launchContinue` Set this flag if you want your application to continue after the specified application is launched. If you do not set this flag, `LaunchApplication` terminates your application after launching the specified application, even if the launch fails.

`launchNoFileFlags`
Set this flag if you want the `LaunchApplication` function to ignore any value specified in the `launchFileFlags` field. If you set the `launchNoFileFlags` flag, the `LaunchApplication` function extracts the Finder flags from the application file for you. If you want to supply the file flags, clear the `launchNoFileFlags` flag and specify the Finder flags in the `launchFileFlags` field of the launch parameter block.

`launchUseMinimum`
Clear this flag if you want the `LaunchApplication` function to attempt to launch the application in the preferred size (as specified in the application's `'SIZE'` resource). If you set the `launchUseMinimum` flag, the `LaunchApplication` function attempts to launch the application using the largest available size greater than or equal to the minimum size but less than the preferred size. If the `LaunchApplication` function returns the result code `memFullErr` or `memFragErr`, the application cannot be launched under the current memory conditions.

`launchDontSwitch`
Set this flag if you do not want the launched application brought to the front. If you set this flag, the launched application runs in the background until the user brings the application to the front—for

example, by clicking in one of the application's windows. Note that most applications expect to be launched in the foreground. If you clear the `launchDontSwitch` flag, the launched application is brought to the front, and your application is sent to the background.

`launchInhibitDaemon`
Set this flag if you do not want `LaunchApplication` to launch a background-only application. (A background-only application has the `onlyBackground` flag set in its `'SIZE'` resource.)

Data Structures

This section describes the data structures that you use to provide information to the Process Manager or that the Process Manager uses to return information to your application.

Process Serial Number

The Process Manager uses process serial numbers to identify open processes. A process serial number is a 64-bit quantity whose structure is defined by the `ProcessSerialNumber` data type.

IMPORTANT

The meaning of the bits in a process serial number is internal to the Process Manager. You should not attempt to interpret the value of the process serial number. If you need to compare two process serial numbers, call the `SameProcess` function. s

```
TYPE ProcessSerialNumber   =
   RECORD
      highLongOfPSN:    LongInt;    {high-order 32 bits of psn}
      lowLongOfPSN:     LongInt;    {low-order 32 bits of psn}
   END;
```

Field descriptions

`highLongOfPSN` The high-order long integer of the process serial number.

`lowLongOfPSN` The low-order long integer of the process serial number.

Process Information Record

The `GetProcessInformation` function returns information in a process information record, which is defined by the `ProcessInfoRec` data type.

```
TYPE ProcessInfoRec =
   RECORD
      processInfoLength:   LongInt;       {length of process info record}
      processName:         StringPtr;     {name of this process}
```

```
   processNumber:       ProcessSerialNumber;
                                {psn of this process}
   processType:         LongInt;       {file type of application file}
   processSignature:    OSType;        {signature of application file}
   processMode:         LongInt;       {'SIZE' resource flags}
   processLocation:     Ptr;           {address of partition}
   processSize:         LongInt;       {partition size}
   processFreeMem:      LongInt;       {free bytes in heap}
   processLauncher:     ProcessSerialNumber;
                                {process that launched this one}
   processLaunchDate:   LongInt;       {time when launched}
   processActiveTime:   LongInt;       {accumulated CPU time}
   processAppSpec:      FSSpecPtr;     {location of the file}
END;
```

Field descriptions

`processInfoLength`
: The number of bytes in the process information record. For compatibility, you should specify the length of the record in this field.

`processName`
: The name of the application or desk accessory. For applications, this field contains the name of the application as designated by the user at the time the application was opened. For example, for foreground applications, the `processName` field contains the name as it appears in the Application menu. For desk accessories, the `processName` field contains the name of the `'DRVR'` resource. You must specify `NIL` in the `processName` field if you do not want the application name or the desk accessory name returned. Otherwise, you should allocate at least 32 bytes of storage for the string pointed to by the `processName` field. Note that the `processName` field specifies the name of either the application or the `'DRVR'` resource, whereas the `processAppSpec` field specifies the location of the file.

`processNumber`
: The process serial number. The process serial number is a 64-bit number; the meaning of these bits is internal to the Process Manager. You should not attempt to interpret the value of the process serial number.

`processType`
: The file type of the application, generally `'APPL'` for applications and `'appe'` for background-only applications launched at startup. If the process is a desk accessory, this field specifies the type of the file containing the `'DRVR'` resource.

`processSignature`
: The signature of the file containing the application or the `'DRVR'` resource (for example, the signature of the TeachText application is `'ttxt'`).

`processMode` Process mode flags. These flags indicate whether the process is an application or desk accessory. For applications, this field also returns information specified in the application's `'SIZE'` resource. This information is returned as flags. You can refer to these flags by using these constants:

```
CONST
   modeDeskAccessory             = $00020000;
   modeMultiLaunch               = $00010000;
   modeNeedSuspendResume         = $00004000;
   modeCanBackground             = $00001000;
   modeDoesActivateOnFGSwitch    = $00000800;
   modeOnlyBackground            = $00000400;
   modeGetFrontClicks            = $00000200;
   modeGetAppDiedMsg             = $00000100;
   mode32BitCompatible           = $00000080;
   modeHighLevelEventAware       = $00000040;
   modeLocalAndRemoteHLEvents    = $00000020;
   modeStationeryAware           = $00000010;
   modeUseTextEditServices       = $00000008;
```

`processLocation`
The beginning address of the application partition.

`processSize` The number of bytes in the application partition (including the heap, stack, and A5 world).

`processFreeMem`
The number of free bytes in the application heap.

`processLauncher`
The process serial number of the process that launched the application or desk accessory. If the original launcher of the process is no longer open, this field contains the constant `kNoProcess`.

`processLaunchDate`
The value of the `Ticks` global variable at the time that the process was launched.

`processActiveTime`
The accumulated time, in ticks, during which the process has used the CPU, including both foreground and background processing time.

`processAppSpec`
The address of a file specification record that stores the location of the file containing the application or `'DRVR'` resource. You should specify `NIL` in the `processAppSpec` field if you do not want the `FSSpec` record of the file returned.

Launch Parameter Block

You specify a launch parameter block as a parameter to the `LaunchApplication` function. The launch parameter block is defined by the `LaunchParamBlockRec` data type.

```
TYPE LaunchParamBlockRec =
   RECORD
      reserved1:              LongInt;             {reserved}
      reserved2:              Integer;             {reserved}
      launchBlockID:          Integer;             {extended block}
      launchEPBLength:        LongInt;             {length of block}
      launchFileFlags:        Integer;             {app's Finder flags}
      launchControlFlags:     LaunchFlags;         {launch options}
      launchAppSpec:          FSSpecPtr;           {location of app's file}
      launchProcessSN:        ProcessSerialNumber; {returned psn}
      launchPreferredSize:    LongInt;             {returned pref size}
      launchMinimumSize:      LongInt;             {returned min size}
      launchAvailableSize:    LongInt;             {returned avail size}
      launchAppParameters:    AppParametersPtr;    {high-level event}
   END;
```

Field descriptions

`reserved1` Reserved.

`reserved2` Reserved.

`launchBlockID` A value that indicates whether you are using the fields following it in the launch parameter block. Specify the constant `extendedBlock` if you use the fields that follow it.

`launchEPBLength`
The length of the fields following this field in the launch parameter block. Use the constant `extendedBlockLen` to specify this value.

`launchFileFlags`
The Finder flags for the application file. Set the `launchNoFileFlags` constant in the `launchControlFlags` field if you want the `LaunchApplication` function to extract the Finder flags from the application file and to set the `launchFileFlags` field for you.

`launchControlFlags`
The launch options that determine how the application is launched. You can specify these constant values to set various options:

```
CONST
   launchContinue       = $4000;
   launchNoFileFlags    = $0800;
```

```
launchUseMinimum     = $0400;
launchDontSwitch     = $0200;
launchInhibitDaemon  = $0080;
```

See “Launch Options” on page 2-15 for a complete description of these flags.

`launchAppSpec` A pointer to a file specification record that gives the location of the application file to launch.

`launchProcessSN`
The process serial number returned to your application if the launch is successful. You can use this process serial number in other Process Manager routines to refer to the launched application.

`launchPreferredSize`
The preferred partition size for the launched application as specified in the launched application’s `'SIZE'` resource. `LaunchApplication` sets this field to 0 if an error occurred or if the application is already open.

`launchMinimumSize`
The minimum partition size for the launched application as specified in the launched application’s `'SIZE'` resource. `LaunchApplication` sets this field to 0 if an error occurred or if the application is already open.

`launchAvailableSize`
The maximum partition size that is available for allocation. This value is returned to your application only if the `memFullErr` result code is returned. If the application launch fails because of insufficient memory, you can use this value to determine if there is enough memory available to launch in the minimum size.

`launchAppParameters`
The first high-level event to send to the launched application. If you set this field to `NIL`, `LaunchApplication` creates and sends the Open Application Apple event to the launched application.

Application Parameters Record

You specify an application parameters record in the `launchAppParameters` field of the launch parameter block whose address is passed to the `LaunchApplication` function. This record specifies the first high-level event to be sent to the newly launched application. The application parameters record is defined by the `AppParameters` data type.

```
TYPE AppParameters =
   RECORD
      theMsgEvent:      EventRecord;       {event (high-level)}
      eventRefCon:      LongInt;           {reference constant}
```

```
    messageLength:    LongInt;          {length of buffer}
    messageBuffer:    ARRAY [0..0] OF SignedByte;
END;
```

Field descriptions

`theMsgEvent`	The event record specifying the first high-level event to be sent to the launched application.
`eventRefCon`	A reference constant. Your application can use this field for its own purposes.
`messageLength`	The length of the buffer specified by the `messageBuffer` field.
`messageBuffer`	A buffer of data. The nature of this data varies according to the event being sent.

Routines

This section describes the Process Manager routines you can use to get information about any currently open applications, to control process execution, to launch other applications, and to terminate your application.

Getting Process Information

You can use the Process Manager to get the process serial number of a particular process, to generate a list of all open processes, to get information about processes, or to change the scheduling status of a process.

GetCurrentProcess

Use the `GetCurrentProcess` function to get information about the current process, if any.

```
FUNCTION GetCurrentProcess (VAR PSN: ProcessSerialNumber): OSErr;
```

`PSN` On output, the process serial number of the current process.

DESCRIPTION

The `GetCurrentProcess` function returns, in the `PSN` parameter, the process serial number of the process that is currently running, that is, the one currently accessing the CPU. This is the application associated with the `CurrentA5` global variable. This application can be running in either the foreground or the background.

Applications can use this function to find their own process serial number. Drivers can use this function to find the process serial number of the current process. You can use the returned process serial number in other Process Manager routines.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `GetCurrentProcess` function are

Trap macro	Selector
`_OSDispatch`	$0037

RESULT CODE

`noErr`	0	No error

GetNextProcess

Use the `GetNextProcess` function to get information about the next process, if any, in the Process Manager's internal list of open processes.

```
FUNCTION GetNextProcess (VAR PSN: ProcessSerialNumber): OSErr;
```

`PSN` On input, the process serial number of a process. This number should be a valid process serial number returned from `LaunchApplication`, `GetNextProcess`, `GetFrontProcess`, or `GetCurrentProcess`, or else the defined constant `kNoProcess`. On output, the process serial number of the next process, or else `kNoProcess`.

DESCRIPTION

The Process Manager maintains a list of all open processes. You can derive this list by using repetitive calls to `GetNextProcess`. Begin generating the list by calling `GetNextProcess` and specifying the constant `kNoProcess` in the `PSN` parameter. You can then use the returned process serial number to get the process serial number of the next process. Note that the order of the list of processes is internal to the Process Manager. When the end of the list is reached, `GetNextProcess` returns the constant `kNoProcess` in the `PSN` parameter and the result code `procNotFound`.

You can use the returned process serial number in other Process Manager routines. You can also use this process serial number to specify a target application when your application sends a high-level event.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `GetNextProcess` function are

Trap macro	Selector
`_OSDispatch`	$0038

RESULT CODES

noErr	0	No error
paramErr	–50	Process serial number is invalid
procNotFound	–600	No process in the process list following the specified process

GetProcessInformation

Use the GetProcessInformation function to get information about a specific process.

```
FUNCTION GetProcessInformation (PSN: ProcessSerialNumber;
                                VAR info: ProcessInfoRec): OSErr;
```

PSN — The process serial number of a process. This number should be a valid process serial number returned from LaunchApplication, GetNextProcess, GetFrontProcess, GetCurrentProcess, or else a high-level event. You can use the constant kCurrentProcess to get information about the current process.

info — A record containing information about the specified process.

DESCRIPTION

The GetProcessInformation function returns, in a process information record, information about the specified process. The information returned in the info parameter includes the application's name as it appears in the Application menu, the type and signature of the application, the address of the application partition, the number of bytes in the application partition, the number of free bytes in the application heap, the application that launched the application, the time at which the application was launched, and the location of the application file. See "Getting Information About Other Processes" on page 2-5 for the structure of the process information record.

The GetProcessInformation function also returns information about the application's 'SIZE' resource and indicates whether the process is an application or a desk accessory.

You need to specify values for the processInfoLength, processName, and processAppSpec fields of the process information record. Specify the length of the process information record in the processInfoLength field. If you do not want information returned in the processName and processAppSpec fields, specify NIL for these fields. Otherwise, allocate at least 32 bytes of storage for the string pointed to by the processName field and, in the processAppSpec field, specify a pointer to an FSSpec record.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `GetProcessInformation` function are

Trap macro	Selector
`_OSDispatch`	$003A

SPECIAL CONSIDERATIONS

Do not call `GetProcessInformation` at interrupt time.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Process serial number is invalid

SameProcess

Use the `SameProcess` function to determine whether two process serial numbers specify the same process.

```
FUNCTION SameProcess (PSN1, PSN2: ProcessSerialNumber;
                      VAR result: Boolean): OSErr;
```

`PSN1`	A process serial number.
`PSN2`	A process serial number.
`result`	A Boolean value that indicates whether the process serial numbers passed in `PSN1` and `PSN2` refer to the same process.

DESCRIPTION

The `SameProcess` function compares two process serial numbers and determines whether they refer to the same process. If the process serial numbers specified in the `PSN1` and `PSN2` parameters refer to the same process, the `SameProcess` function returns `TRUE` in the `result` parameter; otherwise, it returns `FALSE` in the `result` parameter.

Do not attempt to compare two process serial numbers by any means other than the `SameProcess` function, because the interpretation of the bits in a process serial number is internal to the Process Manager.

The values of `PSN1` and `PSN2` must be valid process serial numbers returned from `LaunchApplication`, `GetNextProcess`, `GetFrontProcess`, `GetCurrentProcess`, or a high-level event. You can also use the constant `kCurrentProcess` to refer to the current process.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SameProcess` function are

Trap macro	Selector
`_OSDispatch`	$003D

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Process serial number is invalid

GetFrontProcess

Use the `GetFrontProcess` function to get the process serial number of the front process.

```
FUNCTION GetFrontProcess (VAR PSN: ProcessSerialNumber): OSErr;
```

`PSN` On output, the process serial number of the process running in the foreground.

DESCRIPTION

The `GetFrontProcess` function returns, in the `PSN` parameter, the process serial number of the process running in the foreground. You can use this function to determine if your process or some other process is in the foreground. You can use the process serial number returned in the `PSN` parameter in other Process Manager routines.

If no process is running in the foreground, `GetFrontProcess` returns the result code `procNotFound`.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `GetFrontProcess` function are

Trap macro	Selector
`_OSDispatch`	$0039

RESULT CODES

noErr	0	No error
paramErr	–50	Process serial number is invalid
procNotFound	–600	No process in the foreground

SetFrontProcess

Use the SetFrontProcess function to set the front process.

```
FUNCTION SetFrontProcess (PSN: ProcessSerialNumber): OSErr;
```

PSN — The process serial number of the process you want to move to the foreground. This number should be a valid process serial number returned from LaunchApplication, GetNextProcess, GetFrontProcess, GetCurrentProcess, or a high-level event. You can also use the constant kCurrentProcess to refer to the current process.

DESCRIPTION

The SetFrontProcess function schedules the specified process to move to the foreground. The specified process moves to the foreground after the current foreground process makes a subsequent call to WaitNextEvent or EventAvail.

If the specified process serial number is invalid or if the specified process is a background-only application, SetFrontProcess returns a nonzero result code and does not change the current foreground process.

If a modal dialog box is the frontmost window, the specified process remains in the background until the user dismisses the modal dialog box.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the SetFrontProcess function are

Trap macro	Selector
_OSDispatch	$003B

SPECIAL CONSIDERATIONS

Do not call SetFrontProcess interrupt time.

RESULT CODES

noErr	0	No error
procNotFound	–600	Process with specified process serial number doesn't exist or process is suspended by high-level debugger
appIsDaemon	–606	Specified process runs only in the background

WakeUpProcess

Use the WakeUpProcess function to make a process suspended by WaitNextEvent eligible to receive CPU time.

```
FUNCTION WakeUpProcess (PSN: ProcessSerialNumber): OSErr;
```

PSN — The process serial number of the process to be made eligible. This number should be a valid process serial number returned from LaunchApplication, GetNextProcess, GetFrontProcess, GetCurrentProcess, or a high-level event. You can also use the constant kCurrentProcess to refer to the current process.

DESCRIPTION

The WakeUpProcess function makes a process suspended by WaitNextEvent eligible to receive CPU time. A process is suspended when the value of the sleep parameter in the WaitNextEvent function is not 0 and no events for that process are pending in the event queue. This process remains suspended until the time specified in the sleep parameter expires or an event becomes available for that process. You can use WakeUpProcess to make the process eligible for execution before the time specified in the sleep parameter expires.

The WakeUpProcess function does not change the order of the processes scheduled for execution; it only makes the specified process eligible for execution.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the WakeUpProcess function are

Trap macro	Selector
_OSDispatch	$003C

RESULT CODES

noErr	0	No error
procNotFound	–600	Suspended process with specified process serial number doesn't exist

Launching Applications and Desk Accessories

Your application can use the `LaunchApplication` function to launch other applications and the `LaunchDeskAccessory` function to launch desk accessories.

LaunchApplication

You can use the `LaunchApplication` function to launch an application.

```
FUNCTION LaunchApplication (LaunchParams: LaunchPBPtr): OSErr;
```

`LaunchParams`
A pointer to a launch parameter block specifying information about the application to launch.

Parameter block

→	launchBlockID	Integer	Extended block
→	launchEPBLength	LongInt	Length of following fields
→	launchFileFlags	Integer	Finder flags for the application file
→	launchControlFlags	LaunchFlags	Flags for launch options
→	launchAppSpec	FSSpecPtr	Location of application file to launch
←	launchProcessSN	ProcessSerialNumber	Process serial number
←	launchPreferredSize	LongInt	Preferred application partition size
←	launchMinimumSize	LongInt	Minimum application partition size
←	launchAvailableSize	LongInt	Maximum available partition size
→	launchAppParameters	AppParametersPtr	High-level event for launched application

DESCRIPTION

The `LaunchApplication` function launches the application from the specified file and returns the process serial number, preferred partition size, and minimum partition size if the application is successfully launched.

Note that if you launch another application without terminating your application, the launched application is not actually executed until you make a subsequent call to `WaitNextEvent` or `EventAvail`.

Set the `launchContinue` flag in the `launchControlFlags` field of the launch parameter block if you want your application to continue after the specified application is launched. If you do not set this flag, `LaunchApplication` terminates your application after launching the specified application, even if the launch fails.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and registers on entry and exit for `LaunchApplication` are

Trap macro

`_Launch`

Registers on entry

A0 Pointer to launch parameter block

Registers on exit

A0 Pointer to launch parameter block

D0 Result code

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough memory to allocate the partition size specified in the `'SIZE'` resource
`memFragErr`	–601	Not enough room to launch application with special requirements
`appModeErr`	–602	Memory mode is 32-bit, but application is not 32-bit clean
`appMemFullErr`	–605	More memory is required for the partition size than the amount specified in the `'SIZE'` resource
`appIsDaemon`	–606	Application runs only in the background, and launch flags don't allow background-only applications

LaunchDeskAccessory

You can use the `LaunchDeskAccessory` function to launch desk accessories. Use this function only when your application needs to launch a desk accessory for some reason other than the user's choosing one from the Apple menu. (When the user chooses any Apple menu item that is not specific to your application, use the `OpenDeskAcc` function.)

```
FUNCTION LaunchDeskAccessory (pFileSpec: FSSpecPtr;
                              pDAName: StringPtr): OSErr;
```

`pFileSpec` A pointer to a file system specification of the resource fork to search for the specified desk accessory.

`pDAName` The name of the `'DRVR'` resource to launch.

DESCRIPTION

The `LaunchDeskAccessory` function searches the resource fork of the file specified by the `pFileSpec` parameter for the desk accessory with the `'DRVR'` resource name specified in the `pDAName` parameter. If the `'DRVR'` resource name is found, `LaunchDeskAccessory` launches the corresponding desk accessory. If the desk accessory is already open, it is brought to the front.

Use the `pFileSpec` parameter to specify the file to search. Specify `NIL` as the value of `pFileSpec` if you want to search the current resource file and the resource files opened before it. Otherwise, use a pointer to an `FSSpec` record to specify the file.

In the `pDAName` parameter, specify the `'DRVR'` resource name of the desk accessory to launch. Specify `NIL` as the value of `pDAName` if you want to launch the first `'DRVR'` resource found in the file as returned by the Resource Manager. Because the `LaunchDeskAccessory` function opens the specified resource file for exclusive access, you cannot launch more than one desk accessory from the same resource file.

If the `'DRVR'` resource is in a resource file that is already open by the current process or if the driver is in the System file and the Option key is pressed, `LaunchDeskAccessory` launches the desk accessory in the application's heap. Otherwise, the desk accessory is given its own partition and launched in the system heap.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LaunchDeskAccessory` function are

Trap macro	Selector
`_OSDispatch`	$0036

RESULT CODES

`noErr`	0	No error
`resNotFound`	–192	Resource not found

Terminating Processes

You can use the `ExitToShell` procedure to have your application terminate itself directly. In general, you need to call `ExitToShell` only if you want your application to terminate without reaching the end of its main routine.

ExitToShell

Call `ExitToShell` to terminate your application directly.

```
PROCEDURE ExitToShell;
```

DESCRIPTION

The `ExitToShell` procedure terminates the calling process. The Process Manager removes your application from the list of open processes and performs any other necessary cleanup operations. In particular, all memory in your application partition and any temporary memory still allocated to your application is released. If necessary, the Application Died Apple event is sent to the process that launched your application.

If your application was the foreground process at the time it called `ExitToShell`, its name is removed from the Application menu. The Process Manager selects a new foreground process, switches it into the foreground, and propagates the scrap to the new foreground application.

If your application was the last one running and the shell program is not the Finder, the Process Manager displays a dialog box that gives the user the choice of restarting the computer or shutting it down.

SPECIAL CONSIDERATIONS

Any trap patches installed by your application are removed immediately by `ExitToShell`. They will not affect any trap calls made by `ExitToShell` itself.

RESULT CODES

When `ExitToShell` exits, the system global variable `DSErrCode` holds its result code.

SEE ALSO

See “Terminating an Application” on page 2-11 for details on the parameters passed to the Application Died event.

Summary of the Process Manager

Pascal Summary

Constants

```
CONST
      {Gestalt selector and response bits}
      gestaltOSAttr              = 'os  ';   {O/S attributes selector}
      gestaltLaunchCanReturn     = 1;        {can return from launch}
      gestaltLaunchFullFileSpec  = 2;        {LaunchApplication is available}
      gestaltLaunchControl       = 3;        {Process Manager is available}

      {process identification constants}
      kNoProcess                 = 0;        {process doesn’t exist}
      kSystemProcess             = 1;        {process belongs to OS}
      kCurrentProcess            = 2;        {the current process}

      {launch control flags}
      launchContinue             = $4000;    {continue after launch}
      launchNoFileFlags          = $0800;    {ignore launchFileFlags}
      launchUseMinimum           = $0400;    {use minimum or greater size}
      launchDontSwitch           = $0200;    {launch app. in background}
      launchAllow24Bit           = $0100;    {reserved}
      launchInhibitDaemon        = $0080;    {don't launch background app.}

      {launch parameter block length and ID}
      extendedBlockLen           = sizeof(LaunchParamBlockRec) - 12;
      extendedBlock              = $4C43;    {extended block}

      {flags in processMode field}
      modeDeskAccessory          = $00020000;   {process is desk acc}
      modeMultiLaunch            = $00010000;   {from app file's flags}
      modeNeedSuspendResume      = $00004000;   {from 'SIZE' resource}
      modeCanBackground          = $00001000;   {from 'SIZE' resource}
      modeDoesActivateOnFGSwitch = $00000800;   {from 'SIZE' resource}
      modeOnlyBackground         = $00000400;   {from 'SIZE' resource}
      modeGetFrontClicks         = $00000200;   {from 'SIZE' resource}
```

```
   modeGetAppDiedMsg          = $00000100;   {from 'SIZE' resource}
   mode32BitCompatible        = $00000080;   {from 'SIZE' resource}
   modeHighLevelEventAware    = $00000040;   {from 'SIZE' resource}
   modeLocalAndRemoteHLEvents = $00000020;   {from 'SIZE' resource}
   modeStationeryAware        = $00000010;   {from 'SIZE' resource}
   modeUseTextEditServices    = $00000008;   {from 'SIZE' resource}
```

Data Types

Process Serial Number

```
TYPE
   ProcessSerialNumber     =
   RECORD
      highLongOfPSN:       LongInt;             {high-order 32 bits of psn}
      lowLongOfPSN:        LongInt;             {low-order 32 bits of psn}
   END;

   ProcessSerialNumberPtr  = ^ProcessSerialNumber;
```

Process Information Record

```
   ProcessInfoRec          =
   RECORD
      processInfoLength:   LongInt;             {length of record}
      processName:         StringPtr;           {name of process}
      processNumber:       ProcessSerialNumber; {psn of the process}
      processType:         LongInt;             {file type of app file}
      processSignature:    OSType;              {signature of app file}
      processMode:         LongInt;             {'SIZE' resource flags}
      processLocation:     Ptr;                 {address of partition}
      processSize:         LongInt;             {partition size}
      processFreeMem:      LongInt;             {free bytes in heap}
      processLauncher:     ProcessSerialNumber; {proc that launched this one}
      processLaunchDate:   LongInt;             {time when launched}
      processActiveTime:   LongInt;             {accumulated CPU time}
      processAppSpec:      FSSpecPtr;           {location of the file}
   END;

   ProcessInfoRecPtr       = ^ProcessInfoRec;
```

Application Parameters Record

```
AppParameters           =
RECORD
   theMsgEvent:         EventRecord;         {event (high-level)}
   eventRefCon:         LongInt;             {reference constant}
   messageLength:       LongInt;             {length of buffer}
   messageBuffer:       ARRAY [0..0] OF SignedByte;
END;

AppParametersPtr        =   ^AppParameters;
```

Launch Parameter Block

```
LaunchFlags             =   Integer;

LaunchParamBlockRec     =
RECORD
   reserved1:           LongInt;             {reserved}
   reserved2:           Integer;             {reserved}
   launchBlockID:       Integer;             {extended block}
   launchEPBLength:     LongInt;             {length of block}
   launchFileFlags:     Integer;             {app's Finder flags}
   launchControlFlags:  LaunchFlags;         {launch options}
   launchAppSpec:       FSSpecPtr;           {location of app's file}
   launchProcessSN:     ProcessSerialNumber; {returned psn}
   launchPreferredSize: LongInt;             {returned pref size}
   launchMinimumSize:   LongInt;             {returned min size}
   launchAvailableSize: LongInt;             {returned avail size}
   launchAppParameters: AppParametersPtr;    {high-level event}
END;

LaunchPBPtr             =  ^LaunchParamBlockRec;
```

Routines

Getting Process Information

```
FUNCTION GetCurrentProcess  (VAR PSN: ProcessSerialNumber): OSErr;
FUNCTION GetNextProcess     (VAR PSN: ProcessSerialNumber): OSErr;
FUNCTION GetProcessInformation
                            (PSN: ProcessSerialNumber;
                             VAR info: ProcessInfoRec): OSErr;
```

```
FUNCTION SameProcess        (PSN1: ProcessSerialNumber;
                             PSN2: ProcessSerialNumber;
                             VAR result: Boolean): OSErr;
FUNCTION GetFrontProcess    (VAR PSN: ProcessSerialNumber): OSErr;
FUNCTION SetFrontProcess    (PSN: ProcessSerialNumber): OSErr;
FUNCTION WakeUpProcess      (PSN: ProcessSerialNumber): OSErr;
```

Launching Applications and Desk Accessories

```
FUNCTION LaunchApplication  (LaunchParams: LaunchPBPtr): OSErr;
FUNCTION LaunchDeskAccessory(pFileSpec: FSSpecPtr; pDAName: StringPtr):
                             OSErr;
```

Terminating a Process

```
PROCEDURE ExitToShell;
```

C Summary

Constants

```
/*Gestalt selector and response bits*/
#define gestaltOSAttr               'os  '   /*O/S attributes selector*/
#define gestaltLaunchCanReturn      1        /*can return from launch*/
#define gestaltLaunchFullFileSpec   2        /*LaunchApplication available*/
#define gestaltLaunchControl        3        /*Process Manager is available*/

/*process identification constants*/
enum {
      kNoProcess                    0,       /*process doesn't exist*/
      kSystemProcess                1,       /*process belongs to OS*/
      kCurrentProcess               2        /*the current process*/
};

/*launch control flags*/
enum {
      launchContinue            = 0x4000,   /*continue after launch*/
      launchNoFileFlags         = 0x0800,   /*ignore launchFileFlags*/
      launchUseMinimum          = 0x0400,   /*use minimum or greater size*/
      launchDontSwitch          = 0x0200,   /*launch app. in background*/
```

```
      launchAllow24Bit          = 0x0100,   /*reserved*/
      launchInhibitDaemon       = 0x0080    /*don't launch background app.*/
};

/*launch parameter block length and ID*/
#define extendedBlockLen           (sizeof(LaunchParamBlockRec) - 12)
#define extendedBlock              ((unsigned short)'LC')

/*flags in processMode field*/
enum {
      modeDeskAccessory         = 0x00020000,  /*process is desk acc*/
      modeMultiLaunch           = 0x00010000,  /*from app file's flags*/
      modeNeedSuspendResume     = 0x00004000,  /*from 'SIZE' resource*/
      modeCanBackground         = 0x00001000,  /*from 'SIZE' resource*/
      modeDoesActivateOnFGSwitch = 0x00000800,  /*from 'SIZE' resource*/
      modeOnlyBackground        = 0x00000400,  /*from 'SIZE' resource*/
      modeGetFrontClicks        = 0x00000200,  /*from 'SIZE' resource*/
      modeGetAppDiedMsg         = 0x00000100,  /*from 'SIZE' resource*/
      mode32BitCompatible       = 0x00000080,  /*from 'SIZE' resource*/
      modeHighLevelEventAware   = 0x00000040,  /*from 'SIZE' resource*/
      modeLocalAndRemoteHLEvents = 0x00000020,  /*from 'SIZE' resource*/
      modeStationeryAware       = 0x00000010,  /*from 'SIZE' resource*/
      modeUseTextEditServices   = 0x00000008   /*from 'SIZE' resource*/
};
```

Data Types

Process Serial Number

```
struct ProcessSerialNumber {
      unsigned long        highLongOfPSN;       /*high-order 32 bits of psn*/
      unsigned long        lowLongOfPSN;        /*low-order 32 bits of psn*/
};

typedef struct ProcessSerialNumber ProcessSerialNumber;
typedef ProcessSerialNumber *ProcessSerialNumberPtr;
```

Process Information Record

```
struct ProcessInfoRec {
      unsigned long        processInfoLength;   /*length of record*/
      StringPtr            processName;         /*name of process*/
      ProcessSerialNumber  processNumber;       /*psn of the process*/
```

```
      unsigned long        processType;         /*file type of app file*/
      OSType               processSignature;    /*signature of app file*/
      unsigned long        processMode;         /*'SIZE' resource flags*/
      Ptr                  processLocation;     /*address of partition*/
      unsigned long        processSize;         /*partition size*/
      unsigned long        processFreeMem;      /*free bytes in heap*/
      ProcessSerialNumber  processLauncher;     /*proc that launched this */
                                                /* one*/
      unsigned long        processLaunchDate;   /*time when launched*/
      unsigned long        processActiveTime;   /*accumulated CPU time*/
      FSSpecPtr            processAppSpec;      /*location of the file*/
};

typedef struct ProcessInfoRec ProcessInfoRec;
typedef ProcessInfoRec *ProcessInfoRecPtr;
```

Application Parameters Record

```
struct AppParameters {
      EventRecord          theMsgEvent;         /*event (high-level)*/
      unsigned long        eventRefCon;         /*reference constant*/
      unsigned long        messageLength;       /*length of buffer*/
};

typedef struct AppParameters AppParameters;
typedef AppParameters *AppParametersPtr;
```

Launch Parameter Block

```
typedef unsigned short LaunchFlags;

struct LaunchParamBlockRec {
      unsigned long        reserved1;           /*reserved*/
      unsigned short       reserved2;           /*reserved*/
      unsigned short       launchBlockID;       /*extended block*/
      unsigned long        launchEPBLength;     /*length of block*/
      unsigned short       launchFileFlags;     /*app's Finder flags*/
      LaunchFlags          launchControlFlags;  /*launch options*/
      FSSpecPtr            launchAppSpec;       /*location of app's file*/
      ProcessSerialNumber  launchProcessSN;     /*returned psn*/
      unsigned long        launchPreferredSize; /*returned pref size*/
      unsigned long        launchMinimumSize;   /*returned min size*/
      unsigned long        launchAvailableSize; /*returned avail size*/
      AppParametersPtr     launchAppParameters; /*high-level event*/
```

```
};

typedef struct LaunchParamBlockRec LaunchParamBlockRec;
typedef LaunchParamBlockRec *LaunchPBPtr;
```

Routines

Getting Process Information

```
pascal OSErr GetCurrentProcess
                            (ProcessSerialNumber *PSN);
pascal OSErr GetNextProcess  (ProcessSerialNumber *PSN);
pascal OSErr GetProcessInformation
                            (const ProcessSerialNumber *PSN,
                             ProcessInfoRecPtr info);
pascal OSErr SameProcess    (const ProcessSerialNumber *PSN1,
                             const ProcessSerialNumber *PSN2,
                             Boolean *result);
pascal OSErr GetFrontProcess
                            (ProcessSerialNumber *PSN);
pascal OSErr SetFrontProcess
                            (const ProcessSerialNumber *PSN);
pascal OSErr WakeUpProcess  (const ProcessSerialNumber *PSN);
```

Launching Applications and Desk Accessories

```
pascal OSErr LaunchApplication
                            (const LaunchParamBlockRec *LaunchParams);
pascal OSErr LaunchDeskAccessory
                            (const FSSpec *pFileSpec,
                             ConstStr255Param pDAName);
```

Terminating a Process

```
pascal void ExitToShell     (void);
```

Assembly-Language Summary

Data Structures

Process Serial Number

0	highLongOfPSN	long	high-order 32-bits of process serial number
4	lowLongOfPSN	long	low-order 32-bits of process serial number

Process Information Record

0	processInfoLength	long	length of this record
4	processName	long	name of process
8	processNumber	2 longs	process serial number of the process
16	processType	long	type of application file
20	processSignature	long	signature of application file
24	processMode	long	flags from 'SIZE' resource
28	processLocation	long	address of process partition
32	processSize	long	partition size (in bytes)
36	processFreeMem	long	amount of free memory in application heap
40	processLauncher	2 longs	process that launched this one
48	processLaunchDate	long	value of Ticks at time of launch
52	processActiveTime	long	total time spent using the CPU
56	processAppSpec	long	location of the file

Application Parameters Record

0	theMsgEvent	16 bytes	the high-level event record
16	eventRefCon	long	reference constant
20	messageLength	long	length of buffer
24	messageBuffer	byte	first byte of the message buffer

Launch Parameter Block

0	reserved1	long	reserved
4	reserved2	word	reserved
6	launchBlockID	word	specifies whether block is extended
8	launchEPBLength	long	length (in bytes) of rest of parameter block
12	launchFileFlags	word	the Finder flags for the application file
14	launchControlFlags	word	flags that specify launch options
16	launchAppSpec	long	address of FSSpec that specifies the application file to launch
20	launchProcessSN	2 longs	process serial number
28	launchPreferredSize	long	application's preferred partition size
32	launchMinimumSize	long	application's minimum partition size
36	launchAvailableSize	long	maximum partition size available
40	launchAppParameters	long	high-level event for launched application

Trap Macros

Trap Macro Names

Pascal name	Trap macro name
`LaunchApplication`	`_Launch`
`ExitToShell`	`_ExitToShell`

Trap Macros Requiring Routine Selectors

`_OSDispatch`

Selector	Routine
$0036	`LaunchDeskAccessory`
$0037	`GetCurrentProcess`
$0038	`GetNextProcess`
$0039	`GetFrontProcess`
$003A	`GetProcessInformation`
$003B	`SetFrontProcess`
$003C	`WakeUpProcess`
$003D	`SameProcess`

Result Codes

`noErr`	0	No error
`paramErr`	–50	Process serial number is invalid
`memFullErr`	–108	Not enough memory to allocate the partition size specified in the `'SIZE'` resource
`resNotFound`	–192	Resource not found
`procNotFound`	–600	No eligible process with specified process serial number
`memFragErr`	–601	Not enough room to launch application with special requirements
`appModeErr`	–602	Addressing mode is 32-bit, but application is not 32-bit clean
`appMemFullErr`	–605	Partition size specified in `'SIZE'` resource is not big enough for launch
`appIsDaemon`	–606	Application is background-only

CHAPTER 3

Time Manager

Contents

This chapter describes how you can use the Time Manager to schedule execution of a routine after a specified amount of time has elapsed. It includes information about the original Time Manager, as well as information about the revised Time Manager introduced in system software version 6.0.3 and the extended Time Manager introduced in system software version 7.0.

Because different versions of the Time Manager are available under different system software versions, your application may need to determine which version is available in its current environment. To do so, use the `Gestalt` function explained in the chapter "Gestalt Manager" in *Inside Macintosh: Operating System Utilities*.

To use this chapter, you should be familiar with the Vertical Retrace Manager because it provides an alternative (and sometimes preferable) method for scheduling routines for future or periodic execution. For details on the Vertical Retrace Manager, see the chapter "Vertical Retrace Manager" in this book.

About the Time Manager

The Time Manager allows applications and other software to schedule routines for execution at a later time. By suitably defining the routine that is to be executed later, you can use the Time Manager to accomplish a wide range of time-related activities. For example, because a routine can reschedule itself for later execution, the Time Manager allows your application to perform periodic or repeated actions. You can use the Time Manager to

- schedule routines for execution after a specified delay
- set up tasks that run periodically
- compute the time a routine takes to run
- coordinate and synchronize actions in the Macintosh computer

The Time Manager provides a hardware-independent method of performing these time-related tasks. In general, you should use the Time Manager instead of timing loops, which can vary in duration because they depend on clock speed and interrupt-handling speed.

To use the Time Manager, you must first issue a request by passing the Time Manager the address of a task record, one of whose fields contains the address of the routine that is to run. Then you need to activate that request by specifying the delay until the routine is to run. The Time Manager uses a **Time Manager queue** to maintain requests that you issue. The structure of this queue is similar to that of standard operating-system queues. The Time Manager queue can hold any number of outstanding requests, and each application can add any number of entries to the queue. If there are several requests scheduled for execution at exactly the same time, the Time Manager schedules them for execution as close to the specified time as possible, in the order in which they entered the Time Manager queue.

The routine you place in the queue can perform any desired action so long as it does not call the Memory Manager, either directly or indirectly. (You cannot call the Memory Manager because Time Manager tasks are executed at interrupt time.)

The Time Manager introduced in system software version 7.0 is the third version released. The three versions are known as the original Time Manager, the revised Time Manager, and the extended Time Manager. The three versions are all upwardly compatible—that is, each succeeding Time Manager version is a functional superset of the previous one. However, code written for the extended Time Manager may not run properly with either the original or revised version. For this reason, it is sometimes important to know which Time Manager version is available on a specific computer.

You can use the `Gestalt` function to determine which version of the Time Manager is present. You should pass `Gestalt` the selector `gestaltTimeMgrVersion`.

```
CONST
   gestaltTimeMgrVersion      = 'tmgr';   {Time Manager version}
```

If `Gestalt` executes successfully, it returns one of three constants:

```
CONST
   gestaltStandardTimeMgr     = 1;        {original Time Manager}
   gestaltRevisedTimeMgr      = 2;        {revised Time Manager}
   gestaltExtendedTimeMgr     = 3;        {extended Time Manager}
```

If `Gestalt` returns an error, you should assume that the original Time Manager is present. The following sections describe the features of each version of the Time Manager.

The Original Time Manager

The Time Manager was first introduced with the Macintosh Plus ROMs (which are also used in Macintosh 512K enhanced models) and was intended for use internally by the Operating System. The original Time Manager allows delays as small as 1 millisecond, resulting in a maximum range of about 24 days.

To schedule a task for later execution, place an entry into the Time Manager queue and then activate it. All Time Manager routines manipulate elements of the Time Manager queue, which are stored in a **Time Manager task record.** The task record for the original Time Manager is defined by the `TMTask` data type.

```
TYPE TMTask =     {original and revised Time Manager task record}
   RECORD
      qLink:      QElemPtr;      {next queue entry}
      qType:      Integer;       {queue type}
      tmAddr:     ProcPtr;       {pointer to task}
      tmCount:    LongInt;       {reserved}
   END;
```

Of the four fields in this record, you need to fill in only the `tmAddr` field, which contains a pointer to the routine that is to be executed at some time in the future. The remaining fields are used internally by the Time Manager or are reserved by Apple Computer, Inc. However, you should set the `tmCount` field to 0 when you set up a task record.

The original Time Manager includes three routines:

- The `InsTime` procedure installs a task record into the Time Manager queue.
- The `PrimeTime` procedure schedules a previously queued task record for future execution.
- The `RmvTime` procedure removes a task record from the Time Manager queue.

Note that installing a request into the Time Manager queue (by calling the `InsTime` procedure) does not by itself schedule the specified routine for future execution. After you queue a request, you still need to activate (or **prime**) the request by specifying the desired delay until execution (by calling the `PrimeTime` procedure). Note also that the task record is not automatically removed from the Time Manager queue after the routine is executed. For this reason, you can reactivate the task by subsequent calls to `PrimeTime`; you do not have to reinstall the task record.

To remove a task record from the queue, you must call the `RmvTime` procedure. The `RmvTime` procedure removes a task record from the Time Manager queue whether or not that task was ever activated and whether or not its specified time delay has expired.

The Revised Time Manager

System software version 6.0.3 introduced a revised version of the Time Manager. This version provides better time resolution and more accurate measurements of elapsed time. You can represent time delays in the revised Time Manager as microseconds (µsec) as well as milliseconds (msec), with a finest resolution of 20 microseconds. The external programming interface did not change from the original to the revised Time Manager, although the revised version provides a means to distinguish microsecond delays from millisecond delays.

The revised Time Manager interprets negative time values (which were not formerly allowed) as negated microseconds. For example, a value of –50 is interpreted as a delay of 50 microseconds. Positive time values continue to represent milliseconds. When specified as microseconds, the maximum delay is about 35 minutes. When specified as milliseconds, the maximum delay is about 1 day. (This differs from the maximum delay in the original Time Manager because of the finer resolution of the revised Time Manager.) When passed to `PrimeTime`, the time value is converted to an internal form. For this reason, it makes no difference which unit you use if the delay falls within the ranges of both.

The revised Time Manager provides additional features. The principal change concerns the `tmCount` field of the Time Manager task record (previously reserved for use by Apple Computer, Inc.). When you remove an active task from the revised Time Manager's queue, any time remaining until the scheduled execution time is returned in the `tmCount` field. This change allows you to use the Time Manager to compute elapsed

times (as explained in the section “Computing Elapsed Time” on page 3-14). In addition, the high-order bit of the `qType` field of the task record is used as a flag to indicate whether the task timer is active. The `InsTime` procedure initially clears this bit, and `PrimeTime` sets it. This bit is cleared when the time expires or when your application calls `RmvTime`.

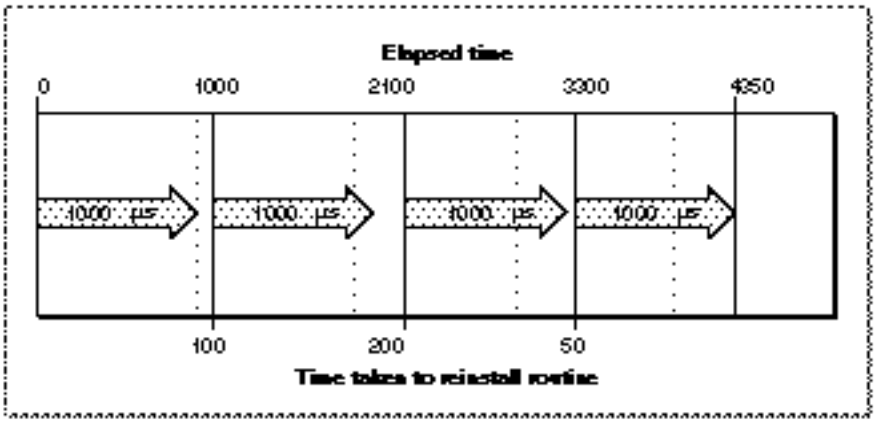
Although the revised Time Manager supports the specification of delay times in microseconds, you should use this feature primarily for the more accurate measurement of elapsed times. You should avoid specifying very small delay times as a way to execute a routine repeatedly at frequent intervals because this technique may use a considerable amount of processor time. The amount of processor time consumed by such timing services varies, depending largely on the performance of the CPU. With low-performance CPUs, little or no time may be left for other processing on the system (for instance, moving the mouse or running the application).

The Extended Time Manager

The extended Time Manager (available with system software version 7.0 and later) contains all the features of earlier Time Managers, with several extensions intended primarily to provide **drift-free, fixed-frequency** timing services. These services, which ensure that a routine is executed promptly after a specified delay, are important for sound and multimedia applications requiring precise timing and real-time synchronization among different events.

In the original and revised Time Managers, the value passed to `PrimeTime` indicates a delay that is relative to the current time (that is, the time when you execute `PrimeTime`). This presents problems if you attempt to implement a fixed-frequency timing service by having the task call `PrimeTime`. The problem is that the time consumed by the Time Manager and by any interrupt latency (which is not predictable) causes the task to be called at a slightly slower and unpredictable frequency, which drifts over time. In Figure 3-1, the desired fixed frequency of 1000 microseconds cannot be achieved because the Time Manager overhead and interrupt latency cause a small and unpredictable delay each time the task is reactivated.

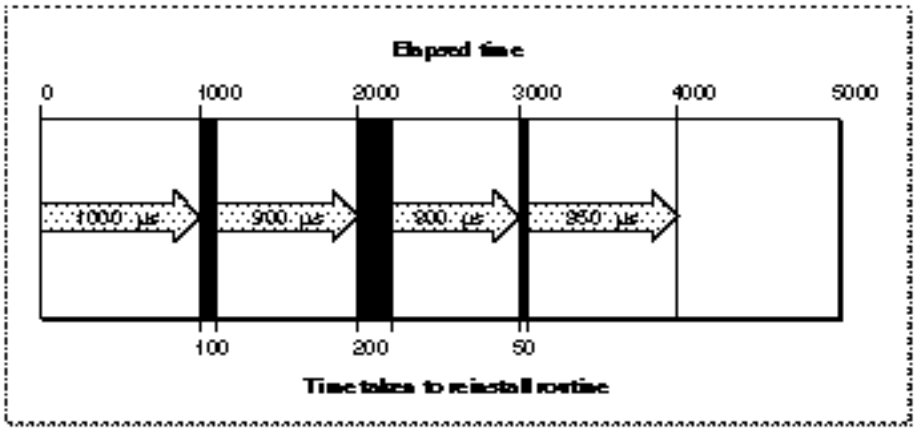
Figure 3-1 Original and revised Time Managers (drifting, unpredictable frequency)



The extended Time Manager solves this problem by allowing you to reinstall a task with an execution time that is relative to the time when the task last expired—not relative to the time when the task is reinstalled. The extended Time Manager compensates for the delay between the time when the task last expired and the time at which it was reinstalled, thereby providing a truly drift-free, fixed-frequency timing service.

For example, if your application needs to execute a routine periodically at 1-millisecond intervals, it can reactivate the existing Time Manager queue element by calling `PrimeTime` in the task with a specified delay of 1 millisecond. When the Time Manager receives this new execution request, it determines how long ago the previous `PrimeTime` task expired and then decrements the specified delay by that amount. For instance, if the previous task expired 100 microseconds ago, then the Time Manager installs the new task with a delay of 900 microseconds. This technique is illustrated in Figure 3-2.

Figure 3-2 The extended Time Manager (drift-free, fixed frequency)



The extended Time Manager implements these features by recognizing an expanded task record and providing a new procedure, `InsXTime`. The Time Manager task record for the extended Time Manager looks like this:

```
TYPE TMTask =        {extended Time Manager task record}
   RECORD
      qLink:         QElemPtr;   {next queue entry}
      qType:         Integer;    {queue type}
      tmAddr:        ProcPtr;    {pointer to task}
      tmCount:       LongInt;    {unused time}
      tmWakeUp:      LongInt;    {wakeup time}
      tmReserved:    LongInt;    {reserved for future use}
   END;
```

Once again, your application fills in the `tmAddr` field. You should set `tmWakeUp` and `tmReserved` to 0 when you first install an extended Time Manager task. The remaining fields are used internally by the Time Manager. As in the revised Time Manager, the `tmCount` field holds the time remaining until the scheduled execution of the task (this field is set by `RmvTime`).

The `tmWakeUp` field contains the time at which the Time Manager task specified by `tmAddr` was last executed (or 0 if it has not yet been executed). Its principal intended use is to provide drift-free, fixed-frequency timing services, which are available only when you use the extended Time Manager and only when you install Time Manager tasks by calling the new `InsXTime` procedure.

When your application installs an extended Time Manager task (by calling the `InsXTime` procedure), the behavior of the `PrimeTime` procedure changes slightly, as described earlier in this section. If the value of the `tmWakeUp` field is zero when `PrimeTime` is called, the delay parameter to `PrimeTime` is interpreted as relative to the current time (just as in the original Time Manager), but the Time Manager sets the `tmWakeUp` field to a nonzero value that indicates when the delay time should expire. When your application calls `PrimeTime` with a Time Manager task whose `tmWakeUp` field contains a nonzero value, the Time Manager interprets the specified delay as relative to the time that the last call to `PrimeTime` on this task was supposed to expire.

Note

Nonzero values in `tmWakeUp` are in a format that is used internally by the Time Manager. This format is subject to change. Your application should never use the value stored in this field and should either set it to 0 or leave it unchanged. When you first create an extended Time Manager task record, make sure that the value of the `tmWakeUp` field is 0; otherwise, the Time Manager may interpret it as a prior execution time. u

The extended Time Manager allows for a previously impossible situation that may lead to undesirable results. It is possible to call `PrimeTime` with an execution time that is in the past instead of in the future. (In the original and revised Time Managers, only future execution times are possible.) This situation arises when the value of the `tmWakeUp` field

specifies a time in the past and you issue a new `PrimeTime` request with a delay value that is not large enough to cause the execution time to be in the future. This may occur when fixed, high-frequency execution is required and the time needed to process each execution, including the Time Manager overhead, is greater than the delay time between requests.

When your application issues a `PrimeTime` request with a `tmWakeUp` value that would result in a negative delay, the actual delay time is set to 0. The Time Manager updates the `tmWakeUp` field to indicate the time when the task should have been performed (in the past). Because the actual delay time is set to 0, the task is executed immediately. If your application continually issues `PrimeTime` requests for times in the past, the Time Manager and the `tmAddr` tasks consume all of the processor cycles. As a result, no time is left for the application to run. Because this situation is a function of processor speed, you should ensure compatibility by using the slowest processors to test applications that use extended Time Manager features. Another solution to this problem is to vary the wakeup frequency according to the processing power of the computer.

Using the Time Manager

The Time Manager is automatically initialized when the system starts up. At that time, the queue of Time Manager task records is empty. The Operating System, applications, and other software components may place records into the queue. Because the delay time for a given task can be as small as 20 microseconds, you need to install an element into the Time Manager queue before actually issuing a request to execute it at some future time. You place elements into the queue by calling the `InsTime` procedure or (if you need the fixed-frequency services of the extended Time Manager) the `InsXTime` procedure. To activate the request, call `PrimeTime`. The Time Manager then marks the specified task record as active by setting the high-order bit in the `qType` field of that record.

The `tmAddr` field of the Time Manager task record contains the address of a task. The Time Manager calls this task when the time delay specified by a previous call to `PrimeTime` has elapsed. The task can perform any desired actions, as long as it does not call the Memory Manager (either directly or indirectly) and does not depend on the validity of handles to unlocked blocks.

Note

If the routine specified in the Time Manager task record is located in your application's heap, then your application must still be active when the specified delay elapses, or the application should call `RmvTime` before it terminates. Otherwise, the Time Manager does not know that the address of that routine is not valid when the routine is called. The Time Manager then attempts to call the task, but with a stale pointer. If you want to let the application terminate after it has installed and activated a Time Manager task record, load the routine into the system heap. u

There are two ways for an active queue element to become inactive. First, the specified time delay can elapse, in which case the routine pointed to by the `tmAddr` field is called. Second, your application can call the `RmvTime` procedure, in which case the amount of time remaining before the delay would have elapsed (the unused time) is reported in the `tmCount` field of the task record. This feature allows you to use the Time Manager to compute elapsed times (see the section "Computing Elapsed Time" on page 3-14), which is useful for obtaining performance measurements. Calling `RmvTime` removes an element from the queue whether or not that task is active when `RmvTime` is called.

To use the Time Manager for periodic execution of a task, simply have the routine pointed to by `tmAddr` call `PrimeTime` again. This technique is illustrated in the section "Performing Periodic Tasks" on page 3-13. Similarly, you can execute a Time Manager task a specific number of times by keeping a count of the number of times the task has been called. In cases where the task needs access to your application's global variables (such as a count variable), make sure that the A5 register points to your application's global variables when the task is executed and that A5 is restored to its original value when your task exits. A technique for this purpose is illustrated in "Using Application Global Variables in Tasks" on page 3-11.

Installing and Activating Tasks

Listing 3-1 shows how to install and activate a Time Manager task. It assumes that the procedure`MyTask` has already been defined; see Listing 3-3 and Listing 3-4 for examples of simple task definitions.

Listing 3-1 Installing and activating a Time Manager task

```
PROCEDURE InstallTMTask;
CONST
   kDelay = 2000;                  {delay value}
BEGIN
   gTMTask.tmAddr := @MyTask;      {get address of task}
   gTMTask.tmWakeUp := 0;          {initialize tmWakeUp}
   gTMTask.tmReserved := 0;        {initialize tmReserved}
   InsXTime(@gTMTask);             {install the task record}
   PrimeTime(@gTMTask, kDelay);    {activate the task record}
END;
```

In this example, `InstallTMTask` installs an extended Time Manager task record into the Time Manager queue and then activates the task. (The extended Time Manager task record, `gTMTask`, is a global variable of type `TMTask`.) After the specified delay has elapsed (in this case, 2000 milliseconds, or 2 seconds), the procedure `MyTask` runs.

In cases where no task is to run after the specified delay has elapsed, you should set the `tmAddr` field to `NIL`. To determine if the time has expired, you can check the task-active bit in the `qType` field.

Avoid calling `PrimeTime` with a Time Manager task record that has not yet expired, because the results are unpredictable. If you wish to reactivate a prior unexpired request in the Time Manager queue and specify a different delay, call `RmvTime` to cancel the prior request, then call `InsTime` to reinstall the timer task, and finally call `PrimeTime` to reschedule the task. Note, however, that it is possible and sometimes desirable to call `PrimeTime` with a Time Manager task that you want to reactivate, because the timer will have expired before the task is called.

Using Application Global Variables in Tasks

When a Time Manager task executes, the A5 world of the application that installed the corresponding task record into the Time Manager queue might not be valid (for example, the task might execute at interrupt time when that application is not the current application). If so, an attempt to read the application's global variables returns erroneous results because the A5 register points to the application global variables of some other application. When a Time Manager task uses an application's global variables, you must ensure that register A5 contains the address of the boundary between the application global variables and the application parameters of the application that launched it. You must also restore register A5 to its original value before the task exits.

It is relatively straightforward to read the current value of the A5 register when a Time Manager task begins to execute (using the `SetCurrentA5` function) and to restore it before exiting (using the `SetA5` function). It is more complicated, however, to pass to a Time Manager task the value to which it should set A5 before accessing its application's global variables. The problem is that neither the original nor the extended Time Manager task record contains an unused field in which your application could pass this information to the task. The situation here is unlike the situation with Notification Manager tasks or Sound Manager callback routines (both of which provide an easy way to pass the address of the application's A5 world to the task), but it is similar to the situation with vertical retrace tasks.

Note
For a more detailed discussion of setting and restoring your application's A5 world, see the chapter "Memory Management Utilities" in *Inside Macintosh: Memory.* u

One way to gain access to the global variables of the application that launched a Time Manager task is to pass to `InsTime` (or `InsXTime`) and `PrimeTime` the address of a structure, the first segment of which is simply the corresponding Time Manager task record and the remaining segment of which contains the address of the application's A5 world. For example, you can define a new data structure, a Time Manager information record, as follows:

```
TYPE TMInfo =              {Time Manager information record}
   RECORD
      myTMTask:   TMTask;  {original and revised TM task record}
      tmRefCon:   LongInt; {space to pass address of A5 world}
   END;

TMInfoPtr = ^TMInfo;
```

Note
The `TMInfo` record defined above is intended for use with the extended Time Manager. u

Then you can install and activate your Time Manager task as illustrated in Listing 3-2. The global variable `gTMInfo` is an information record of type `TMInfo`.

Listing 3-2 Passing the address of the application's A5 world to a Time Manager task

```
PROCEDURE InstallTMTask;
CONST
   kDelay = 2000;                      {delay value}
BEGIN
   gTMInfo.myTMTask.tmAddr := @MyTask; {get address of task}
   gTMInfo.myTMTask.tmWakeUp := 0;     {initialize tmWakeUp}
   gTMInfo.myTMTask.tmReserved := 0;   {initialize tmReserved}
   gTMInfo.tmRefCon := SetCurrentA5;   {store address of A5 }
                                       { world}
   InsTime(@gTMInfo);                  {install the info record}
   PrimeTime(@gTMInfo, kDelay);        {activate the info record}
END;
```

With the revised and extended Time Managers, the task is called with register A1 containing the address passed to `InsTime` (or `InsXTime`) and `PrimeTime`. Thus, the Time Manager task simply needs to retrieve the `TMInfo` record and extract the appropriate value of the application's A5 world. Listing 3-3 illustrates a task definition for this purpose.

Listing 3-3 Defining a Time Manager task that can manipulate global variables

```
FUNCTION GetTMInfo: TMInfoPtr;
   INLINE $2E89;                          {MOVE.L A1,(SP)}

PROCEDURE MyTask;
VAR
   oldA5:   LongInt;                      {A5 when task is called}
   recPtr:  TMInfoPtr;
BEGIN
   recPtr := GetTMInfo;                   {first get your record}
   oldA5 := SetA5(recPtr^.tmRefCon);      {set A5 to app's A5 world}

   {Do something with the application's globals here.}

   oldA5 := SetA5(oldA5);                 {restore original A5 }
                                          { and ignore result}
END;
```

This technique works primarily because the revised and extended Time Managers do not care if the record whose address is passed to `InsTime` (or `InsXTime`) and `PrimeTime` is larger than expected. If you use this technique, however, be sure to retrieve the address of the task record from register A1 as soon as you enter the Time Manager task (because some compilers generate code that uses registers A0 and A1 to dereference structures).

IMPORTANT

You cannot use the technique illustrated in Listing 3-3 with the original Time Manager because it does not pass the address of the task record in register A1. To gain access to your application's global variables when using the original Time Manager, you would need to store your application's A5 value in one of the application's code segments (in particular, in the code segment that contains the Time Manager task). This technique involves the use of self-modifying code segments and is not in general recommended. Applications that attempt to modify their own `'CODE'` resources may crash in operating environments (for example, A/UX) that restrict an application's access to its own code segments. s

Performing Periodic Tasks

One way to install a periodic Time Manager task is to have the task reactivate itself. Because the task record is already inserted into the Time Manager task queue, the task can simply call `PrimeTime` to reactivate itself. To call `PrimeTime`, however, the task needs to know the address of the corresponding task record. In the revised and extended Time Managers, the task record's address is placed into register A1 when the task is

called. Listing 3-4 illustrates how the task can reactivate itself by retrieving the address in register A1 and passing that address to `PrimeTime`.

Listing 3-4 Defining a periodic Time Manager task

```
FUNCTION GetTMInfo: TMInfoPtr;
   INLINE $2E89;                {MOVE.L A1,(SP)}

PROCEDURE MyTask;               {for revised and extended TMs}
VAR
   recPtr:        TMInfoPtr;
CONST
   kDelay = 2000;               {delay value}
BEGIN
   recPtr := GetTMInfo;         {first get your own address}

   {Do something here.}

   PrimeTime(QElemPtr(recPtr), kDelay);
END;
```

IMPORTANT

You cannot use the technique illustrated in Listing 3-4 with the original Time Manager because it does not pass the address of the task record in register A1. s

Computing Elapsed Time

In the revised and extended Time Managers, the `RmvTime` procedure returns, in the `tmCount` field of the task record, a value representing any unused time. This feature makes the Time Manager extremely useful for computing elapsed times.

To compute the amount of time that a routine takes to run, call `PrimeTime` at the beginning of the interval to be measured and specify a delay greater than the expected elapsed time. Then call `RmvTime` at the end of the interval and subtract the unused time returned in `tmCount` from the original delay passed to `PrimeTime`.

To obtain the most accurate results, you should calculate all times in microseconds (in which case the `tmCount` field of the task record has a range of about 35 minutes). To get an exact measurement, compute the overhead associated with calling the Time Manager and subtract it from the preliminary result. Listing 3-5 illustrates a technique for calculating that overhead.

Listing 3-5 Calculating the time required to install and activate a Time Manager task

```
FUNCTION TMOverhead: LongInt;
VAR
   myTask:     TMTask;     {a Time Manager task record}
   myStart:    LongInt;    {initial delay passed to PrimeTime}
   myElapsed:  LongInt;    {elapsed time}
BEGIN
   myStart := -(MAXLONG);  {use a large negative number}

   WITH myTask DO          {set up the task record}
      BEGIN
         tmAddr := NIL;    {no task to execute}
         tmWakeUp := 0;
         tmReserved := 0;
      END;

   InsTime(@myTask);             {install the task}
   PrimeTime(@myTask, myStart);  {prime the task}
   RmvTime(@myTask);             {remove the task}

   myElapsed := myStart - myTask.tmCount;
   TMOverhead := -(myElapsed);   {the elapsed time}
END;
```

The `TMOverhead` function defined in Listing 3-5 sets up a Time Manager task record with no completion routine. In this case, you can allocate the task record as a local variable on the stack because the task record is removed before the function exits. Then the task is activated by calling `PrimeTime` with a very large negative value. (The negative value represents microseconds.) Immediately the task is deactivated and removed. The function determines the elapsed time by subtracting the value returned in the `tmCount` field of the task record from the original delay time.

Listing 3-6 illustrates how to measure the elapsed time associated with a request to delay program execution by 1 tick.

Listing 3-6 Calculating the time consumed by a 1-tick delay

```
FUNCTION CheckDelayTiming: LongInt;
VAR
   myTask:     TMTask;     {a Time Manager task record}
   myStart:    LongInt;    {initial delay passed to PrimeTime}
   myEnd:      LongInt;    {unused time}
   myTicks:    LongInt;    {ignored; needed for Delay procedure}
   myElapsed:  LongInt;    {elapsed time}
BEGIN
   myStart := -(MAXLONG);  {use a large negative number}

   WITH myTask DO          {set up the task record}
      BEGIN
         tmAddr := NIL;    {no task to execute}
         tmWakeUp := 0;
         tmReserved := 0;
      END;

   InsTime(@myTask);             {install the task}
   PrimeTime(@myTask, myStart);  {prime the task}
   Delay(1, myTicks);            {delay for 1 tick}
   RmvTime(@myTask);             {remove the task}

   myEnd := myTask.tmCount;      {get unused part of myStart}

   IF myEnd < 0 THEN             {myEnd is in microseconds}
      myElapsed := ABS(myStart - myEnd) - TMOverhead
   ELSE                          {myEnd is in milliseconds}
      myElapsed := ABS(myStart + (myEnd * 1000)) - TMOverhead;

   CheckDelayTiming := myElapsed;{the elapsed time}
END;
```

The `CheckDelayTiming` function is similar to the `TMOverhead` function except that the section of code to be timed occurs between the calls to `PrimeTime` and `RmvTime`. The `CheckDelayTiming` function simply times a call to the `Delay` procedure with a 1-tick delay time. Once `Delay` has completed and the task record has been deactivated, `CheckDelayTiming` determines whether the unused time returned in the `tmCount` field represents microseconds or milliseconds. The value returned by `CheckDelayTiming` is in microseconds.

Time Manager Reference

This section describes the data structures and routines that are specific to the Time Manager. It also describes the application-defined Time Manager task procedure whose address is specified in the task record.

Data Structures

All Time Manager routines require that you pass the address of a Time Manager task record, defined by the `TMTask` data type. If you are using the original or revised Time Manager, the task record has this structure:

```
TYPE TMTask =     {original and revised Time Manager task record}
   RECORD
      qLink:      QElemPtr;      {next queue entry}
      qType:      Integer;       {queue type}
      tmAddr:     ProcPtr;       {pointer to task}
      tmCount:    LongInt;       {reserved}
   END;
```

Field descriptions

`qLink` A pointer to the next element in the Time Manager queue. This field is used internally by the Time Manager.

`qType` The type of queue. The Time Manager automatically sets this field to the appropriate value. In the revised Time Manager, the high-order bit of this field is a flag that indicates whether the task is active.

`tmAddr` A pointer to the routine to be executed after the delay specified in a call to `PrimeTime`.

`tmCount` Reserved in the original Time Manager. In the revised Time Manager, the amount of time remaining until the task's scheduled execution time; this field is valid only after you call `RmvTime` with a task that has not yet executed.

If you are using the extended Time Manager, the task record has this structure:

```
TYPE TMTask =        {extended Time Manager task record}
   RECORD
      qLink:         QElemPtr;   {next queue entry}
      qType:         Integer;    {queue type}
      tmAddr:        ProcPtr;    {pointer to task}
      tmCount:       LongInt;    {unused time}
```

```
      tmWakeUp:      LongInt;    {wakeup time}
      tmReserved:    LongInt;    {reserved for future use}
   END;
```

Field descriptions

`qLink`	A pointer to the next element in the Time Manager queue. This field is used internally by the Time Manager.
`qType`	The type of queue. The Time Manager automatically sets this field to the appropriate value. The high-order bit of this field is a flag that indicates whether the task is active.
`tmAddr`	A pointer to the routine that is to be executed after the delay specified in a call to `PrimeTime`.
`tmCount`	The time remaining until the task's scheduled execution time. This field is valid only after you call `RmvTime` with a task that has not yet executed.
`tmWakeUp`	The time when the task specified in the `tmAddr` field was last executed. This field is used internally by the Time Manager. You should set it to 0 when you first install a task record.
`tmReserved`	Reserved.

Time Manager Routines

You can insert a task record into the Time Manager's queue by calling `InsTime` or `InsXTime`. Use `InsXTime` only if you wish to use the drift-free, fixed-frequency timing services of the extended Time Manager; use `InsTime` in all other cases. After you have queued a task record, you can activate it by calling `PrimeTime`. You can remove a task record from the queue by calling `RmvTime`.

InsTime

You can install a task record into the Time Manager task queue using the `InsTime` procedure.

```
PROCEDURE InsTime (tmTaskPtr: QElemPtr);
```

`tmTaskPtr` A pointer to an original task record to be installed in the queue.

DESCRIPTION

The `InsTime` procedure adds the Time Manager task record specified by `tmTaskPtr` to the Time Manager queue. Your application should fill in the `tmAddr` field of the task record and should set the `tmCount` field to 0. The `tmTaskPtr` parameter must point to an original Time Manager task record.

With the revised and extended Time Managers, you can set `tmAddr` to `NIL` if you do not want a task to execute when the delay passed to `PrimeTime` expires. Also, the revised Time Manager resets the high-order bit of the `qType` field to 0 when you call `InsTime`.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `InsTime` are

Registers on entry

A0 Address of the task record

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error

InsXTime

Use the `InsXTime` procedure to install a task if you want to take advantage of the drift-free, fixed-frequency timing services of the extended Time Manager.

```
PROCEDURE InsXTime (tmTaskPtr: QElemPtr);
```

`tmTaskPtr` A pointer to an extended task record to be installed in the queue.

DESCRIPTION

The `InsXTime` procedure adds the Time Manager task record specified by `tmTaskPtr` to the Time Manager queue. The `tmTaskPtr` parameter must point to an extended Time Manager task record. Your application must fill in the `tmAddr` field of that task. You should set the `tmWakeUp` and `tmReserved` fields to 0 the first time you call `InsXTime`.

With the extended Time Manager, you can set `tmAddr` to `NIL` if you do not want a task to execute when the delay passed to `PrimeTime` expires. Also, `InsXTime` resets the high-order bit of the `qType` field to 0.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `InsXTime` are

Registers on entry

A0 Address of the task record

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error

PrimeTime

Use the `PrimeTime` procedure to activate a task in the Time Manager queue.

```
PROCEDURE PrimeTime (tmTaskPtr: QElemPtr; count: LongInt);
```

`tmTaskPtr`	A pointer to a task record already installed in the queue.
`count`	The desired delay before execution of the task.

DESCRIPTION

The `PrimeTime` procedure schedules the task specified by the `tmAddr` field of `tmTaskPtr` for execution after the delay specified by the `count` parameter has elapsed.

If the `count` parameter is a positive value, it is interpreted as milliseconds. If `count` is a negative value, it is interpreted in negated microseconds. (Microsecond delays are allowable only in the revised and extended Time Managers.)

The task record specified by `tmTaskPtr` must already be installed in the queue (by a previous call to `InsTime` or `InsXTime`) before your application calls `PrimeTime`. `PrimeTime` returns immediately, and the specified task is executed after the specified delay has elapsed. If you call `PrimeTime` with a time delay of 0, the task runs as soon as interrupts are enabled.

In the revised and extended Time Managers, `PrimeTime` sets the high-order bit of the `qType` field to 1. In addition, any value of the `count` parameter that exceeds the maximum millisecond delay is reduced to the maximum. If you stop an unexpired task (by calling `RmvTime`) and then reinstall it (by calling `InsXTime`), you can continue the previous delay by calling `PrimeTime` with the `count` parameter set to 0.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `PrimeTime` are

Registers on entry

A0 Address of the task record

D0 Specified delay time (long)

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error

RmvTime

Use the `RmvTime` procedure to remove a task from the Time Manager queue.

```
PROCEDURE RmvTime (tmTaskPtr: QElemPtr);
```

`tmTaskPtr` A pointer to a task record to be removed from the queue.

DESCRIPTION

The `RmvTime` procedure removes the Time Manager task record specified by `tmTaskPtr` from the Time Manager queue. In both the revised and extended Time Managers, if the specified task record is active (that is, if it has been activated but the specified time has not yet elapsed), the `tmCount` field of the task record returns the amount of time remaining. To provide the greatest accuracy, the unused time is reported as negated microseconds if that value is small enough to fit into the `tmCount` field (even if the delay was originally specified in milliseconds); otherwise, the unused time is reported in positive milliseconds. If the time has already expired, `tmCount` contains 0.

In the revised and extended Time Managers, `RmvTime` sets the high-order bit of the `qType` field to 0.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `RmvTime` are

Registers on entry

A0 Address of the task record

Registers on exit

D0 Result code

RESULT CODES

noErr	0	No error

Application-Defined Routine

The Time Manager allows your software to install an application-defined routine that is executed after a specified delay.

Time Manager Tasks

You pass the address of an application-defined Time Manager task in the `tmAddr` field of the Time Manager task record.

MyTimeTask

A Time Manager task has the following syntax:

```
PROCEDURE MyTimeTask;
```

DESCRIPTION

The `tmAddr` field of a Time Manager task record contains the address of a task procedure that is executed after the delay time passed to `PrimeTime`.

SPECIAL CONSIDERATIONS

Because the task procedure is executed at interrupt time, it should not allocate, move, or purge memory (either directly or indirectly) and should not depend on the validity of handles to unlocked blocks.

ASSEMBLY-LANGUAGE INFORMATION

In the revised and extended Time Managers, when the task procedure is called, register A1 contains a pointer to the Time Manager task record associated with that procedure.

A task procedure must preserve all registers other than A0–A3 and D0–D3.

SEE ALSO

See the section “Using Application Global Variables in Tasks” on page 3-11 for instructions on how to access your application’s global variables from within a task procedure. See “Performing Periodic Tasks” on page 3-13 for instructions on how to define a periodic task procedure.

Summary of the Time Manager

Pascal Summary

Constants

```
CONST
   {Gestalt selector}
   gestaltTimeMgrVersion      = 'tmgr';   {Time Manager version}

   {values returned by Gestalt}
   gestaltStandardTimeMgr     = 1;        {original Time Manager}
   gestaltRevisedTimeMgr      = 2;        {revised Time Manager}
   gestaltExtendedTimeMgr     = 3;        {extended Time Manager}
```

Data Types

Original and Revised Time Manager Task Record

```
TYPE TMTask =
   RECORD
      qLink:         QElemPtr;      {next queue entry}
      qType:         Integer;       {queue type}
      tmAddr:        ProcPtr;       {pointer to task}
      tmCount:       LongInt;       {reserved}
   END;
```

Extended Time Manager Task Record

```
TYPE TMTask =
   RECORD
      qLink:         QElemPtr;      {next queue entry}
      qType:         Integer;       {queue type}
      tmAddr:        ProcPtr;       {pointer to task}
      tmCount:       LongInt;       {unused time}
      tmWakeUp:      LongInt;       {wakeup time}
      tmReserved:    LongInt;       {reserved for future use}
   END;
```

```
TMTaskPtr = ^TMTask;
```

Time Manager Routines

```
PROCEDURE InsTime           (tmTaskPtr: QElemPtr);
PROCEDURE InsXTime          (tmTaskPtr: QElemPtr);
PROCEDURE PrimeTime         (tmTaskPtr: QElemPtr; count: LongInt);
PROCEDURE RmvTime           (tmTaskPtr: QElemPtr);
```

Application-Defined Routine

```
PROCEDURE MyTimeTask;
```

C Summary

Constants

```
/*Gestalt selector*/
#define gestaltTimeMgrVersion    'tmgr'      /*Time Manager version*/

/*values returned by Gestalt*/
#define gestaltStandardTimeMgr   1           /*original Time Manager*/
#define gestaltRevisedTimeMgr    2           /*revised Time Manager*/
#define gestaltExtendedTimeMgr   3           /*extended Time Manager*/
```

Data Types

```
typedef pascal void (*TimerProcPtr)(void);
```

Original and Revised Time Manager Task Record

```
struct TMTask {
      QElemPtr       qLink;         /*next queue entry*/
      short          qType;         /*queue type*/
      TimerProcPtr   tmAddr;        /*pointer to task*/
      long           tmCount;       /*reserved*/
};
```

Extended Time Manager Task Record

```
struct TMTask {
      QElemPtr       qLink;         /*next queue entry*/
      short          qType;         /*queue type*/
      TimerProcPtr   tmAddr;        /*pointer to task*/
      long           tmCount;       /*unused time*/
      long           tmWakeUp;      /*wakeup time*/
      long           tmReserved;    /*reserved for future use*/
};

typedef struct TMTask TMTask;
typedef TMTask *TMTaskPtr;
```

Time Manager Routines

```
pascal void InsTime         (QElemPtr tmTaskPtr);
pascal void InsXTime        (QElemPtr tmTaskPtr);
pascal void PrimeTime       (QElemPtr tmTaskPtr, long count);
pascal void RmvTime         (QElemPtr tmTaskPtr);
```

Application-Defined Routine

```
pascal void MyTimeTask      (void);
```

Assembly-Language Summary

Data Structures

Structure of Original and Revised Time Manager Queue Entry

0	qLink	long	pointer to next queue entry
4	qType	word	queue type
6	tmAddr	long	pointer to task
10	tmCount	long	unused time; returned to caller

Structure of Extended Time Manager Queue Entry

0	qLink	long	pointer to next queue entry
4	qType	word	queue type
6	tmAddr	long	pointer to task
10	tmCount	long	unused time; returned to caller
14	tmWakeUp	long	wakeup time; used internally by the Time Manager
18	tmReserved	long	reserved for future use

Result Codes

noErr	0	No error

CHAPTER 4

Vertical Retrace Manager

Contents

Vertical Retrace Manager

This chapter describes the Vertical Retrace Manager, the part of the Operating System that schedules and executes recurrent tasks during vertical retrace interrupts. You can use the Vertical Retrace Manager to execute simple, repetitive tasks and avoid having to execute those tasks repeatedly in your application's main event loop.

You should read the information in this chapter if you want your application to schedule tasks for execution during a vertical retrace interrupt. For example, you can use the Vertical Retrace Manager to cycle among a series of cursors while some lengthy operation is happening, thus presenting the illusion of a spinning cursor.

In general, you should use the Vertical Retrace Manager only when you need to synchronize actions with the redrawing of the screen or when the tasks don't need to be executed at very precise intervals. As explained later in this chapter, certain conditions can cause the Operating System to turn off vertical blanking interrupts for a period of time. When this happens, the tasks in the vertical retrace task queue are not executed as scheduled. As a result, you should not use the Vertical Retrace Manager to handle tasks that must be executed consistently or with precise timing. For precise, uninterrupted task execution, you should use the Time Manager. See the chapter "Time Manager" in this book for details.

To use this chapter, you need to be familiar with interrupt-time processing and with the general limitations on such processing. The chapter "Introduction to Processes and Tasks" in this book describes these issues in detail. As emphasized in that chapter, you should in general avoid executing tasks at interrupt time. If you must install a VBL task, the code should be as short as possible. In addition, the code and any data it accesses should be locked into physical memory if virtual memory is in operation.

To use this chapter, you might also need to be familiar with techniques for accessing information in your application's A5 world at interrupt time. The chapter "Introduction to Memory Management" in *Inside Macintosh: Memory* describes the A5 world and the routines you can use to manipulate the A5 register. This chapter provides complete code samples that illustrate how to access your application's A5 world in a VBL task. As a result, you might be able to use the Vertical Retrace Manager to accomplish simple, repetitive tasks without reading that chapter.

This chapter describes how the Vertical Retrace Manager works and then shows how you can use the Vertical Retrace Manager to

- install a simple task to be executed during vertical retrace interrupts
- access information about a task record installed in the vertical retrace queue from within that task
- access your application's global variables in a vertical retrace task
- spin the cursor to indicate that the user must wait while the computer completes some lengthy processing
- install a vertical retrace task in the system heap so that it continues to be executed even when your application is switched out

About the Vertical Retrace Manager

The video circuitry in a Macintosh computer, whether built-in or external, refreshes the screen at regular intervals. For built-in monitors, the screen is refreshed approximately 60 times per second; for external monitors, the screen is refreshed at intervals determined by the associated video hardware. To refresh the screen, the monitor's electron beam draws one pixel at a time, starting at the upper-left corner of the screen and moving quickly to the lower-right corner. When the electron beam returns from the lower-right corner of the screen to the upper-left corner, the video circuitry generates a **vertical retrace interrupt** or **vertical blanking (VBL) interrupt.**

The Vertical Retrace Manager is the part of the Operating System that schedules and executes tasks—known as **VBL tasks**—during a vertical retrace interrupt. The Operating System itself uses the Vertical Retrace Manager to perform some important housekeeping operations, such as moving the cursor in response to mouse movements and checking whether the current application's stack has expanded into its heap. Within the limitations described in this chapter, you can use the Vertical Retrace Manager to install your own recurrent tasks. For example, you can use the Vertical Retrace Manager to spin the cursor to indicate that the user must wait while some processing initiated by your application completes.

In general, the Vertical Retrace Manager is useful for small, repetitive tasks that do not allocate or release memory and that you do not want to execute in your main event loop. Whenever possible, it is best to manage periodic tasks within your main event loop. For example, you can call the TextEdit routine `TEIdle` once each time through the loop, thus causing the insertion point in a block of text to blink. However, if you want some task to execute repetitively at a time when you do not want to reenter your main event loop (perhaps because you don't want your application to be switched out during some lengthy operation), it might be possible to use the Vertical Retrace Manager to execute the task.

The principal limitation on VBL tasks (aside from the limitations on any interrupt-time processing) is that they cannot execute more frequently than once per VBL interrupt. The exact amount of time between successive VBL interrupts depends on the refresh frequency of the screen, which varies. On Macintosh computers that have a built-in screen (such as the Macintosh Plus or Macintosh Classic), the vertical retrace frequency is approximately 60.15 Hz, resulting in a period of approximately 16.63 milliseconds. If you need a task to be executed more often than that, you should use the Time Manager, which has a much finer resolution (up to 250 microseconds for drift-free task execution).

Unlike the Time Manager, the Vertical Retrace Manager is not an absolute timing mechanism. Its operations are always relative to the VBL interrupt, which may be disabled (for instance, during disk access). As a result, you should use the Time Manager in cases where absolute time delays are important. Use the Vertical Retrace Manager, however, in cases where the scheduled actions need simply to be synchronized with other VBL tasks, such as moving the cursor or refreshing the screen.

VBL Tasks Installed by the Operating System

The Operating System uses the Vertical Retrace Manager to accomplish a number of repetitive tasks at uniform intervals. These are some of the VBL tasks installed by the Operating System, grouped by the intervals at which they execute:

- Every interrupt
  - Update the value of the global variable `Ticks`, which a program may access through the routine `TickCount`.
  - Call the "stack sniffer" to see if the current application's stack and heap have collided. If so, the task calls the System Error Handler.
  - Update the position of the cursor.
- Every 30 interrupts
  - Check whether the user has inserted a disk or mounted a volume. If so, the task posts a disk-inserted event.
- Every 32 interrupts
  - Check whether a keyboard has been reattached after having been detached. If so, the task resets the keyboard.

Some VBL routines may execute only on certain computers or only in certain versions of system software. For example, on early Macintosh computers, a VBL task checks every other interrupt to determine whether the state of the mouse button has changed from its previous state and then remained unchanged for at least four interrupts. If so, that task posts a mouse-down or mouse-up event, as appropriate. In Macintosh computers equipped with Apple Desktop Bus mouse devices, the Operating System uses a different mechanism for posting mouse-down and mouse-up events.

Note

VBL tasks installed by the Operating System are not maintained in the same queue used for application-defined VBL tasks. u

Types of VBL Tasks

There are two general types of VBL tasks. A **slot-based VBL task** is linked to an external video monitor. Because different monitors can have different refresh rates and hence might execute VBL tasks at different times, the Vertical Retrace Manager maintains a separate task queue for each video device attached to the computer. When a VBL interrupt occurs for a particular device, the Vertical Retrace Manager executes any tasks in the queue for the slot holding that monitor's video card. You can install a slot-based VBL task by calling the `SlotVInstall` function.

For Macintosh computers that have only a built-in monitor (such as a Macintosh Plus or Macintosh Classic), there is no need to isolate VBL tasks into separate queues. Instead, the Operating System maintains just one task queue and processes the tasks in that queue when it receives a VBL interrupt. A VBL task that is not linked to an external video device is known as a **system-based VBL task.** You can install a system-based VBL task by calling the `VInstall` function.

To maintain compatibility on modular Macintosh computers for software that uses the `VInstall` function, the Operating System generates a special interrupt at a frequency identical to the retrace rate on compact Macintosh computers. This special interrupt is generated approximately 60.15 times a second and mimics the vertical retrace interrupt on compact models. This ensures that application tasks installed using the `VInstall` function, as well as periodic system tasks such as updating the tick count and checking whether the stack has expanded into the heap, are performed as usual.

To ensure the synchronization of your VBL task with the retracing of the screen, you should check whether the `SlotVInstall` function is available in the current operating environment. If it is, you should use the slot-based routines to install and remove your VBL task. If not, you should use the system-based routines.

However, even if you synchronize your VBL task to the retracing of the screen correctly, tasks may not always execute as scheduled. Some types of system activity, such as disk access, may cause VBL interrupts to be disabled temporarily. (This is why cursor movement sometimes becomes jerky during disk operations.) Also, if a VBL task takes longer to perform than the time it takes to retrace the screen, other interrupt tasks may miss one or more vertical retrace interrupts.

Like all interrupt tasks, VBL tasks cannot do everything that ordinary routines can. The following list summarizes the operations that VBL tasks should not perform. A VBL task that violates one of these rules may cause a system crash:

- A VBL task must not allocate, move, or purge memory, or call any Toolbox routines that might do so.
- A VBL task must preserve all registers other than A0–A3 and D0–D3.
- A VBL task cannot call a routine from another code segment unless it sets up the application's A5 world properly. In addition, that segment must already be loaded in memory.
- A VBL task cannot access your application global variables unless it sets up the application's A5 world properly. This technique is explained in "Accessing Application Global Variables in a VBL Task," beginning on page 4-13.
- A VBL task's code and any data accessed during the execution of the task must be locked into physical memory if virtual memory is in operation.

The VBL Task Record

You install a VBL task by passing the Vertical Retrace Manager the address of a **VBL task record,** which holds information about your VBL task. This information includes the address of the procedure the Vertical Retrace Manager is to execute at interrupt time and the number of interrupts before it should next execute the task. The `VBLTask` data type defines a VBL task record.

```
TYPE VBLTask =
RECORD
   qLink:    QElemPtr;   {next entry in vertical retrace queue}
   qType:    Integer;    {queue type}
   vblAddr:  ProcPtr;    {pointer to task procedure}
   vblCount: Integer;    {interrupts until next execution}
   vblPhase: Integer;    {task phase}
END;
```

Your application needs to fill in only the `qType`, `vblAddr`, `vblCount`, and `vblPhase` fields of the VBL task record. The `qLink` field, which contains a pointer to the next entry in the VBL task's vertical retrace queue, is set by the Vertical Retrace Manager when you install the task by calling `VInstall` or `SlotVInstall`. Your application does not need to set up the `qLink` field.

The Vertical Retrace Manager installs your VBL task record into the appropriate VBL queue. A vertical retrace queue is a standard operating-system queue.

Note
For more information about the structure of operating-system queues, see the chapter "Queue Utilities" in *Inside Macintosh: Operating System Utilities*. u

You must set the `qType` field to `ORD(vType)` before you install the task. This specifies that the task's queue is a vertical retrace queue and not some other type of operating-system queue.

The `vblAddr` field holds a pointer to the procedure that the Vertical Retrace Manager is to execute.

When installing a VBL task, you specify, in the `vblCount` field, the number of interrupts before the routine first executes. The Vertical Retrace Manager lowers this number by 1 during each interrupt. If decrementing `vblCount` produces a value of 0, the Vertical Retrace Manager executes the procedure specified in the task record's `vblAddr` field. If you want the procedure to be executed again, that procedure is responsible for resetting the value of the `vblCount` field to the desired value.

If you do not want the Vertical Retrace Manager to execute the task again, your task should leave the value of `vblCount` at 0. Setting the `vblCount` field to 0 is one way of disabling a task. (A more common approach is to remove the task record from its queue by calling `VRemove` or `SlotVRemove`, but this should not be done by the VBL task itself.) Note that if you set `vblCount` to 0 when installing a VBL task, the task will never execute. If you want a task to execute immediately upon installation, set `vblCount` to 1.

The `vblPhase` field specifies the task's phase count, indicating which interrupts are to trigger the execution of the VBL task. You can set two VBL tasks installed at the same time and scheduled for execution after the same number of interrupts out of phase with one another—that is, executed during different interrupts—by specifying different phase counts for each task. Unless you add many tasks to a VBL queue at one time, you can usually set `vblPhase` to 0.

Vertical Retrace Queues

The Vertical Retrace Manager stores application-defined VBL task records in **vertical retrace queues**, which are standard operating-system queues. If multiple tasks in the same vertical retrace queue are scheduled to be executed during the same interrupt, the Vertical Retrace Manager will execute the tasks in the order they were installed in the queue.

Compact Macintosh computers maintain only one vertical retrace queue, because these computers have only one screen. However, computers with multiple screens require multiple vertical retrace queues. Because slot-based task installation and removal routines apply to just one slot, the Vertical Retrace Manager maintains a separate vertical retrace queue for each slot that contains a video card. In addition, to maintain compatibility with the system-based VBL task installation and removal routines, the Vertical Retrace Manager maintains a single, system-based vertical retrace queue for all applications to share.

Ordinarily, you do not need to inspect or manipulate the contents of vertical retrace queues directly. Instead, you can use the Vertical Retrace Manager routines for installing task records in and removing them from vertical retrace queues.

In one case, however, you might need to inspect the header of a vertical retrace queue. If you need to know whether some code is being called in response to a VBL interrupt, you can inspect the `qFlags` field of the queue header. The Vertical Retrace Manager sets bit 6 of the `qFlags` field in the queue header to indicate that a VBL task in the queue is being executed.

Assembly-Language Note

You can use the global constant `inVBL` to test this bit. u

VBL Tasks and Application Execution

Often, a VBL task performs services that are useful only to the application that installed it. For instance, consider the VBL task defined in “Spinning the Cursor” beginning on page 4-16. This task spins the cursor while your application performs some lengthy operation and should be executed only if your application is in the foreground. If the user switches your application into the background while it is occupied with that lengthy operation, you probably want to disable that task for as long as your application is in the background. Otherwise, the cursor will continue to spin, probably confusing the user.

In other cases, a VBL task should continue to be executed even when the application that installed it is no longer in the foreground. For instance, you probably wouldn’t want to disable a VBL task that periodically checks for the arrival of electronic mail just because your application is moved to the background.

The Process Manager automatically disables a system-based VBL task when the application that installed it is swapped out in a major or minor switch, if the address of the VBL task is anywhere in the application’s partition. Then, when that application regains control of the processor, the Process Manager reenables that VBL task. If,

however, the address of a system-based VBL task is in the system partition, the VBL task continues to be executed, regardless of the processing status of the application that launched it.

Note

When your system-based VBL task continues to be executed in this way, the Process Manager does not restore the context of your application before executing the VBL task. In particular, any trap patches installed by your application might not be available to the VBL task. When a VBL task depends on your application context, your task can call the Process Manager function `GetCurrentProcess` to check whether your application is the current process and hence that its context is valid. u

The address of a system-based VBL task, not the address of the VBL task record, determines whether the Process Manager disables the task. See "Installing a Persistent VBL Task," beginning on page 4-20, for a technique you can use to prevent the disabling of a task when the application that installed it is switched out.

By contrast, the Process Manager never disables a slot-based VBL task, no matter where the task is located. As a result, if you want to disable a slot-based VBL task when your application is in the background, you must do so yourself, either by removing the task record from the VBL queue or by setting the `vblCount` field of the task record to 0. You can do this in response to a suspend event. Then, when your application receives a resume event, you can reenable the VBL task by reinstalling the task record or by resetting the `vblCount` field of the task record to the appropriate value.

In some cases, you might want to disable a *system*-based VBL task manually, even though the Process Manager also disables it when your application is switched out. This is because the Process Manager reenables system-based VBL tasks when your application receives processing time as a result of a minor switch, when your application is still in the background. If the VBL task should be executed only when your application is in the foreground, you need to disable it when your application receives a suspend event and reenable it when your application receives a resume event. The easiest way to do this is to set and reset the `vblCount` field of the task record, as described in the previous paragraph.

The Process Manager treats VBL tasks slightly differently when your application quits or crashes than when it is switched out. If either the task record for a VBL task or the code of the VBL task is located in your application partition, the Process Manager removes that task record from its VBL queue. (This is true for both slot-based and system-based VBL tasks.) Conversely, if both a VBL task record and the task itself are located in the system partition, the Process Manager doesn't remove the task record from its VBL queue when the application that installed them quits or crashes.

s **WARNING**

Failure to remove VBL task records installed in the system partition from their queues can lead to a system crash if the VBL task is located in the system partition but accesses data in your application partition. Because the Process Manager deallocates your application partition when your application quits or crashes, the VBL task may attempt to

access undefined data. The easiest way to avoid this problem is to patch the Process Manager's `ExitToShell` procedure so that it removes all VBL task records installed by your application. s

Using the Vertical Retrace Manager

You can use the Vertical Retrace Manager to install VBL task records in and remove VBL task records from system-based or slot-based vertical retrace queues. To install a task record, you must first fill in some of its fields and then call either `VInstall` or `SlotVInstall`. See the next section, "Installing a VBL Task," for information on installing VBL tasks.

If it is to be executed more than once, a VBL task must access the task record and reset the value of the task record's `vblCount` field. The section "Accessing a Task Record at Interrupt Time" on page 4-12 describes this technique. To disable a task temporarily, you can simply set the `vblCount` field of its task record to 0. To remove a VBL task from its VBL queue, call `VRemove` if you installed the task by calling `VInstall` or call `SlotVRemove` if you installed the task by calling `SlotVInstall`.

If your VBL task needs to access your application global variables, you can put the application's A5 value or the global variables themselves into the second field of a record whose first field contains the VBL task itself. The sections "Accessing Application Global Variables in a VBL Task," beginning on page 4-13, and "Spinning the Cursor," beginning on page 4-16, explain these techniques.

Installing a VBL Task

For any particular VBL task, you need to decide whether to install it as a system-based VBL task or as a slot-based VBL task. You need to install a task as a slot-based VBL task only if the execution of the task needs to be synchronized with the retrace rate of a particular external monitor. If the task performs no processing that is likely to affect the appearance of the screen or that depends on the state of an external monitor, it is probably safe to install the task as a system-based VBL task.

s **WARNING**

If you do decide that the execution of some VBL task needs to be synchronized with the retrace rate of a monitor, you should first check that the `SlotVInstall` function is available in the operating environment. You can do this by calling the `TrapAvailable` function defined in the chapter "Gestalt Manager" in *Inside Macintosh: Operating System Utilities*. If you call `SlotVInstall` and it is not available, your application will crash. s

If you are uncertain whether to install a task as a system-based or as a slot-based VBL task, you should first install it as a system-based task (by calling `VInstall`). Then test your application on a modular Macintosh computer with an external monitor whose refresh rate is different from the refresh rate on a compact Macintosh computer

(approximately 60.15 Hz). If any screen updating that occurs as a result of processing done by your VBL task has an unacceptable appearance, you probably need to install the task as a slot-based VBL task (by calling `SlotVInstall`). Remember, however, to check whether `SlotVInstall` is available before you call it; if it isn't available, call `VInstall`. You can determine whether `SlotVInstall` is available by calling the `SlotRoutinesAvailable` function defined in Listing 4-1.

Listing 4-1 Checking whether you can use slot-based VBL routines

```
FUNCTION SlotRoutinesAvailable: Boolean;
CONST
   _SlotVInstall = $A06F;
BEGIN
   SlotRoutinesAvailable := TrapAvailable(_SlotVInstall);
END;
```

If the slot-based routines are available and you want to use them, you need to know the slot number of the video device to whose retrace the VBL task is to be synchronized. Listing 4-2 illustrates a way to find the slot number of the main graphics device. To access the device control entry for the main graphics device, you must first find the device's reference number. Then you can cast the device control entry into type `AuxDCEHandle` and access the slot number directly. You can use a similar technique to find the slot number of some other graphics device.

Listing 4-2 Determining the slot number of the main graphics device

```
FUNCTION MainSlotNumber: Integer;
VAR
   mainDeviceRefNum: Integer;    {number of main graphics device}
BEGIN
   mainDeviceRefNum := GetMainDevice^^.gdRefNum;
   MainSlotNumber :=
      AuxDCEHandle(GetDCtlEntry(mainDeviceRefNum))^^.dCtlSlot;
END;
```

Note

For the sake of simplicity, the remainder of this chapter illustrates how to use the Vertical Retrace Manager to handle system-based VBL tasks only. Virtually all of the techniques shown here, however, can be used in connection with slot-based VBL tasks as well. u

The `InstallVBL` function defined in Listing 4-3 shows how to fill in a VBL task record and install it in the system-based VBL queue. It assumes that the task record `gMyVBLTask` is a global variable of type `VBLTask` and that you have already defined the procedure `DoVBL`, the actual VBL task. That procedure is subject to all of the usual

limitations on VBL and other interrupt tasks. Also, if DoVBL is to be executed recurrently, it must reset the vblCount field of the task record each time it is executed. The next section, "Accessing a Task Record at Interrupt Time," describes how to do this.

Listing 4-3 Initializing and installing a task record

```
FUNCTION InstallVBL: OSErr;
CONST
   kInterval = 6;                {frequency in interrupts}
BEGIN
   WITH gMyVBLTask DO            {initialize the VBL task}
      BEGIN
         qType := ORD(vType);    {set queue type}
         vblAddr := @DoVBL;      {set address of VBL task}
         vblCount := kInterval;  {set task frequency}
         vblPhase := 0;          {no phase}
      END;

   InstallVBL := VInstall(@gMyVBLTask);
END;
```

Accessing a Task Record at Interrupt Time

A repetitive VBL task must access its task record so that it can reset the vblCount field. As explained in "The VBL Task Record" on page 4-6, the Vertical Retrace Manager decrements the vblCount field during each interrupt and executes the task when that field reaches 0. The task is removed from its queue if the value of the vblCount field is left at 0.

When the Vertical Retrace Manager executes the VBL task, it places the address of the VBL task record into the A0 register. Listing 4-4 defines an inline function that moves this value onto the stack.

Note
You should call the inline function in Listing 4-4 only from a VBL task. It will not work if called from your main program. In addition, the call to this function should be the first line of your VBL task, because other processing might change the value in A0. u

Listing 4-4 Finding the address of the task record from within a VBL task

```
FUNCTION GetVBLRec: LongInt;
      INLINE $2E88;              {MOVE.L A0,(SP)}
```

The `GetVBLRec` function defined in Listing 4-4 returns a long integer specifying the address of the task record. Now that you can access the task record, you can easily reset the value of the `vblCount` field. Listing 4-5 provides an example of a generic VBL task that accesses the task record and resets the `vblCount` field.

Listing 4-5 Resetting a VBL task so that it executes again

```
PROCEDURE DoVBL;
CONST
   kInterval = 6;                    {frequency in interrupts}
TYPE
   VBLTaskPtr = ^VBLTask;            {pointer to a VBLTask record}
VAR
   taskPtr: VBLTaskPtr;
BEGIN
   taskPtr := VBLTaskPtr(GetVBLRec);{get address of task record}

   {Put task-specific code here.}

   taskPtr^.vblCount := kInterval;  {reset vblCount}
END;
```

Accessing Application Global Variables in a VBL Task

The Operating System stores the address of the boundary between the current application's global variables and its application parameters in the microprocessor's A5 register. For this reason, most compilers generate references to application global variables as offsets from the address contained in the A5 register. Therefore, if the value in register A5 does not point to the boundary between your application's global variables and its application parameters, your attempts to access your application's global variables will fail.

Ordinarily, applications do not need to keep track of the value in the A5 register. Although all applications share the register, the Process Manager keeps track of the address of your application's A5 world when a major or minor switch causes your application to yield the CPU to other processes, and it restores that value when your application regains access to the CPU. The A5 register is guaranteed to be correct for all code that your application executes directly (that is, for all code that is not executed in response to an interrupt or by a Toolbox or Operating System routine).

Because VBL tasks are interrupt routines, they might be executed when the value in the A5 register does not point to the A5 world of your application. As a result, if you want to access your application's global variables in a VBL task, you need to set the A5 register to its correct value when your VBL task begins executing and restore the previous value upon exit.

Note
For a more complete discussion of the A5 register, see the chapter "Memory Management Utilities" in *Inside Macintosh: Memory.* u

The solution to this problem is to find a memory location that both the main program and the VBL task can access. The main program can store the value of register A5 there, and the VBL task can set A5 correctly by reading that value. The functions `SetCurrentA5` and `SetA5` can be used for this purpose. The application can store the value of its A5 register by calling `SetCurrentA5`. Then, at interrupt time, the task can begin by calling `SetA5` to set the register to that value and end by calling `SetA5` again, this time to restore the register to its initial value, the one used by the main program.

The only memory location that a VBL task has access to is the address of the task record, as explained in the previous section, "Accessing a Task Record at Interrupt Time." So, if your application stores the value of A5 directly following the task record in memory, it can locate the value of A5 by first locating the task record. You can do this by defining a new data type (called `VBLRec` in Listing 4-6) whose first field contains the VBL task and whose second field contains a long integer specifying the value of the A5 register.

Listing 4-6 Storing the value of the A5 register directly after the task record in memory

```
TYPE VBLRec =
   RECORD
      myVBLTask:     VBLTask;    {the actual VBL task record}
      vblA5:         LongInt;    {saved value of application's A5}
   END;

   VBLRecPtr = ^VBLRec;
```

Now you can modify the application-defined procedure that installs a VBL task so that it stores the value of register A5 in the `vblA5` field of the `VBLRec`, as illustrated in Listing 4-7.

Listing 4-7 Saving the value of the A5 register when installing a VBL task

```
FUNCTION InstallVBL: OSErr;
CONST
   kInterval = 6;                  {frequency in interrupts}
BEGIN
   WITH gMyVBLRec.myVBLTask DO     {initialize the VBL task}
      BEGIN
         qType := ORD(vType);      {set queue type}
         vblAddr := @DoVBL;        {set address of VBL task}
         vblCount := kInterval;    {set task frequency}
         vblPhase := 0;            {no phase}
```

```
      END;
   myVBLRec.vblA5 := SetCurrentA5;  {get our A5}

   InstallVBL := VInstall(@gMyVBLRec.myVBLTask);
END;
```

You must also modify the VBL task so that it sets and restores the value of register A5 correctly. Listing 4-8 illustrates a simple VBL task that increments the global variable `gCounter` and then resets itself to run again after the specified number of interrupts.

Listing 4-8 Setting up the A5 register and modifying a global variable in a VBL task

```
PROCEDURE DoVBL;
CONST
   kInterval = 6;                   {frequency in interrupts}
VAR
   curA5:   LongInt;                {stored value of A5}
   recPtr:  VBLRecPtr;              {pointer to task record}
BEGIN
   recPtr := VBLRecPtr(GetVBLRec);  {get address of task record}
   curA5 := SetA5(recPtr^.vblA5);   {set our application's A5 }
                                    { and store old A5 in curA5}
   gCounter := gCounter + 1;        {modify a global variable}

   {Reset vblCount so that this procedure executes again.}
   recPtr^.myVBLTask.vblCount := kInterval;

   curA5 := SetA5(curA5);           {restore the old A5 value}
END;
```

Because of the optimizations performed by some compilers, the actual work of the VBL task and the setting and restoring of the A5 register might have to be placed in separate procedures. If necessary, you can define a routine `DoVBL` that loads the proper value of A5, calls another routine called `RunVBL`, and then restores the old value of A5. The `RunVBL` routine does the work of the VBL task and resets the task record's `vblCount` field so that the `DoVBL` routine executes again. Listing 4-9 illustrates a sample definition of the `RunVBL` function that modifies an application global variable.

Listing 4-9 Modifying application global variables in a VBL task

```
PROCEDURE RunVBL (aRecPtr: VBLRecPtr);
CONST
   kInterval = 6;                   {frequency in interrupts}
BEGIN
```

```
   gCounter := gCounter + 1;        {modify global variable}

   {Reset vblCount so that this procedure executes again.}
   aRecPtr^.myVBLTask.vblCount := kInterval;
END;
```

Listing 4-10 shows how to call RunVBL from the VBL task.

Listing 4-10 Setting up and restoring the A5 register in a VBL task

```
PROCEDURE DoVBL;
VAR
   curA5:   LongInt;                {stored value of A5}
   recPtr:  VBLRecPtr;              {pointer to task record}
BEGIN
   recPtr := VBLRecPtr(GetVBLRec);  {get address of task record}
   curA5 := SetA5(recPtr^.vblA5);   {set our application's A5 }
                                    { and store old A5 in curA5}
   RunVBL(recPtr);                  {run the actual VBL task}
   curA5 := SetA5(curA5);           {restore the old A5 value}
END;
```

If this separation of routines is necessary, you must make sure that the two routines (DoVBL and RunVBL) are in the same code segment.

Spinning the Cursor

Some VBL tasks need access only to global variables that they do not share with the main program. For example, you might wish to design a VBL task that animates the beachball or watch cursor to indicate that the user must wait while the computer finishes some lengthy processing. The main application might use the application-defined procedures StartSpinning and StopSpinning to install and remove the VBL task, but the application might not need to know, for example, which beachball or watch cursor the VBL task is displaying at any given time. The VBL task itself would need to know this information, because it must know which cursor to display when it is time to change the cursor.

One way to implement such a VBL task is to use application global variables and set up the A5 register properly, as described in the previous section, "Accessing Application Global Variables in a VBL Task." An alternate method, however, is simply to store the information that the VBL task needs directly after the task record in memory, just as you can store information about the program's A5 value there. Then, because the VBL task has access to all of the information it needs, it does not need to set up and restore the A5 register.

The listings that follow use that strategy to implement cursor spinning. This cursor spinning task implements simple animation of any number of cursor frames stored in contiguous resources in the program's resource fork.

Listing 4-11 provides a type definition for a cursor information record. This record holds the task record and information specific to cursor spinning. Listing 4-11 also defines several constants and a global variable to hold a cursor information record.

Listing 4-11 Defining a cursor information record

```
CONST
   kInterval = 4;                    {frequency in interrupts}
   kNumberOfCursors = 4;             {total number of frames}
   kInitialResID = 128;              {ID of first cursor resource}
TYPE
   CursorsList = ARRAY[1..kNumberOfCursors] OF CursHandle;
   CursorTask =
      RECORD
         myVBLTask:  VBLTask;        {the actual VBLTask}
         myCursors:  CursorsList;    {handles to the cursors}
         myFrame:    Integer;        {cursor frame to display next}
      END;
   CursorTaskPtr = ^CursorTask;
VAR
   gMyCursTask:   CursorTask;        {global cursor info. record}
```

Listing 4-12 shows the VBL task itself. The task changes the cursor and resets the task record's `vblCount` field so that the Vertical Retrace Manager executes the task again.

Listing 4-12 Changing the cursor within a VBL task

```
PROCEDURE ChangeCursor;
TYPE
   BooleanPtr = ^Boolean;                    {to check a low-memory global}
VAR
   recPtr:  CursorTaskPtr;
BEGIN
   recPtr := CursorTaskPtr(GetVBLRec);       {get cursor information}
   {If the cursor is busy, we should not change it.}
   IF NOT BooleanPtr(CrsrBusy)^ THEN
      WITH recPtr^ DO                        {update cursor information}
      BEGIN
         SetCursor(myCursors[myFrame]^^);    {display the next cursor}
         myFrame := myFrame + 1;             {advance to next cursor frame}
```

```
         IF myFrame > kNumberOfCursors THEN
            myFrame := 1;                    {wrap around to first frame}
      END;
   recPtr^.myVBLTask.vblCount := kInterval;  {set task to run again}
END;
```

The `ChangeCursor` procedure retrieves the address of the VBL task record. If the cursor isn't already being changed, then `ChangeCursor` changes the cursor to the next one in sequence and resets the index of the next cursor to display. Finally, `ChangeCursor` sets itself to run again after the appropriate number of interrupts have occurred.

Note

It is permissible to call `SetCursor` at interrupt time, provided that the cursor handle is locked and that some other routine is not currently modifying the cursor. The system global variable `CrsrBusy` has the value `TRUE` if the cursor is busy; in that case, you should not call `SetCursor`. Listing 4-12 illustrates the proper way to change the cursor at interrupt time. u

Listing 4-13 defines the procedure `StartSpinning`, which you can call before beginning some lengthy operation. Because VBL tasks cannot depend on the validity of unlocked handles, the `StartSpinning` procedure must lock the cursor handles in memory before `SetCursor` is called in the `ChangeCursor` procedure.

Listing 4-13 Installing the cursor-spinning task into a vertical retrace queue

```
PROCEDURE StartSpinning;
CONST
   kInitialDelay = 120;          {initial delay before starting to spin}
VAR
   myErr:   OSErr;
   count:   Integer;
BEGIN
   {Initialize cursor information.}
   FOR count := 1 TO kNumberOfCursors DO
   BEGIN
      {Load cursor into memory.}
      gMyCursTask.myCursors[count] := GetCursor(kInitialResID + count - 1);
      {Lock cursor so that we can call SetCursor at interrupt time.}
      HLockHi(Handle(gMyCursTask.myCursors[count]));
   END;
   gMyCursTask.myFrame := 1;     {display cursor with kInitialResID first}

   WITH gMyCursTask.myVBLTask DO {initialize the VBL task record}
   BEGIN
```

```
      qType := ORD(vType);       {set queue type}
      vblAddr := @ChangeCursor;  {get address of VBL task}
      vblCount := kInitialDelay; {set task frequency}
      vblPhase := 0;             {no phase}
   END;

   myErr := VInstall(@gMyCursTask.myVBLTask);
END;
```

Notice that the initial delay (specified by the `kInitialDelay` constant in the `vblCount` field) is much larger than the number of interrupts between subsequent cursor changes (specified by the `kInterval` constant). This prevents the cursor from starting to spin until a reasonable time (about 2 seconds) has elapsed.

Listing 4-14 shows how to remove the cursor-spinning task from the vertical retrace queue.

Listing 4-14 Removing the cursor-spinning task from its vertical retrace queue

```
PROCEDURE StopSpinning;
VAR
   myErr:   OSErr;
   count:   Integer;
BEGIN
   {Remove the task record from its queue.}
   myErr := VRemove(@gMyCursTask.myVBLTask);

   {Free memory occupied by the cursors.}
   FOR count := 1 TO kNumberOfCursors DO
      ReleaseResource(Handle(gMyCursTask.myCursors[count]));

   InitCursor;                        {restore the arrow cursor}
END;
```

Depending on the needs of your application, you might want to load the cursors into memory at application-launch time and release them when your application quits. If so, you need to modify the `StartSpinning` and `StopSpinning` procedures accordingly.

Installing a Persistent VBL Task

A **persistent VBL task** continues to be executed even when the Process Manager switches out the application that installed it and that application is no longer in control of the CPU. If you want to install a persistent system-based VBL task, you need to load its VBL task record into the system partition. (Slot-based VBL tasks are always persistent, no matter where you put the task record.) Listing 4-15 illustrates a simple way to load a VBL task record into the system heap.

Listing 4-15 Installing a persistent VBL task

```
FUNCTION InstallPersistentVBL (VAR theVBLRec: VBLTask): OSErr;
TYPE
   ProcPtrPtr = ^ProcPtr;                        {a pointer to a ProcPtr}
CONST
   kJMPInstr = $4EF9;                            {this is an absolute JMP}
   kJMPSize = 6;                                 {size of an absolute JMP}
VAR
   myErr:      OSErr;
   SysHeapPtr: Ptr;
   tempPtr:    Ptr;
BEGIN
   SysHeapPtr := NewPtrSys(kJMPSize);            {get a block in system heap}
   myErr := MemError;
   IF myErr <> noErr THEN                        {make sure we have the block}
      BEGIN
         InstallPersistentVBL := myErr;
         Exit(InstallPersistentVBL);
      END;
   IntegerPtr(SysHeapPtr)^ := kJMPInstr;         {move in the JMP instruction}
   tempPtr := Ptr(ORD(SysHeapPtr)+SizeOf(Integer));
   ProcPtrPtr(tempPtr)^ := theVBLRec.vblAddr;    {move in the JMP address}
   theVBLRec.vblAddr := ProcPtr(SysHeapPtr);     {point record at sys heap}
   InstallPersistentVBL := VInstall(@theVBLRec);{install the VBL task record}
END;
```

The `InstallPersistentVBL` function defined in Listing 4-15 allocates enough bytes in the system heap to hold an integer that encodes an assembly-language `JMP` instruction together with the absolute address to which to jump. It loads into that space the assembly-language instruction and the address of the original VBL task, which is extracted from the VBL task record passed to it as a parameter. Then `InstallPersistentVBL` replaces the address of the original VBL task in that record with the address of the block in the system heap. The net result is that the `vblAddr` field of the VBL task record now contains an address in the system partition, making the VBL task persistent.

Vertical Retrace Manager Reference

This section describes the data structure and routines provided by the Vertical Retrace Manager. The section "Data Structure" shows the Pascal data structure for the VBL task record. The section "Vertical Retrace Manager Routines" describes the routines you can use to install and remove slot-based and system-based VBL tasks; it also describes several utility routines for advanced programmers. The section "Application-Defined Routine" describes VBL tasks.

Data Structure

This section describes the VBL task record, the data structure you use to install VBL tasks in and remove them from vertical retrace queues.

The VBL Task Record

A VBL task record describes a vertical retrace task. It indicates which task record (if any) comes next in the vertical retrace queue, what procedure to use for the task, how many interrupts to wait before the task is executed, and in what phase to execute the task. The `VBLTask` data type defines a VBL task record.

```
TYPE VBLTask =
RECORD
   qLink:      QElemPtr;   {next entry in vertical retrace queue}
   qType:      Integer;    {queue type}
   vblAddr:    ProcPtr;    {pointer to task procedure}
   vblCount:   Integer;    {interrupts until next execution}
   vblPhase:   Integer;    {task phase}
END;
```

Field descriptions

`qLink` A pointer to the next entry in the task's vertical retrace queue.

`qType` The queue type. This field must be set to `ORD(vType)`.

`vblAddr` A pointer to the VBL task.

`vblCount` The number of interrupts between successive calls to the VBL task specified in the `vblAddr` field. If the value of `vblCount` is 0, the task will never be executed. The Vertical Retrace Manager decrements the value of this field after each interrupt. If decrementing `vblCount` produces a value of 0, the Vertical Retrace Manager executes the task. The task must then reset `vblCount`, or its entry will be removed from the queue after it has been executed.

`vblPhase` The phase count of the VBL task. In most cases, you can set this field to 0. However, if you install multiple tasks with the same `vblCount` at the same time, you can assign them different `vblPhase` values so that the tasks are not executed during the same interrupt. The value of the `vblPhase` field must be less than the value of the `vblCount` field.

For more information about using the `vblCount` and `vblPhase` fields, see "The VBL Task Record" on page 4-6.

Vertical Retrace Manager Routines

This section describes routines that allow you to install slot-based and system-based task records in vertical retrace queues and to remove task records. This section also describes utility routines that are of interest only to advanced programmers.

Slot-Based Installation and Removal Routines

You can use the functions `SlotVInstall` and `SlotVRemove` to install task records in and remove them from slot-based vertical retrace queues.

SlotVInstall

You can use the `SlotVInstall` function to install a task record in a slot-based vertical retrace queue.

```
FUNCTION SlotVInstall (vblTaskPtr: QElemPtr; theSlot: Integer):
                          OSErr;
```

`vblTaskPtr` A pointer to the task record to add to a queue.

`theSlot` The slot number of the video device to whose vertical retrace queue the task record is added.

DESCRIPTION

The `SlotVInstall` function installs the task record specified by the `vblTaskPtr` parameter in the vertical retrace queue associated with the video device specified by the `theSlot` parameter. The Vertical Retrace Manager executes the task at intervals determined by the task record's `vblCount` and `vblPhase` fields. The task must reset the value of the task record's `vblCount` field if you want the task to be executed again.

The Vertical Retrace Manager continues to execute tasks installed using the `SlotVInstall` function even when the application that installed them is switched out.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `SlotVInstall` are

Registers on entry

A0 Pointer to the task record

D0 Slot number of the device associated with the vertical retrace queue

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`vTypErr`	–2	Invalid `qType` value (must be `ORD(vType)`)
`slotNumErr`	–360	Invalid slot number

SlotVRemove

You can use the `SlotVRemove` function to remove a task record from a slot-based vertical retrace queue.

```
FUNCTION SlotVRemove (vblTaskPtr: QElemPtr; theSlot: Integer):
                      OSErr;
```

`vblTaskPtr`
A pointer to the task record to remove from its queue.

`theSlot` The slot number of the video device from whose vertical retrace queue the task record is removed.

DESCRIPTION

The `SlotVRemove` function removes the task record specified by the `vblTaskPtr` parameter from the vertical retrace queue associated with the video device specified by the `theSlot` parameter.

To disable a slot-based VBL task temporarily, you can set the `vblCount` field of the task record to 0.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `SlotVRemove` are

Registers on entry

A0 Pointer to the task record

D0 Slot number of the device associated with the vertical retrace queue

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`qErr`	–1	Task record isn't in the queue
`vTypErr`	–2	Invalid `qType` value (must be `ORD(vType)`)
`slotNumErr`	–360	Invalid slot number

System-Based Installation and Removal Routines

You can use the functions `VInstall` and `VRemove` to install task records in and remove them from the system-based vertical retrace queue. These routines exist to provide compatibility with Macintosh computers that have built-in monitors. You can also use these routines when you don't need to synchronize the execution of your VBL task to any monitor.

VInstall

You can use the `VInstall` function to install a task record into the system-based vertical retrace queue.

```
FUNCTION VInstall (vblTaskPtr: QElemPtr): OSErr;
```

`vblTaskPtr`
A pointer to the task record to add to the queue.

DESCRIPTION

The `VInstall` function installs the VBL task record specified by the `vblTaskPtr` parameter in the system-based vertical retrace queue. The Vertical Retrace Manager executes the task at intervals determined by the task record's `vblCount` and `vblPhase` fields. The task must reset the value of the task record's `vblCount` field if you want the task to be executed again.

In current versions of system software, the Vertical Retrace Manager does not continue to execute tasks installed using the `VInstall` function when the application that installed

them is switched out, unless the address in the `vblAddr` field of the task record points in the system partition.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `VInstall` are

Registers on entry

A0 Pointer to the task record

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`vTypErr`	–2	Invalid `qType` value (must be `ORD(vType)`)

VRemove

You can use the `VRemove` function to remove a task record from the system-based vertical retrace queue.

```
FUNCTION VRemove (vblTaskPtr: QElemPtr): OSErr;
```

`vblTaskPtr`
A pointer to the task record to remove from the queue.

DESCRIPTION

The `VRemove` function removes the task record specified by the `vblTaskPtr` parameter from the system-based vertical retrace queue.

To disable a system-based VBL task temporarily, you can set the `vblCount` field of the task record to 0.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `VRemove` are

Registers on entry

A0 Pointer to the task record

Registers on exit

D0 Result code

RESULT CODES

noErr	0	No error
qErr	–1	Task record isn't in the queue
vTypErr	–2	Invalid qType value (must be ORD(vType))

Utility Routines

The Vertical Retrace Manager provides several utility routines that allow you to change the slot number of the primary video monitor, execute all tasks in a slot-based vertical retrace queue, and access the head of the system-based vertical retrace queue.

Note

Most applications do not need to use the routines described in this section. u

AttachVBL

The AttachVBL function changes the slot number of the primary video monitor.

```
FUNCTION AttachVBL (theSlot: Integer): OSErr;
```

theSlot The new slot number for the primary video monitor.

DESCRIPTION

The AttachVBL function changes the slot number of the primary monitor to the number specified by the theSlot parameter. System software uses this routine to ensure correct cursor updating.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for AttachVBL are

Registers on entry

D0 Slot number

Registers on exit

D0 Result code

RESULT CODES

noErr	0	No error
slotNumErr	–360	Invalid slot number

DoVBLTask

Slot interrupt handlers for video cards should call the `DoVBLTask` function to handle the execution of VBL tasks.

```
FUNCTION DoVBLTask (theSlot: Integer): OSErr;
```

`theSlot` Slot number corresponding to the vertical retrace queue whose tasks are to be executed.

DESCRIPTION

The `DoVBLTask` function decrements the `vblCount` field of each task in the vertical retrace queue corresponding to the `theSlot` parameter (except for tasks whose `vblCount` field already contains the value 0). The function executes a task if decrementing the `vblCount` field for a task record results in a value of 0.

If `theSlot` designates the slot of the primary video device, the position of the cursor is also updated.

Slot interrupt handlers for video cards need to call this function to execute any tasks in the queue for that slot. You can also call this function if you need to simulate vertical retrace interrupts.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `DoVBLTask` are

Registers on entry

D0 Slot number

Registers on exit

D0 Result code

To reduce overhead at interrupt time, instead of executing the `_DoVBLTask` trap, you can load the jump vector `jDoVBLTask` into an address register and execute a `JSR` instruction using that register.

RESULT CODES

`noErr`	0	No error
`slotNumErr`	–360	Invalid slot number

GetVBLQHdr

You can obtain the header of the system-based vertical retrace queue by calling the `GetVBLQHdr` function.

```
FUNCTION GetVBLQHdr: QHdrPtr;
```

DESCRIPTION

The `GetVBLQHdr` function returns a pointer to the header of the system-based vertical retrace queue. In general, you need to call this function only if you want to manipulate the contents of the system-based vertical retrace queue directly or if you want to read the information stored in the queue header.

ASSEMBLY-LANGUAGE INFORMATION

The global variable `VBLQueue` contains the header of the system-based vertical retrace queue.

The global variable `ScrnVBLPtr` contains a pointer to the header of the vertical retrace queue associated with the slot for the primary monitor.

Application-Defined Routine

The Vertical Retrace Manager allows your software to install an application-defined routine that is executed during vertical retrace interrupts.

VBL Tasks

You pass the address of an application-defined VBL task in the `vblAddr` field of the VBL task record.

MyVBLTask

A VBL task has the following syntax:

```
PROCEDURE MyVBLTask;
```

DESCRIPTION

The `vblAddr` field of a VBL task record contains the address of a VBL task that is executed after the number of interrupts specified in the `vblCount` field of the task record. The task can be set to execute at any frequency (up to once per vertical retrace interrupt). If the task uses application global variables or calls routines in another code

segment, it must ensure that register A5 contains the address of the boundary between the application global variables and the application parameters. In addition, if your task calls routines in another code segment, that segment must already be loaded in memory.

Because of the optimizations performed by some compilers, the actual work of the VBL task and the setting and restoring of the A5 register might have to be placed in separate procedures. See Listing 4-9 and Listing 4-10 for an example of how you can do this.

Your VBL tasks shouldn't call `VRemove` or `SlotVRemove` to remove its entry from the queue. Instead, either your application should call one of those functions at noninterrupt time or your task should simply not reset the `vblCount` of the task record.

SPECIAL CONSIDERATIONS

Because a VBL task is executed at interrupt time, it should not allocate, move, or purge memory (either directly or indirectly) and should not depend on the validity of handles to unlocked blocks.

The code of the VBL task and any data accessed during its execution must be locked into physical memory if virtual memory is in operation.

Unless directed to do otherwise, some compilers insert code into your compiled application to facilitate debugging operations. This additional code can, however, cause trouble for VBL tasks and other interrupt processing. You might need to disable the generation of debugging code by enclosing the interrupt code between the appropriate compiler directives. Here's an example:

```
{$PUSH}
{$D-}
{Don't generate debugging code for this procedure.}
PROCEDURE DoVBL;
BEGIN
   ...
END;
{$POP}
```

Consult the documentation for your development system to see whether this is necessary and, if it is, how to do it.

ASSEMBLY-LANGUAGE INFORMATION

When the VBL task is called, register A0 contains a pointer to the VBL task record associated with that procedure.

A VBL task must preserve all registers other than A0–A3 and D0–D3. It must exit with an `RTS` instruction.

SEE ALSO

See the section “Accessing Application Global Variables in a VBL Task” beginning on page 4-13 for instructions on how to access your application’s global variables in a VBL task.

Summary of the Vertical Retrace Manager

Pascal Summary

Data Type

```
TYPE VBLTask   =              {VBL queue element}
   RECORD
      qLink:      QElemPtr;   {next entry in vertical retrace queue}
      qType:      Integer;    {queue type}
      vblAddr:    ProcPtr;    {pointer to task procedure}
      vblCount:   Integer;    {interrupts until next execution}
      vblPhase:   Integer;    {task phase}
   END;
```

Vertical Retrace Manager Routines

Slot-Based Installation and Removal Routines

```
FUNCTION SlotVInstall       (vblTaskPtr: QElemPtr; theSlot: Integer): OSErr;
FUNCTION SlotVRemove        (vblTaskPtr: QElemPtr; theSlot: Integer): OSErr;
```

System-Based Installation and Removal Routines

```
FUNCTION VInstall           (vblTaskPtr: QElemPtr): OSErr;
FUNCTION VRemove            (vblTaskPtr: QElemPtr): OSErr;
```

Utility Routines

```
FUNCTION AttachVBL          (theSlot: Integer): OSErr;
FUNCTION DoVBLTask          (theSlot: Integer): OSErr;
FUNCTION GetVBLQHdr         : QHdrPtr;
```

Application-Defined Routine

```
PROCEDURE MyVBLTask;
```

C Summary

Data Types

```
typedef pascal void (*VBLProcPtr)(void);

typedef struct {              /*VBL queue element*/
      QElemPtr    qLink;      /*next entry in vertical retrace queue*/
      short       qType;      /*queue type*/
      VBLProcPtr  vblAddr;    /*pointer to task procedure*/
      short       vblCount;   /*interrupts until next execution*/
      short       vblPhase;   /*task phase*/
} VBLTask;
```

Vertical Retrace Manager Routines

Slot-Based Installation and Removal Routines

```
pascal OSErr SlotVInstall   (QElemPtr vblTaskPtr, short theSlot);
pascal OSErr SlotVRemove    (QElemPtr vblTaskPtr, short theSlot);
```

System-Based Installation and Removal Routines

```
pascal OSErr VInstall       (QElemPtr vblTaskPtr);
pascal OSErr VRemove        (QElemPtr vblTaskPtr);
```

Utility Routines

```
pascal OSErr AttachVBL      (short theSlot);
pascal OSErr DoVBLTask      (short theSlot);
#define GetVBLQHdr()        ((QHdrPtr) 0x0160)
```

Application-Defined Routine

```
pascal void MyVBLTask       (void);
```

Assembly-Language Summary

Constants

```
vType           EQU     1       ;VBL queue element type
inVBL           EQU     6       ;bit index for VBL active flag
```

Data Structures

VBL Queue Element

0	vblink	long	next entry in vertical retrace queue
4	vblType	word	queue type
6	vblAddr	long	address of task procedure
10	vblCount	word	interrupts until next execution
12	vblPhase	word	phase count

Global Variables

CrsrBusy	byte	Set to TRUE if the cursor is being changed.
jDoVBLTask	long	Jump vector for DoVBLTask routine.
ScrnVBLPtr	long	Pointer to the primary monitor's vertical retrace queue's header.
VBLQueue	10 bytes	Header of the vertical retrace queue.

Result Codes

noErr	0	No error
qErr	–1	Task entry isn't in the queue
vTypErr	–2	Invalid qType value (must be ORD(vType))
slotNumErr	–360	Invalid slot number

CHAPTER 5

Notification Manager

Contents

This chapter describes how you can use the Notification Manager to inform users of significant occurrences in applications that are running in the background or in software that is largely invisible to the user. This software includes device drivers, vertical blanking (VBL) tasks, Time Manager tasks, completion routines, and desk accessories that operate behind the scenes. It also includes code that executes during the system startup sequence, such as code contained in `'INIT'` resources.

The Notification Manager is available in system software versions 6.0 and later. You can use the `Gestalt` function to determine whether the Notification Manager is present. See the chapter "Gestalt Manager" in *Inside Macintosh: Operating System Utilities* for complete details on using `Gestalt`.

You need to read this chapter if your application, desk accessory, or device driver might need to notify the user of some occurrence while it is running in the background or is otherwise invisible to the user. You also need to read this chapter if you want to write `'INIT'` resources that might need to inform the user of important occurrences during their execution at system startup time.

About the Notification Manager

The Notification Manager provides a notification service. It allows software running in the background (or otherwise unseen by the user) to communicate information to the user. For example, applications that manage lengthy background tasks (such as printing many documents or transferring large amounts of data to other machines) might need to inform the user that the operation is complete. These applications cannot use the standard methods of communicating with the user, such as alert or dialog boxes, because such windows might easily be obscured by the windows of other applications. Moreover, even if those windows are visible, the background application cannot be certain that the user is aware of the change. A more reliable method is needed to manage the communication between a background application and the user, who might be awaiting the completion of the background task while running some other application in the foreground.

In the same way, relatively invisible operations such as Time Manager tasks, VBL tasks, or device drivers might need to inform the user that some previously started routine is complete or perhaps that some error has rendered further execution undesirable or impossible.

In all these cases, the communication generally needs to occur in one direction only, from the background application (or task, or driver) to the user. The Notification Manager, included in system software versions 6.0 and later, allows you to alert the user by posting a **notification,** which is an audible or visible indication that your application (or other piece of software) requires the user's attention. You post a notification by issuing a **notification request** to the Notification Manager, which places your request in a queue. When your request reaches the top of the queue, the Notification Manager posts a notification to the user.

You can request three types of notification:

- **Polite notification.** A small icon blinks, by periodically alternating with the Apple menu icon (the Apple logo) or the Application menu icon in the menu bar.
- **Audible notification.** The Sound Manager plays the system alert sound or a sound contained in an `'snd '` resource.
- **Alert notification.** An alert box containing a short message appears on the screen. The user must dismiss the alert box (by clicking the OK button) before foreground processing can continue.

These types of notification are not mutually exclusive; for example, an application can request both audible and alert notifications. Moreover, if the requesting software is listed in the Application menu (and hence represents a process that is loaded into memory), you can instruct the Notification Manager to place a diamond-shaped mark next to the name of the requesting process. The mark is usually intended to prompt the user to switch the marked application into the foreground. Finally, you can request that the Notification Manager execute a **notification response procedure,** which is executed as the final step in a notification.

In short, a notification consists of one or more of five possible actions. If you request more than one action, they occur in the following order:

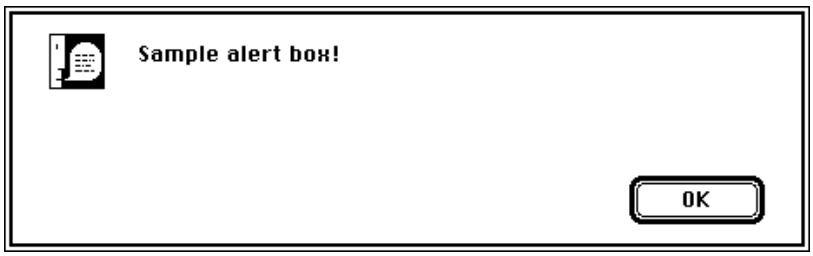
1. A diamond-shaped mark appears next to the name of your application in the Application menu, as illustrated in Figure 5-1. Note that the diamond is present only when your application is in the background (because the diamond is replaced by a checkmark if your application is the active application). In Figure 5-1, the Traffic Light application is the active application.

Figure 5-1 A notification in the Application menu

2. A small icon blinks, alternating with either the Apple menu icon or the Application menu icon in the menu bar. Typically, the small icon is your application's small icon. Because several applications can post notifications, there might be a series of small icons blinking in the menu bar. The location of each blinking icon varies according to the posting application's mark (if any). If your application is marked with a diamond (or a checkmark) in the Application menu, the icon blinks above the Application menu; otherwise, the icon blinks above the Apple menu.

3. The Sound Manager plays a sound. Your application can supply its own sound (by passing the Notification Manager a handle to an `'snd '` resource loaded into memory) or request that the Sound Manager use the user's system alert sound.
4. An alert box like the one in Figure 5-2 appears, and the user dismisses it. Your application specifies the text in the alert box.

Figure 5-2 A notification alert box



5. A response procedure is executed. You can use the response procedure to remove the notification request from the queue or perform other processing.

The mark in the Application menu and the blinking small icon remain until the requesting application removes the notification request from the queue. However, the sound and the alert box are presented only once, if at all.

Any applications, desk accessories, tasks, routines, or drivers can use the Notification Manager, whether they are running in the background or not. It is especially useful for background applications, such as the PrintMonitor application. (The system alarm, which is called by the Alarm Clock desk accessory, also uses the Notification Manager.) Foreground applications can, however, use the Notification Manager to achieve effects (such as the blinking small icon) that are otherwise more difficult to create. For the same reasons, the Notification Manager can be useful even to applications that might be executing in a Finder-only environment under system software version 6.0.

The Notification Manager provides applications with a standard user interface for notifying the user of significant events. The following three-level notification strategy for communicating with the user is recommended:

1. Display a diamond next to the name of the application in the Application menu.
2. Insert a small icon into the list of icons displayed alternately with the Apple menu icon or the Application menu icon in the menu bar, and display a diamond next to the name of your application in the Application menu.
3. Display a diamond, insert a small icon, and display an alert box to notify the user that something needs to be done.

Ideally, the user should be allowed to set the desired level of notification. The suggested default level of notification is level 2. In levels 2 and 3, you might also play a sound, but the user should have the ability to turn the sound off. In addition, a user should have the

ability to turn off background notification altogether, except when damage might occur or data might be lost.

Note
This suggested notification strategy may not be appropriate for your application. Notifications posted by system software might not follow these guidelines. u

Each application, desk accessory, and device driver can issue any number of notification requests. Each requested notification is presented separately to the user. For this reason, avoid posting multiple notification requests for the same occurrence. Depending on the method of notification you specify, multiple requests might result in an annoying number of notification sounds or many alert boxes that the user must dismiss before continuing.

Note that the Notification Manager provides a one-way communications path from an application to the user. There is no provision for carrying information back from the user to the requesting application, although it is possible for the requesting application to determine if the notification was received. If you require this secondary communications link, do not use the Notification Manager. Instead, you should wait until the user switches your application into the foreground and then use standard means (for example, a dialog box) to obtain the required information.

Using the Notification Manager

To issue a notification to the user, you need to create a notification request and install it in the notification queue. The Notification Manager interprets the request and presents the notification to the user at the earliest possible time. After you have notified the user in the desired manner (that is, placed a diamond mark in the Application menu, added a small blinking icon to the menu bar, played a sound, or displayed an alert box), you might want the Notification Manager to call a response procedure. The response procedure is useful for determining that the user has indeed seen the notification or for reacting to the successful posting of the notification. Eventually, you need to remove the notification request from the notification queue; you can do this in the response procedure or when your application returns to the foreground.

The Notification Manager is automatically initialized at system startup time. It includes two functions, one that allows you to install a request into the notification queue and one that allows you to remove a request from that queue.

Creating a Notification Request

Information describing each notification request is contained in the **notification queue,** which is a standard operating-system queue (as described in the chapter “Queue Utilities” in *Inside Macintosh: Operating System Utilities*). Each entry in the notification queue is a **notification record**—a static and nonrelocatable record of type `NMRec`. When

installing a request in the notification queue, your application must supply a pointer to a notification record that indicates the type of notification you desire. Here is the NMRec data structure:

```
TYPE NMRec =
   RECORD
      qLink:      QElemPtr;   {next queue entry}
      qType:      Integer;    {queue type}
      nmFlags:    Integer;    {reserved}
      nmPrivate:  LongInt;    {reserved}
      nmReserved: Integer;    {reserved}
      nmMark:     Integer;    {item to mark in menu}
      nmIcon:     Handle;     {handle to icon}
      nmSound:    Handle;     {handle to sound resource}
      nmStr:      StringPtr;  {string to appear in alert box}
      nmResp:     ProcPtr;    {pointer to response procedure}
      nmRefCon:   LongInt;    {for application's use}
   END;
```

To set up a notification request, you need to fill in the fields qType, nmMark, nmIcon, nmSound, nmStr, nmResp, and nmRefCon. The remaining fields of this record are used internally by the Notification Manager or are reserved for use by Apple Computer, Inc.

Field descriptions

qLink	Points to the next element in the queue. This field is used internally by the Notification Manager.
qType	Indicates the type of queue. You should set this field to the value ORD(nmType), which is 8.
nmFlags	Reserved for use by Apple Computer, Inc.
nmPrivate	Reserved for use by Apple Computer, Inc.
nmReserved	Reserved for use by Apple Computer, Inc.
nmMark	Indicates whether to place a diamond-shaped mark next to the name of the application in the Application menu. If the value of nmMark is 0, no such mark appears. If the value of nmMark is 1, the mark appears next to the name of the calling application. If the value of nmMark is neither 0 nor 1, it is interpreted as the reference number of a desk accessory. An application should pass 1, a desk accessory should pass its own reference number, and a driver or a detached background task (such as a VBL task or Time Manager task) should pass 0.
nmIcon	Contains a handle to a small icon that is to blink periodically in the menu bar. If the value of nmIcon is NIL, no icon appears in the menu bar. This handle must be valid at the time that the notification occurs; it does not need to be locked, but it must be nonpurgeable.
nmSound	Contains a handle to a sound resource to be played with SndPlay. If the value of nmSound is NIL, no sound is produced. If the value

	of nmSound is –1, then the system alert sound plays. This handle does not need to be locked, but it must be nonpurgeable.
nmStr	Points to a string that appears in the alert box. If the value of nmStr is NIL, no alert box appears. Because the Notification Manager does not make a copy of this string, your application should not release this memory until it removes the notification request.
nmResp	Points to a response procedure. If the value of nmResp is NIL, no response procedure is executed when the notification is posted. If the value of nmResp is –1, then a predefined procedure removes the notification request immediately after it has completed.
nmRefCon	A long integer available for your application's own use.

Listing 5-1 illustrates how to set up a notification record. In this listing, gMyNotification is a global variable of type NMRec and gText is a global variable of type Str255.

Listing 5-1 Setting up a notification record

```
VAR
   myResNum:          Integer;    {resource ID of small icon}
   myResHand:         Handle;     {handle to small icon resource}
BEGIN
   myResNum := 1234;              {resource ID in resource fork}
   myResHand := GetResource('SICN', myResNum);
                                  {get icon from resource fork}
   gText := 'Sample Alert Box';   {set message for alert box}
   WITH gMyNotification DO
   BEGIN
      qType := ORD(nmType);       {set queue type}
      nmMark := 1;                {put mark in Application menu}
      nmIcon := myResHand;        {blinking icon}
      nmSound := Handle(-1);      {play system alert sound}
      nmStr := @gText;            {display alert box}
      nmResp := NIL;              {no response procedure}
      nmRefCon := 0;              {not needed here}
   END;
END;
```

This notification record requests all three types of notification—polite (blinking small icon), audible (system alert sound), and alert (alert box). In addition, the diamond appears in front of the application's name in the Application menu. In this case, the small icon has resource ID 1234 of type 'SICN' in the application's resource fork.

Defining a Response Procedure

The nmResp field of the notification record contains the address of a response procedure executed as the final stage of a notification. If no processing is necessary in response to the notification, then you can supply the value NIL in that field. If you supply the address of your own response procedure in the nmResp field, the Notification Manager passes it one parameter, a pointer to your notification record. For example, this is how you would declare a response procedure having the name MyResponse:

```
PROCEDURE MyResponse (nmReqPtr: NMRecPtr);
```

When the Notification Manager calls this response procedure, it does not set up the A5 register or application-specific system global variables for you. If you need to access your application's global variables, you should save its A5 value in the nmRefCon field. See the chapter "Memory Management Utilities" in the book *Inside Macintosh: Memory* for more information on saving and restoring the A5 world.

Response procedures should never cause anything to be drawn on the screen or otherwise affect the human interface. Rather, you should use them simply to remove notification requests from the notification queue and free any memory. If you specify the special nmResp value of –1, the Notification Manager removes the queue element from the queue automatically, and you don't have to do it yourself. You have to pass your own response routine, however, if you need to do anything else in the response procedure, such as free the memory block containing the queue element or set an application global variable indicating that the notification was received.

If you use audible or alert notifications, you should probably set nmResp to –1 to remove the notification record from the queue as soon as the sound ends or the user dismisses the alert box. However, if either nmMark or nmIcon has a nonzero value, you should not set nmResp to –1 (because the Notification Manager would remove the diamond mark or the small icon before the user could see it). Note that when the value of nmResp is –1, the Notification Manager does not free the memory block containing the queue element; it merely removes that element from the notification queue.

Because the execution of the response procedure is the last step in the notification process, your application can determine whether the notification was posted by examining a global variable that you set in the response procedure. In addition, to determine that the user has actually received the notification, you need to request an alert notification. This is necessary because the response procedure is executed only after the user clicks the OK button in the alert box.

Installing a Notification Request

To add a notification request to the notification queue, call the NMInstall function. For example, you can install the notification request defined in Listing 5-1 with the following line of code:

```
myErr := NMInstall(@gMyNotification);     {install request}
```

If the call to `NMInstall` returns an error, then you cannot install the notification request in the notification queue. In that case, your application should wait for the user to switch it to the foreground before doing further processing. While waiting for a resume event, your application should take care of other events, such as updates. Note, however, that `NMInstall` fails only if it is passed invalid information, namely, the wrong value for `qType`.

You can install notification requests at any time, even when the system is executing `'INIT'` resources as part of the system startup sequence. If you need to notify the user of some important occurrence during the execution of your `'INIT'` resource, use the Notification Manager to install a request in the notification queue. The system notifies the user after the startup process completes, that is, when the normal event mechanism begins. This saves you from having to interrupt the system startup sequence with dialog or alert boxes and results in a cleaner and more uniform startup appearance.

Removing a Notification Request

To remove a notification request from the notification queue, call the `NMRemove` function. For example, you can remove a notification request with this code:

```
myErr := NMRemove(@gMyNotification);      {remove request}
```

You can remove requests at any time, either before or after the notification actually occurs. Note that requests already issued by the Notification Manager are not automatically removed from the queue.

Notification Manager Reference

This section describes the routines that are specific to the Notification Manager. It also describes the application-defined notification response procedure.

Notification Manager Routines

The Notification Manager includes two functions, one to install a notification request and one to remove it.

NMInstall

To install a notification request, use the `NMInstall` function.

```
FUNCTION NMInstall (nmReqPtr: NMRecPtr): OSErr;
```

`nmReqPtr` A pointer to a notification record.

DESCRIPTION

The `NMInstall` function adds the notification request specified by the `nmReqPtr` parameter to the notification queue and returns a result code.

SPECIAL CONSIDERATIONS

Because `NMInstall` does not move or purge memory, you can call it from completion routines or interrupt handlers as well as from the main body of an application and from the response procedure of a notification request.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `NMInstall` are

Registers on entry

A0 Address of `NMRec` record

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`nmTypErr`	–299	Invalid `qType` value (must be `ORD(nmType)`)

NMRemove

To remove a notification request, use the `NMRemove` function.

```
FUNCTION NMRemove (nmReqPtr: NMRecPtr): OSErr;
```

`nmReqPtr` A pointer to a notification record.

DESCRIPTION

The `NMRemove` function removes the notification request identified by the `nmReqPtr` parameter from the notification queue and returns a result code.

SPECIAL CONSIDERATIONS

Because `NMRemove` does not move or purge memory, you can call it from completion routines or interrupt handlers as well as from the main body of an application and from the response procedure of a notification request.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `NMRemove` are

Registers on entry

A0	Address of `NMRec` record

Registers on exit

D0	Result code

RESULT CODES

`noErr`	0	No error
`qErr`	–1	Not in queue
`nmTypErr`	–299	Invalid `qType` (must be `ORD(nmType)`)

Application-Defined Routine

The Notification Manager allows you to define a notification response procedure.

Notification Response Procedures

You pass the address of an application-defined notification response procedure in the `nmResp` field of a notification record.

MyResponse

If desired, you can specify the address of a completion or response procedure that is executed as the last stage in a notification. The response procedure should have this syntax:

```
PROCEDURE MyResponse (nmReqPtr: NMRecPtr);
```

`nmReqPtr` A pointer to a notification record.

DESCRIPTION

The `nmResp` field of the notification record contains the address of a response procedure executed as the final stage of a notification. If no processing is necessary in response to the notification, then you can supply the value `NIL` in that field. If you supply the address of your own response procedure in the `nmResp` field, the Notification Manager passes it one parameter, a pointer to your notification record.

SEE ALSO

For more details on a response procedure, see “Defining a Response Procedure” on page 5-9.

Summary of the Notification Manager

Pascal Summary

Constant

```
CONST
   nmType            = 8;              {queue type of notification queue}
```

Data Types

```
TYPE NMRec =
   RECORD
      qLink:         QElemPtr;         {next queue entry}
      qType:         Integer;          {queue type}
      nmFlags:       Integer;          {reserved}
      nmPrivate:     LongInt;          {reserved}
      nmReserved:    Integer;          {reserved}
      nmMark:        Integer;          {item to mark in menu}
      nmIcon:        Handle;           {handle to icon}
      nmSound:       Handle;           {handle to sound resource}
      nmStr:         StringPtr;        {string to appear in alert box}
      nmResp:        ProcPtr;          {pointer to response procedure}
      nmRefCon:      LongInt;          {for application's use}
   END;

   NMRecPtr = ^NMRec;
```

Notification Manager Routines

```
FUNCTION NMInstall          (nmReqPtr: NMRecPtr): OSErr;
FUNCTION NMRemove           (nmReqPtr: NMRecPtr): OSErr;
```

Application-Defined Routine

```
PROCEDURE MyResponse        (nmReqPtr: NMRecPtr);
```

C Summary

Constant

```
enum {nmType         = 8};            /*queue type of notification queue*/
```

Data Types

```
typedef pascal void (*NMProcPtr)(struct NMRec *);

struct NMRec {
      QElemPtr       qLink;           /*next queue entry*/
      short          qType;           /*queue type*/
      short          nmFlags;         /*reserved*/
      long           nmPrivate;       /*reserved*/
      short          nmReserved;      /*reserved*/
      short          nmMark;          /*item to mark in menu*/
      Handle         nmIcon;          /*handle to icon*/
      Handle         nmSound;         /*handle to sound resource*/
      StringPtr      nmStr;           /*string to appear in alert box*/
      NMProcPtr      nmResp;          /*pointer to response procedure*/
      long           nmRefCon;        /*for application's use*/
};

typedef struct NMRec NMRec;
typedef NMRec *NMRecPtr;
```

Notification Manager Routines

```
pascal OSErr NMInstall      (NMRecPtr nmReqPtr);
pascal OSErr NMRemove       (NMRecPtr nmReqPtr);
```

Application-Defined Routine

```
pascal void MyResponse      (NMRecPtr nmReqPtr);
```

Result Codes

`noErr`	0	No error
`qErr`	–1	Not in queue
`nmTypErr`	–299	Invalid `qType` value (must be `ORD(nmType)`)

CHAPTER 6

Deferred Task Manager

Contents

This chapter describes how your application or device driver can use the Deferred Task Manager to defer the execution of lengthy tasks until interrupts are reenabled. Time-consuming tasks, if executed at interrupt time, can prevent the execution of interrupt tasks having the same or lower priority. The Deferred Task Manager allows you to improve interrupt handling by deferring a task until all other interrupts have been serviced.

Lengthy tasks are often initiated by slot cards. As a result, you probably need to read the information in this chapter only if your application or driver deals with slot-card interrupts. However, you can use the services provided by the Deferred Task Manager whenever you need to install a lengthy interrupt task capable of running with all interrupts enabled. You can, for example, defer the execution of completion routines, Time Manager routines, and VBL tasks.

To use this chapter, you should be familiar with interrupts and interrupt tasks in general. See the chapter "Introduction to Processes and Tasks" in this book for an overview of both interrupt and noninterrupt processing. Because the Deferred Task Manager maintains all deferred tasks in a queue until their execution, you should also be familiar with operating-system queues, as described in the chapter "Queue Utilities" in *Inside Macintosh: Operating System Utilities*.

This chapter begins with a description of interrupt priority levels and explains when you might need to use the Deferred Task Manager. Then it shows how you can use the Deferred Task Manager to defer a task. The chapter concludes with a description of the Deferred Task Manager's data structure and routine.

About the Deferred Task Manager

Every type of interrupt has an **interrupt priority level,** a number that identifies the importance of the interrupt. The microprocessor also maintains several bits in the status register of the CPU that indicate which interrupts are currently to be processed and which are to be ignored. This **processor priority** is always set to the interrupt priority level of the highest-priority interrupt currently executing. For example, if no interrupts are being serviced, the processor priority is 0. If the current application is then interrupted by a vertical retrace interrupt, the interrupt priority is set to 1 during the servicing of the interrupt and restored to 0 upon completion. If, during the servicing of the vertical retrace interrupt, a level-2 interrupt occurs, the processor priority is set to 2 during the servicing of the interrupt and restored to 1 upon completion of any level-2 interrupt tasks.

The microprocessor ordinarily services an interrupt only if its interrupt priority level is higher than the processor priority. Accordingly, when no interrupt routines are executing, the microprocessor can service any new interrupt. If, however, a slot interrupt is executing, the microprocessor ignores other slot interrupts and interrupts of lower priority. As a result, a lower-priority interrupt (for example, a vertical retrace interrupt) might not execute on schedule.

When the microprocessor is servicing one interrupt, it is said to **disable** other interrupts whose priority level is lower than or the same as that of the interrupt being serviced. This feature prevents the interruption of tasks by interrupts of lesser or equal priority. You might, however, initiate an interrupt task that does not need this extra protection. If an interrupt task takes so much time to execute that the disabling of other interrupts during execution becomes significant, you might prefer it to have your interrupt task executed at a time when all other interrupt tasks have been serviced and interrupts are reenabled. The Deferred Task Manager provides a mechanism for this purpose.

Instead of immediately performing the main work of a task, such as a slot-interrupt task, you can **defer** the task, or schedule it for execution when all interrupts have been reenabled. You do this by placing information about the task to be deferred in a **deferred task record,** which you then insert in the **deferred task queue.** The task is then known as a **deferred task.** All system interrupt handlers check the deferred task queue just before returning. If there are tasks in the queue and the microprocessor's status register is about to be reset to 0, the system interrupt handlers reenable interrupts and pass control to the Deferred Task Manager to execute all the deferred tasks.

The Deferred Task Manager checks whether a VBL task is active. If so, the Deferred Task Manager exits, and the deferred tasks remain deferred until the VBL task completes. (The VBL task is interrupt code, and so the Deferred Task Manager is called again when the Vertical Retrace Manager returns control to the primary interrupt handler.) If a VBL task is not active, the Deferred Task Manager checks whether a deferred task is already active. If so, the Deferred Task Manager exits. Otherwise, a deferred task is removed from the queue and executed. When all deferred tasks have been removed from the queue and executed, the Deferred Task Manager returns control to the primary interrupt handler.

Each interrupt task is removed from the deferred task queue before it is executed. For this reason, your interrupt code must reinstall the task record into the queue each time the task is to be deferred. If your task is simple enough that reinstalling the task record into the deferred task queue takes about as much time as doing the real work of the task, then the Deferred Task Manager is not useful for your application. Note that interrupts are disabled during the reinstallation of a task record into the deferred task queue, even though they are reenabled before the reinstalled task is executed.

Although you can use the Deferred Task Manager for all types of interrupt tasks, it is especially convenient for slot-interrupt tasks. Interrupts from NuBus™ slot devices are received and decoded by special hardware on the main logic board. This hardware generates level-2 interrupts. Because of the way the hardware works, the microprocessor must disable lower-priority interrupts until it services the level-2 interrupts (otherwise, a system error occurs). During the execution of slot-interrupt tasks, the microprocessor disables other level-2 interrupts, such as those for sound, as well as all level-1 interrupts. By using the Deferred Task Manager, you can defer the processing of slot interrupts until all of the slots are scanned. Just before returning, the slot-interrupt handler executes any tasks having records in the deferred task queue.

It is important to remember that deferred tasks are executed at the end of a hardware interrupt cycle, before the secondary interrupt handler returns. In addition, the tasks in the deferred task queue are executed only if the status register is being restored to 0 (that

is, all interrupts reenabled). If the status register is not being restored to 0, but only to some higher level, deferred tasks are not executed during that hardware interrupt cycle.

This behavior, if not properly understood, can lead to some puzzling situations. For example, applications can mask the CPU's status register to disable certain interrupts. Suppose that your application installs and activates a Time Manager task, which is triggered by level-2 interrupts. If you don't want the task to be executed during a specific period of time, you can set the status register to 2, thus disabling all level-1 and level-2 interrupts. (In this case, the status register is set to 2, but not in response to a level-2 interrupt.)

Now suppose that a level-4 interrupt occurs, perhaps triggered by the arrival of some LocalTalk data at a serial port. The LocalTalk interrupt handler is executed with the status register set to 4. That handler might install a deferred task and then return. Because the interrupt cycle is nearly complete, the system interrupt handler checks whether the status register is about to be restored to 0. In the situation described, the status register is about to be restored to 2, not to 0. As a result, any pending deferred tasks, including the newly installed LocalTalk deferred task, are ignored. Moreover, if the status register remains masked at 2, any additional deferred tasks installed by the LocalTalk interrupt handler remains queued and are not executed.

Eventually, the application that masked the status register (to disable its Time Manager task) will restore the status register to 0. At the end of the next hardware interrupt cycle, all the pending deferred tasks are finally executed.

As you can see, it's possible for an interrupt routine—in this example, the LocalTalk interrupt handler—to install a deferred task that is not executed until after some future hardware interrupt cycle. Indeed, that future hardware interrupt might well be another LocalTalk interrupt. In other words, it's possible for an interrupt routine to install a deferred task and to be called again, *before* the deferred task has been executed. It's even possible for the interrupt routine to interrupt the deferred task that it installed during some previous interrupt cycle. You need to make sure, for instance, that your interrupt code doesn't modify a data buffer that a deferred task is processing.

Keep these points in mind when you use the Deferred Task Manager to defer tasks:

- The purpose of the Deferred Task Manager is to allow lengthy interrupt tasks to be deferred until all interrupts can be reenabled.
- Deferred tasks are executed with all interrupts enabled (that is, with the status register set to 0).
- Deferred tasks are not executed if some other interrupt code is executing. For example, a deferred task will not interrupt a VBL task.
- A deferred task is not executed if some other deferred task is being executed. A deferred task cannot interrupt another deferred task.
- Deferred tasks can be interrupted.
- Deferred tasks are executed within the hardware interrupt cycle, even though the status register is set to 0 before the tasks are executed. As a result, deferred tasks are subject to all the normal limitations on interrupt-level code. In particular, deferred tasks cannot call any routine that directly or indirectly allocates or moves memory,

and cannot depend on the validity of unlocked handles. See the chapter "Introduction to Processes and Tasks" in this book for a complete description of these limitations.

- Deferred tasks are not prioritized. They are executed in the order they were added to the deferred task queue, no matter what interrupt level the code that installed them was running at.

Using the Deferred Task Manager

You can use the Deferred Task Manager to defer the execution of some code that is to be executed as a result of an interrupt. This section shows how to install a deferred task and how to use a high-level language to access the optional parameter passed to your task in register A1. Because the Deferred Task Manager is not available in all operating environments, you need to check that it is available before using it. The following section shows how to do this.

Checking for the Deferred Task Manager

The Deferred Task Manager was introduced primarily to allow slot handlers to defer lengthy processing initiated by a slot interrupt and, until system software version 7.0, was not available on all computers running the Macintosh Operating System. For example, the Deferred Task Manager is not available on Macintosh Plus or Macintosh SE computers running system software version 6.0. In addition, there is no support for the Deferred Task Manager in versions of A/UX earlier than version 3.0.

As a result, you should always make sure that the Deferred Task Manager is available in the current operating environment before attempting to use it. You can use the function `DeferredTasksAvailable`, defined in Listing 6-1, to do this.

Listing 6-1 Checking for the availability of the Deferred Task Manager

```
FUNCTION DeferredTasksAvailable: Boolean;
CONST
   _DTInstall = $A082;
BEGIN
   DeferredTasksAvailable := TrapAvailable(_DTInstall);
END;
```

The `DeferredTasksAvailable` function simply calls the function `TrapAvailable` to determine whether the trap `_DTInstall` is implemented. See the chapter "Gestalt Manager" in *Inside Macintosh: Operating System Utilities* for a definition of the `TrapAvailable` function.

System software versions 7.0 and later support the Deferred Task Manager on all Macintosh computers, including the Macintosh Plus and Macintosh SE. However, the

system global variables DTQueue (containing the address of the deferred task queue header) and jDTInstall (containing the jump vector for the DTInstall function) are not supported on the Macintosh Plus. You should not use DTQueue or jDTInstall on the Macintosh Plus.

Installing a Deferred Task

The Deferred Task Manager provides a single routine, DTInstall, that you can use to install elements into the deferred task queue. The deferred task queue is a standard operating-system queue whose elements are defined by the DeferredTask data type.

```
TYPE DeferredTask =
RECORD
   qLink:      QElemPtr;   {next queue entry}
   qType:      Integer;    {queue type}
   dtFlags:    Integer;    {reserved}
   dtAddr:     ProcPtr;    {pointer to task}
   dtParm:     LongInt;    {optional parameter passed in A1}
   dtReserved: LongInt;    {reserved; should be 0}
END;
```

Your application or driver needs to fill in only the qType, dtAddr, and dtReserved fields. The dtAddr field specifies the address of the routine whose execution you want to defer. You can also specify a value for the dtParm field, which contains an optional parameter that is loaded into register A1 just before the routine specified by the dtAddr field is executed. The dtFlags and dtReserved fields of the deferred task record are reserved. You should set the dtReserved field to 0.

Listing 6-2 defines a routine, InstallDeferredTask, for installing a task element in the deferred task queue. This element corresponds to the routine MyDeferredTask, which does the real work of your interrupt task. The InstallDeferredTask routine sets up a deferred task record and then installs it in the deferred task queue by calling the DTInstall function. Note that you should call DTInstall only at interrupt time.

Listing 6-2 Installing a task into the deferred task queue

```
PROCEDURE InstallDeferredTask (theTask: DeferredTask);
VAR
   myErr:   OSErr;
BEGIN
   WITH theTask DO
   BEGIN
      qType := ORD(dtQType);     {set the queue type}
      dtAddr := @MyDeferredTask; {set address of deferred task}
      dtParm := 0;               {no parameter needed here}
```

```
      dtReserved := 0;           {clear reserved field}
   END;
   myErr := DTInstall(@theTask);
END;
```

Defining a Deferred Task

You define a deferred task as a procedure taking no parameters and put the address of that procedure in the deferred task element whose address you pass to the `DTInstall` function. When your task is executed, register A1 contains the optional parameter that you put in the `dtParm` field of the task record.

If you write your deferred task in a high-level language, such as Pascal, you might need to retrieve the value loaded into register A1. The function `GetA1` defined in Listing 6-3 returns the value of the A1 register.

Listing 6-3 Finding the value of the A1 register

```
FUNCTION GetA1: LongInt;
INLINE
   $2E89;         {MOVE.L A1,(SP)}
```

You can call `GetA1` in your deferred task, as illustrated in Listing 6-4.

Listing 6-4 Defining a deferred task

```
PROCEDURE DoDeferredTask (dtParm: LongInt);
BEGIN
   {Your deferred task code goes here.}
END;

PROCEDURE MyDeferredTask;
VAR
   myParm:     LongInt;
BEGIN
   myParm := GetA1;        {retrieve parameter put in register A1}
   DoDeferredTask(myParm); {run the deferred task}
END;
```

Note that `MyDeferredTask` calls `GetA1` to retrieve the parameter passed in the register A1. Then `MyDeferredTask` calls the application-defined procedure `DoDeferredTask`, passing it that parameter. The `DoDeferredTask` procedure does the real work of the deferred task. (This division into two routines is necessary to prevent problems caused by some optimizing compilers.)

Deferring a Slot-Based VBL Task

As indicated earlier in this chapter, you are most likely to use the Deferred Task Manager when dealing with slot interrupts. All slot interrupts, including slot-based VBL interrupts, disable all other slot interrupts. For this reason, as a slot-interrupt routine (installed using `SIntInstall`) or a slot-based VBL interrupt routine (installed using `SlotVInstall`) runs to completion, interrupts at that level and below are disabled. You can help improve interrupt handling by using the Deferred Task Manager to defer your slot-interrupt processing until interrupts have been reenabled.

Listing 6-5 provides another example of how to use the Deferred Task Manager. The program defined there defers the cursor updating that would normally occur as a slot-based VBL task. The time required to update the cursor can range from about 700 to 900 microseconds for monitors having a screen depth of 1 to 8 bits. Because the cursor updating is done at slot-based VBL time, all other slot interrupts are put off until updating is finished. This might adversely affect interrupt processing by your application. Accordingly, it is useful to defer the cursor updating to noninterrupt time by installing the updating as a deferred task.

The program defined in Listing 6-5 replaces the cursor-updating routine pointed at by the system global variable `jCrsrTask` with a different routine. This new routine installs the original routine as a deferred task.

Listing 6-5 Deferring cursor updating to noninterrupt time

```
*** MyDefTask
TaskBegin
MyDefTask
   DC.L        0                ;qLink   (handled by OS)
   DC.W        0                ;qType   (queue type: dtQType)
   DC.W        0                ;dtFlags (reserved)
   DC.L        0                ;dtAddr  (pointer to routine to be executed)
   DC.L        0                ;dtParm  (optional parameter; not used here)
   DC.L        0                ;dtReserved (should be zero)
SysCrsrTask
   DC.L        0                ;pointer to system jCrsrTask
DefCrsrFlag
   DC.W        0                ;1 if using a deferred task, 0 otherwise
PendingFlag
   DC.W        0                ;1 if a jCrsrTask is pending, 0 otherwise

*** MyjCrsrTask
MyjCrsrTask
   MOVEM.L     A0/A1/D0,-(SP)
   LEA         PendingFlag,A0   ;see if a deferred jCrsrTask task is pending
   TST.W       (A0)
```

```
   BNE.S       bailOut           ;if yes, exit
   MOVE.W      #1,(A0)           ;if no, set the pending flag
   LEA         MyDefTask,A0      ;point to our deferred task element
   LEA         DefjCrsrTask,A1   ;get address of deferred task routine
   MOVE.L      A1,dtAddr(A0)     ;set up pointer to routine
   MOVE.W      #dtQType,dtType(A0)     ;set queue type
   _DTInstall                    ;install the task
   MOVEM.L     (SP)+,A0/A1/D0
   RTS
bailOut
   MOVEM.L     (SP)+,A0/A1/D0
   RTS
DefjCrsrTask
   MOVEM.L     A0,-(SP)
   LEA         SysCrsrTask,A0    ;get system cursor task address
   MOVEA.L     (A0),A0
   JSR         (A0)              ;and call it
   LEA         PendingFlag,A0    ;clear pending call flag
   CLR.W       (A0)
   MOVEM.L     (SP)+,A0
   RTS
TaskEnd

*** Entry
TaskSize       EQU   TaskEnd-TaskBegin

Entry
   MOVE.L      #TaskSize,D0      ;put TaskSize into D0
   _NewPtr     SYS,CLEAR         ;make a block in the system heap
   BNE.S       Quit              ;no room in system heap, so quit
   MOVE.L      0,A2              ;got a good pointer; keep a copy
   MOVE.L      A0,A1             ;set up registers for BlockMove
   LEA         MyDefTask,A0
   MOVE.W      #TaskSize,D0
   _BlockMove                    ;copy the task etc. into system heap
   LEA         dtQElSize(A2),A0  ;move original task pointer into our
   MOVE.L      jCrsrTask,(A0)    ; pointer holder
   LEA         dtQElSize+4(A2),A0;replace jCrsrTask pointer with a pointer
   MOVE.L      A0,jCrsrTask      ; to our jCrsrTask

Quit
   RTS                           ;exit the program
END
```

This code allocates a block of memory in the system heap. The allocated block is large enough to hold a deferred task element, a pointer to the original cursor-updating routine, and the replacement routine. The replacement routine simply retrieves the relevant information (namely, the deferred task element and the saved address of the original cursor-updating routine) stored in that block of memory and calls `_DTInstall` to install a deferred task. The address of the replacement routine is placed into the low-memory global variable `jCrsrTask`, whose original contents are stored in the system heap.

Once the program defined in Listing 6-5 is run, the cursor-updating routine is subsequently performed with interrupts enabled, thereby allowing other interrupts. Because the cursor-updating routine is run with interrupts enabled, you may see a slight flickering of the cursor when using this technique.

Deferred Task Manager Reference

This section summarizes the structure of the deferred task record and describes the `DTInstall` function, which you can use to install a deferred task record into the deferred task queue. It also describes the application-defined deferred task.

Data Structure

The deferred task queue is a standard operating-system queue. The `DeferredTask` data type defines an element in the deferred task queue.

```
TYPE DeferredTask =
RECORD
   qLink:      QElemPtr;   {next queue entry}
   qType:      Integer;    {queue type}
   dtFlags:    Integer;    {reserved}
   dtAddr:     ProcPtr;    {pointer to task}
   dtParm:     LongInt;    {optional parameter passed in A1}
   dtReserved: LongInt;    {reserved; should be 0}
END;
```

Field descriptions

`qLink`	A pointer to the next entry in the deferred task queue, or `NIL` if there are no more entries in the queue. You do not need to set this field; the Deferred Task Manager does it for you.
`qType`	The queue type. You must set this field to `ORD(dtQType)`.
`dtFlags`	Reserved.
`dtAddr`	A pointer to the task to be executed. Set this field to the address of the routine that you want to execute after interrupts have been enabled.

dtParm	An optional parameter that is loaded into register A1 just before the routine specified by the dtAddr field is executed.
dtReserved	Reserved. You should set this field to 0.

Deferred Task Manager Routine

The Deferred Task Manager provides a single routine for installing task records into the deferred task queue, the DTInstall function.

DTInstall

After defining the fields of a deferred task record, you can call the DTInstall function to install the record into the deferred task queue.

```
FUNCTION DTInstall (dtTaskPtr: QElemPtr): OSErr;
```

dtTaskPtr A pointer to a queue element to add to the deferred task queue.

DESCRIPTION

The DTInstall function adds the specified task record to the deferred task queue. Your application should fill in all fields of the task record except qLink and qFlags.

Ordinarily, you call DTInstall only at interrupt time. The DTInstall function does not actually execute the routine specified in the dtAddr field of the task record. Each system interrupt handler executes routines stored in the deferred task queue after reenabling interrupts. After a routine in the queue is executed, it is removed from the deferred task queue.

If the qType field of the task record is not set to ORD(dtQType), DTInstall returns vTypErr and does not add the record to the queue. Otherwise, DTInstall returns noErr.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for DTInstall are

Registers on entry

A0 Pointer to new queue entry

Registers on exit

D0 Result code

To reduce overhead at interrupt time, instead of executing the DTInstall trap, you can load the jump vector jDTInstall into an address register other than A0 and execute a JSR instruction using that register.

RESULT CODES

`noErr`	0	No error
`vTypErr`	–2	Invalid `qType` value (must be `ORD(dtQType)`)

Application-Defined Routine

The Deferred Task Manager allows your interrupt routines to install an application-defined routine whose execution is deferred until after all interrupts are reenabled.

Deferred Tasks

You pass the address of an application-defined deferred task in the `dtAddr` field of a deferred task record.

MyDeferredTask

A deferred task has the following syntax:

```
PROCEDURE MyDeferredTask;
```

DESCRIPTION

The `dtAddr` field of a deferred task record contains the address of a procedure that is executed at the end of a hardware interrupt cycle when all interrupts are reenabled.

SPECIAL CONSIDERATIONS

Because the deferred task is executed during a hardware interrupt cycle, it should not allocate, move, or purge memory (either directly or indirectly) and should not depend on the validity of handles to unlocked blocks.

If a deferred task uses application global variables, it must ensure that register A5 contains the address of the boundary between the application global variables and application parameters. For details, see the discussion of setting up and restoring the A5 register in the chapter "Memory Management Utilities" in *Inside Macintosh: Memory*.

A deferred task should avoid accessing system global variables or calling a trap that would access one.

ASSEMBLY-LANGUAGE INFORMATION

When the deferred task is called, register A1 contains the value of the `dtParm` field in the deferred task record passed to `DTInstall`.

A deferred task must preserve all registers other than A0–A3 and D0–D3.

Summary of the Deferred Task Manager

Pascal Summary

Data Type

```
TYPE DeferredTask =
   RECORD
      qLink:      QElemPtr;         {next queue entry}
      qType:      Integer;          {queue type}
      dtFlags:    Integer;          {reserved}
      dtAddr:     ProcPtr;          {pointer to task}
      dtParm:     LongInt;          {optional parameter passed in A1}
      dtReserved: LongInt;          {reserved; should be 0}
   END;
```

Deferred Task Manager Routine

```
FUNCTION DTInstall          (dtTaskPtr: QElemPtr): OSErr;
```

Application-Defined Routine

```
PROCEDURE MyDeferredTask;
```

C Summary

Data Type

```
struct DeferredTask {
      QElemPtr    qLink;            /*next queue entry*/
      short       qType;            /*queue type*/
      short       dtFlags;          /*reserved*/
      ProcPtr     dtAddr;           /*pointer to task*/
      long        dtParm;           /*optional parameter passed in A1*/
      long        dtReserved;       /*reserved; should be 0*/
};
```

Deferred Task Manager Routine

```
pascal OSErr DTInstall      (QElemPtr dtTaskPtr);
```

Application-Defined Routine

```
pascal void MyDeferredTask  (void);
```

Assembly-Language Summary

Deferred Task Manager Queue Element

0	qLink	long	pointer to next queue entry
4	qType	word	queue type
6	dtFlags	word	reserved
8	dtAddr	long	pointer to task
12	dtParm	long	optional parameter to be passed in A1
16	dtReserved	long	reserved; should be 0

Global Variables

DTQueue	10 bytes	Deferred task queue header.
jDTInstall	long	Jump vector for DTInstall function.

Result Codes

noErr	0	No error
vTypErr	–2	Invalid qType value (must be ORD(dtQType))

CHAPTER 7

Segment Manager

Contents

This chapter describes the Segment Manager, the part of the Macintosh Operating System that loads and unloads your application's code segments into and out of memory. By dividing your application's executable code into segments, you allow it to run in a memory partition that is smaller than the total size of the application itself and the data it is using.

To use this chapter, you should already be familiar with the basic concepts of the Resource Manager and the Memory Manager. You need to know about the basic operation of the Resource Manager because segments are stored as resources. You need to know about the basic operation of the Memory Manager to understand when and why segments might be purged from memory. See the chapter "Introduction to Memory Management" in *Inside Macintosh: Memory*.

You should read this chapter if your application contains multiple code segments that do not all need to be in memory at one time.

About the Segment Manager

Your application's executable code is stored in its resource fork as one or more resources of type `'CODE'`. These code resources are known as **segments** because the division of routines into code resources is controlled by segmentation directives you provide to your development system.

The Process Manager loads some code segments into memory when your application is launched. The Segment Manager loads other segments whenever you call any externally referenced routine contained in those segments. Both of these operations occur completely automatically and rely on information stored in your application's jump table and in the individual code segments themselves.

The Segment Manager loads segments into relocatable, purgeable blocks in your application heap. A segment is locked when it is first read into memory and at any time thereafter when routines in the segment are executing. This locking prevents the block from being moved during heap compaction and from being purged during heap purging.

Although needed code segments are loaded into memory automatically, it is your application's responsibility to unload any segments that are not currently being used. The Segment Manager provides a single procedure, `UnloadSeg`, that you can call to unload a segment. To **unload** a segment is simply to unlock it. By unlocking unneeded segments, you allow them to be relocated or purged if necessary to accommodate a later memory-allocation request. Thus, using the Segment Manager to unload unneeded segments is one important aspect of an efficient memory-management policy.

The following sections describe in detail the reasons for segmenting an application and the structure of the jump table.

Code Segmentation

Your development system's linker divides your application's executable code into segments according to directives that you provide. The **main segment** contains the main program. This segment is loaded into memory when your application starts to run and is never purged or unlocked as long as the application is running. The main event loop and other frequently needed small routines are generally stored in the main segment.

Most applications, however, consist of multiple code segments. There are two principal reasons for dividing code into different segments:

- **Compiler limitations.** Most development systems generate PC-relative instructions for intrasegment references (references to other routines within the same code segment). Because PC-relative instructions on an MC68000 use a 16-bit offset, the offset to the last routine in the segment cannot be larger than 32K bytes. Some development systems therefore restrict the size of any one code segment to 32K bytes.
- **Memory limitations.** Many applications are so large that the entire executable code, together with static data (such as your application's global data and resources) and data created dynamically during the execution of the application (such as windows and the items they contain), simply cannot fit into a memory partition of reasonable size.

By dividing your executable code into segments, you can circumvent both these limitations. The size of your application can increase as required to provide the desired capabilities without necessitating an increased run-time memory partition. For example, code that isn't executed very often (such as code for printing a document) can be put into a separate segment; it's loaded when needed and can be unloaded to free the memory for other uses when it's no longer needed.

Note

Some development systems allow you to create segments that are larger than 32K bytes. Consult your development system's documentation to determine how and when to increase segment size. u

The key fact to keep in mind when deciding how to group routines into segments is that an entire segment is loaded into memory whenever you call one of the routines in the segment. It makes sense, therefore, to group related routines in the same segment. You should segment routines according to your run-time call chain rather than on a simple file-by-file basis.

There are also some less obvious guidelines to follow when grouping routines into segments.

- Put your main event loop into the main segment.
- Put any routines that handle low-memory conditions into a locked segment (commonly the main segment). For example, if your application provides a grow-zone function, put that function into a locked segment.
- Put any routines that execute at interrupt time, including VBL tasks and Time Manager tasks, into a locked segment (commonly the main segment).

- Put into a separate segment any initialization routines that are executed exactly once at application startup time. Then unload that segment after those routines are executed. There is, however, at least one important exception to this rule. Routines that allocate nonrelocatable objects in your application heap should be called in the main segment, before you load any code segments that will later be unloaded. If you put such allocation routines into a code segment that is later unloaded and purged, you increase heap fragmentation. Routines such as `MoreMasters` and `InitWindows`, which are typically called at the beginning of an application, allocate nonrelocatable objects and should therefore be in the main segment.

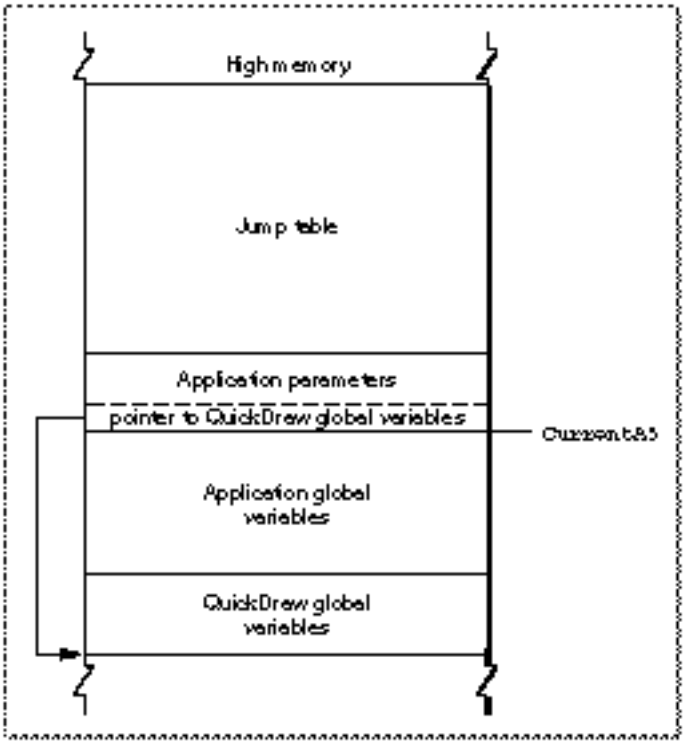
The Jump Table

Note

This section describes how the Segment Manager works internally and is included for informational purposes only. You don't need this information to use the Segment Manager routine. Moreover, the information presented here might not be accurate for your development system. See the note on page 7-7.

The loading and unloading of segments are implemented through your application's **jump table,** an area of memory in your application's partition that contains one entry for every externally referenced routine in every code segment of your application. The location of the jump table is illustrated in Figure 7-1.

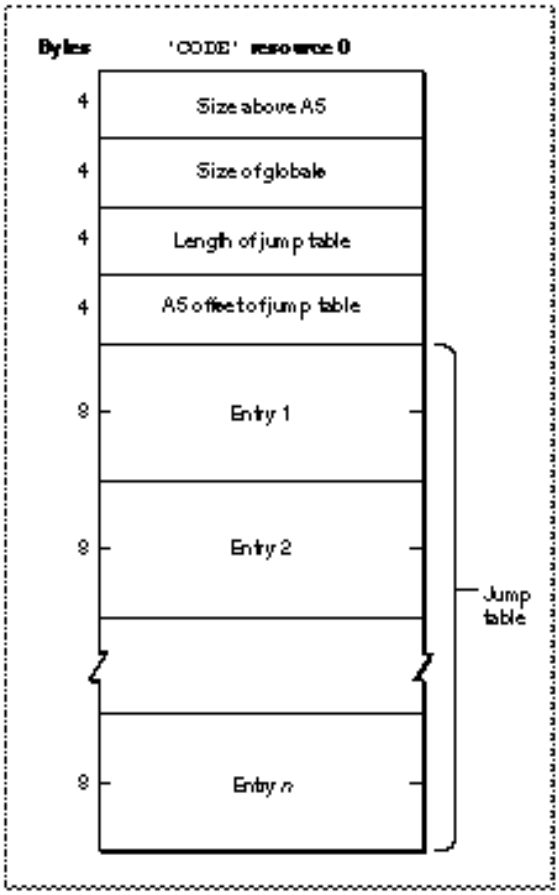
Figure 7-1 The location of the jump table



The jump table is accessed through the A5 register and is therefore part of your application's A5 world.

The jump table is created by your development system's linker and is stored in segment 0 of your application (which is the `'CODE'` resource with an ID of 0). Segment 0 is a special segment created by the linker for every application; it contains information about the A5 world and the jump table. Figure 7-2 illustrates the structure of segment 0.

Figure 7-2 The structure of segment 0



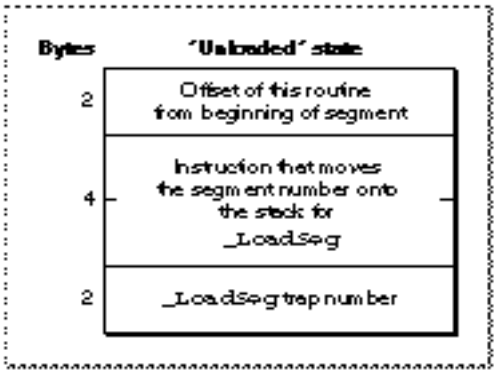
Segment 0 consists of these elements:

- Size above A5. The size (in bytes) from the location pointed to by register A5 to the upper end of the application space.
- Size of globals. The size (in bytes) of the application global variables plus the QuickDraw global variables.
- Length of jump table. The size (in bytes) of the jump table.

- A5 offset of jump table. The offset (in bytes) to the jump table from the location pointed to by register A5. This offset is stored in the global variable `CurJTOffset`.
- Jump table. A contiguous list of jump table entries.

When the MPW linker encounters a call to a routine in another code segment, it creates a **jump table entry** for that routine. (All entries for a particular segment are stored contiguously in the jump table.) The structure of a jump table entry varies according to whether the segment it references is loaded or unloaded. If the segment is not yet loaded into memory, the jump table entry has the structure illustrated in Figure 7-3.

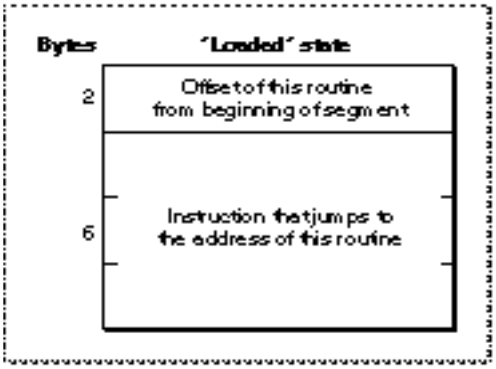
Figure 7-3 Format of an MPW jump table entry when the segment is unloaded



Note
Some development systems use a different format for jump table entries of unloaded routines to circumvent the 32K-byte limitation on the size of segments, global data, or the jump table itself. Consult the documentation for your development system to see whether it uses the jump table entry formats described in this section and whether you can safely call the `UnloadSeg` procedure (which changes jump table entries). u

The jump table refers to segments by segment numbers assigned by the linker. If the segment isn't loaded, the entry contains code that loads the segment. When a segment is unloaded, all its jump table entries are in the "unloaded" state. When a call to a routine in an unloaded segment is made, the code in the last 6 bytes of its jump table entry is executed. This code calls the `_LoadSeg` trap, which loads the segment into memory, transforms all of its jump table entries to a "loaded" state, and invokes the routine by executing the instruction in the last 6 bytes of its jump table entry. Figure 7-4 illustrates the format of a jump table entry in the "loaded" state.

Figure 7-4 Format of an MPW jump table entry when the segment is loaded



Subsequent calls to the routine also execute this instruction. When you call `UnloadSeg`, it restores the jump table entries to their "unloaded" state. Notice that the last 6 bytes of the jump table entry are always executed; the effect depends on the state of the entry at the time.

To set all the jump table entries for a segment to a particular state, the Segment Manager needs to know exactly where in the jump table all the entries are located. It gets this information from the **segment header,** 4 bytes at the beginning of the segment that contain the offset of the first routine's entry from the beginning of the jump table (2 bytes) and the number of entries for the segment (2 bytes).

Using the Segment Manager

The Segment Manager provides one routine for use by applications, the `UnloadSeg` procedure. You use this routine to unload code segments. The Operating System also provides two low-memory global variables that you can use to override the default segment-loading behavior and to monitor the system's automatic loading of code segments.

Unloading Code Segments

You can use the `UnloadSeg` procedure to unload segments. To unload a particular segment, pass `UnloadSeg` the address of any externally referenced routine contained in that segment. For example, to unload the segment that contains the procedure `DoPrintFile`, execute this line of code:

```
UnloadSeg(@DoPrintFile);
```

You can call `UnloadSeg` at any time except when you are executing code contained in the segment to be unloaded. A typical strategy is to unload all code segments except

segment 1 and any other essential code segments each time through your application's main event loop.

s **WARNING**
Before you unload a segment, make sure that your application no longer needs it. Never unload a segment that contains a completion routine or other interrupt task (such as a Time Manager task or VBL task) that might be executed after the segment is unloaded. Never unload a segment that contains routines in the current call chain. s

The `UnloadSeg` procedure does not actually remove the segment from memory. Instead, it unlocks the segment, thereby making the segment relocatable and purgeable. This permits the Memory Manager to relocate or purge the segment if necessary to gain some space in the application heap.

Loading Code Segments

The Segment Manager loads a code segment into memory automatically when you call any externally referenced routine in that segment. In most cases, the Segment Manager moves the block occupied by the code segment as high in the application heap as possible (by calling the Memory Manager procedure `MoveHHi`) and locks the block (by calling `HLock`) so that it cannot be moved or purged. You can disable or enable the call to `MoveHHi` and monitor the loading of segments into memory by manipulating two low-memory global variables.

If a code segment to be loaded is unlocked (that is, if it's not in memory and its `resLocked` attribute is clear, or if it is in memory and is unlocked), then the `_LoadSeg` trap calls the Memory Manager procedure `MoveHHi` to move the segment toward the top of the current heap. To prevent heap fragmentation, you should call the Memory Manager procedure `MaxApplZone` early in your application's execution. Otherwise, the heap will grow incrementally, and these automatic calls to `MoveHHi` may leave your code segments scattered throughout the heap. You can, however, disable the call to `MoveHHi` by setting the low-memory global variable `SegHiEnable` to 0. If this variable contains the value 0, `_LoadSeg` does not call `MoveHHi` to move the segment toward the top of the heap.

Occasionally, especially during application development, it is useful to monitor the otherwise largely invisible process of loading segments. You can do this by manipulating the system global variable `LoadTrap`. Before any routine in a newly loaded code segment is executed, the `_LoadSeg` trap inspects the `LoadTrap` global variable. If `LoadTrap` has a nonzero value, then `_LoadSeg` calls the `_Debugger` trap. This provides a useful way for you to monitor the loading of segments by the Segment Manager.

Segment Manager Reference

This section describes the routine provided by the Segment Manager.

Routine

The Segment Manager provides only one routine, the `UnloadSeg` procedure.

UnloadSeg

You can unload a segment by calling the `UnloadSeg` procedure.

```
PROCEDURE UnloadSeg (routineAddr: Ptr);
```

`routineAddr`
The address of any externally referenced routine in the segment to unload.

DESCRIPTION

The `UnloadSeg` procedure unloads a segment, making its storage relocatable and purgeable. You specify which segment to unload by passing the address of any externally referenced routine in that segment. The segment won't actually be purged until the memory it occupies is needed. If the segment is purged, the Segment Manager reloads it the next time one of the routines in it is called.

Note
The `UnloadSeg` procedure works only if called from outside the segment to be unloaded. u

Summary of the Segment Manager

Pascal Summary

Routine

```
PROCEDURE UnloadSeg          (routineAddr: Ptr);
```

C Summary

Routine

```
pascal void UnloadSeg        (void *routineAddr);
```

Assembly-Language Summary

Global Variables

`CurJTOffset`	word	Offset to jump table from location pointed to by A5.
`LoadTrap`	byte	If nonzero, call `_Debugger` before executing routine in a newly loaded segment.
`SegHiEnable`	byte	If nonzero, don't call `MoveHHi` when loading segments.

Advanced Routine

Trap macro	On entry
`_LoadSeg`	stack: segment number (word)

CHAPTER 8

Shutdown Manager

Contents

This chapter describes the Shutdown Manager, the part of the Operating System that manages the final stages of shutting down or restarting a Macintosh computer. The Shutdown Manager allows you to install a custom procedure that is executed during the process of shutting down or restarting. You can also use the Shutdown Manager to restart or shut down the computer directly, although this practice is strongly discouraged.

WARNING

For reasons described later, you should avoid shutting down or restarting the computer directly except in an emergency (for instance, when data on the disk might be destroyed). If you need to restart or shut down the system, send a Shutdown or Restart event to the Finder, as described in "Sending a Shutdown or Restart Event" on page 8-7.

Read the information in this chapter if your application or other software component needs to intervene in the standard process of shutting down or restarting the computer. In general, applications do not need to intervene in this process. You are likely to use the Shutdown Manager only if you are designing a device driver or system extension requiring notification that the computer is about to be shut down or restarted.

If you want to install a custom shutdown procedure, you should know how to install a code segment into the system heap, as described in the chapter "Memory Manager" in *Inside Macintosh: Memory*. If you want to shut down or restart the computer and need to familiarize yourself with the process of sending Apple events, see the chapter "Apple Event Manager" in *Inside Macintosh: Interapplication Communication*.

This chapter begins with a description of the Shutdown Manager and of the typical shutdown or restart process. Then it describes how you can

- use Apple events to request that the system be shut down or restarted
- install a custom shutdown procedure to be executed during the shutdown or restart process
- remove a shutdown procedure that you have previously installed

About the Shutdown Manager

The Shutdown Manager gives applications and other software a chance to perform any necessary shutdown processing before the computer is turned off or restarted. It also shuts down or restarts Macintosh computers, providing a consistent human interface for shutting down and restarting different models. Before restarting a computer or turning off the power, the Shutdown Manager checks for open device drivers and desk accessories and allows them to perform any necessary housekeeping.

The Shutdown Manager does not notify open applications that they are about to be shut down. This notification is handled by the Process Manager, as explained in "Using the Shutdown Manager" beginning on page 8-7. The Shutdown Manager provides two procedures that allow you to shut down or restart a Macintosh computer. However, these procedures do not perform the preliminary tasks that the Finder initiates when a

user chooses Shut Down or Restart from the Finder's Special menu. Accordingly, your application should not call these procedures directly because all open applications will terminate abruptly without the opportunity to save their current states and exit gracefully. Instead, your application should send a Shutdown or Restart event to the Finder, as described in "Sending a Shutdown or Restart Event" on page 8-7.

The main function of the Shutdown Manager is to execute a custom **shutdown procedure** that lets your application perform some additional tasks before the computer shuts down. The types of software most likely to install shutdown procedures are device drivers and system extensions. For example, drivers of early hard disk drives that use stepper motors usually need to park the drive heads in a safe zone before the power is turned off. A shutdown procedure could notify a driver to park the head. In another case, a user could install a system extension that displays an alert box asking whether to back up the hard disk before the computer shuts down.

The Shutdown Process

When a user chooses Shut Down or Restart from the Finder's Special menu, the tasks performed to shut down or restart a Macintosh computer differ in one important respect from those performed when you call the Shutdown Manager directly. In the former case, the Finder receives notification of a Shutdown or Restart event and calls the Process Manager to notify any open applications to quit. Only after the applications return does the Finder call the appropriate Shutdown Manager procedures to shut down or restart the system. To have your driver or application initiate this process, you can send a Shutdown or Restart event to the Finder, as described in "Sending a Shutdown or Restart Event" on page 8-7.

The Shutdown Manager procedures for shutting down or restarting the system (either `ShutDwnPower` or `ShutDwnStart`) perform an identical five-step process:

1. Checking for and executing custom shutdown procedures installed by calls to `ShutDwnInstall`. (This step occurs three times during the shutdown process.)
2. Checking the Device Manager's unit table to determine whether any drivers or desk accessories are open and, if so, notifying them of the impending shutdown or restart.
3. Saving the desk scrap, if any.
4. Unmounting mounted volumes.
5. Turning off the computer.

This section describes the shutdown process in detail, beginning with the preliminary step, mediated by the Finder, of closing applications.

Closing Open Applications

When a user or application notifies the Finder to shut down or restart the computer, the Finder calls the Process Manager. The Process Manager performs the important task of notifying all open applications to quit. It checks its list of open applications and sends a Quit Application event to those applications that can process Apple events. For applications that cannot process Apple events, the Process Manager sends a mouse-down event indicating that Quit was chosen from the File menu. This technique works for applications that display Quit in the File menu. Applications that display Quit in a different menu or that display a different form of Quit (such as Quit Document or Quit…) must specify a resource of type `'mstr'` or `'mst#'` with a resource ID of 100 or 101, respectively. The Process Manager reads these resources to locate the menu containing the Quit item or to find the exact Quit string to send to the application.

Once notified, open applications have the opportunity to save data and execute other exit procedures before they quit. Note that `ShutDwnPower` and `ShutDwnStart` do not notify open applications to quit. For this reason, you should not call these routines directly.

Checking for Custom Shutdown Procedures

After all open applications have quit, the Finder calls either `ShutDwnPower` or `ShutDwnStart`, respectively, depending on whether the user chose Shut Down or Restart from the Finder's Special menu. Because these two procedures perform the same set of tasks, the ensuing explanation applies to both routines.

The `ShutDwnPower` routine first checks for custom shutdown procedures installed by calls to `ShutDwnInstall`. The Shutdown Manager maintains a queue that contains the address of each custom procedure and a constant indicating when during the shutdown process to execute each procedure. The `ShutDwnPower` routine reads this queue three times during the shutdown process: before notifying drivers to shut down, before unmounting volumes, and before turning off the power. (See the description of `ShutDwnInstall` on page 8-13 for an explanation of the shutdown constants.) At this point, `ShutDwnPower` executes any custom procedures that specify the `sdOnDrivers` constant. Then it begins the next step of the shutdown process: checking for open device drivers.

Checking for Open Device Drivers

After locating any custom shutdown procedures, `ShutDwnPower` checks the Device Manager's unit table to determine whether any device drivers or desk accessories are open. It also inspects the `dNeedGoodBye` bit in the `drvrFlags` word for each driver. This bit, if set, indicates that the driver requests notification when the application heap is reinitialized or the system shuts down. Accordingly, `ShutDwnPower` calls the driver's `Control` function with the `csCode` field set to –1 (the `goodBye` global constant). This notification of impending termination is called a **good-bye message.**

A driver in an application heap also receives a good-bye message every time an application quits. For this reason, the driver cannot always determine whether a good-bye message means that the system is about to shut down. If making this distinction is important, you can call `ShutDwnInstall` to install a simple procedure that informs your driver when the computer is about to shut down. For more information about the `drvrFlags` word and the `Control` function, see the chapter "Device Manager" in *Inside Macintosh: Devices.*

The `ShutDwnPower` procedure does not actually close the drivers. They stay open until the power is switched off.

Saving the Desk Scrap

Having sent a good-bye message to any open driver that requested one, `ShutDwnPower` next calls the Scrap Manager function `UnloadScrap` to write the desk scrap, if any, from the Clipboard to the disk. Later, when the user restarts the computer, your application can retrieve the desk scrap by calling the `LoadScrap` function, as described in the chapter "Scrap Manager" of *Inside Macintosh: More Macintosh Toolbox*.

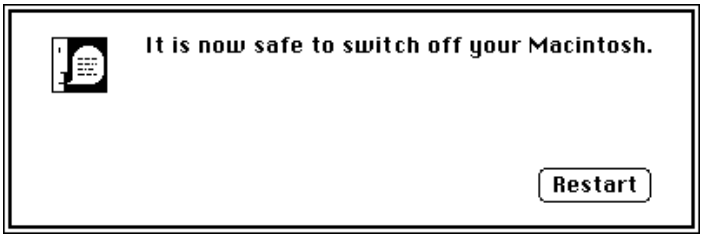
Unmounting Volumes

After saving the desk scrap, the `ShutDwnPower` procedure reads the Shutdown Manager's queue and executes any shutdown procedures that specify the `sdOnUnmount` constant. Next, `ShutDwnPower` searches the volume control block queue for mounted volumes. It unmounts each one by calling the File Manager functions `Eject` and `UnmountVol`. The `ShutDwnPower` procedure then reads the Shutdown Manager's queue and executes any shutdown procedures that specify the `sdOnRestart` constant, the `sdOnPowerOff` constant, or both.

Turning Off the Computer

Currently, there are two methods of turning off the various Macintosh models: one is software-controlled; the other, manual. With the software-controlled method, the Shutdown Manager actually turns off the power. With the manual method, by contrast, the Shutdown Manager darkens the screen and displays an alert box (Figure 8-1) stating that it is safe to turn off the computer.

Figure 8-1 A shutdown alert box



Currently, the product lines that employ the software-controlled method are the Macintosh II models, the Macintosh Quadra models, the Macintosh Portable computers, and the PowerBook computers. Those that employ the manual method are the Macintosh LC computers, the Macintosh SE computers, the Macintosh Classic computers, the Macintosh Plus models, and all earlier models.

All Macintosh models restart the same way when a user chooses Restart from the Special menu or when the Finder or other software calls the `ShutDwnStart` procedure. Remember not to call `ShutDwnPower` and `ShutDwnStart` directly because these procedures abruptly terminate other applications that are currently running, possibly resulting in a loss of data.

Using the Shutdown Manager

The Shutdown Manager provides four procedures. The procedures `ShutDwnPower` and `ShutDwnStart` perform the same set of shutdown tasks, except that `ShutDwnPower` turns off a Macintosh computer, whereas `ShutDwnStart` restarts it. The `ShutDwnInstall` routine installs a custom shutdown procedure to perform a certain task before the computer shuts down or restarts. The `ShutDwnPower` or `ShutDwnRestart` routine calls your shutdown procedure at a predetermined point during the shutdown or restart process. The last procedure, `ShutDwnRemove`, removes custom shutdown procedures installed by `ShutDwnInstall`.

s **WARNING**
Usually, only the Finder or other system software should call `ShutDwnPower` and `ShutDwnInstall`. An application calling these procedures will cause other open applications to terminate abruptly, potentially destroying their data. s

Sending a Shutdown or Restart Event

Applications that support high-level events can send a Shutdown or Restart event to the Finder to request the system to shut down or restart. Once notified, the Finder calls the Process Manager, which gives open applications the opportunity to exit gracefully before the computer shuts down or restarts. The Process Manager checks its list of open applications and sends a Quit Application event to applications that can process Apple events. For applications that can't, the Process Manager sends a mouse-down event indicating that Quit was chosen from the File menu. Applications that display the Quit item in a different menu or that use a different wording must specify a resource of type `'mstr'` or `'mst#'` with a resource ID of 100 or 101, respectively. Once notified, the open applications then have time to perform cleanup operations (such as displaying a Save Changes alert box) before quitting.

The Shutdown and Restart events have the event class defined by the `kAEFinderEvents` constant.

```
CONST
   kAEFinderEvents = 'FNDR';     {event class for Finder}
```

The Restart event has the event ID defined by the `kAERestart` constant, and the Shutdown event has the event ID defined by the `kAEShutDown` constant:

```
CONST
   kAERestart = 'rest';    {event ID for Restart event}
   kAEShutDown = 'shut';   {event ID for Shutdown event}
```

Listing 8-1 defines a function that sends a Shutdown event to the Finder.

Listing 8-1 Sending a Shutdown event

```
FUNCTION ShutDownSafely: OSErr;
CONST
   kFinderSig = 'FNDR';
VAR
   myErr:         OSErr;
   finderAddr:    AEDesc;
   myShutDown:    AppleEvent;
   nilReply:      AppleEvent;
BEGIN
   myErr := AECreateDesc(typeApplSignature, kFinderSig,
                         SizeOf(OSType), finderAddr);
   IF myErr = noErr THEN
      myErr := AECreateAppleEvent(kAEFinderEvents, kAEShutDown,
                                  finderAddr, kAutoGenerateReturnID,
                                  kAnyTransactionID, myShutDown);
   IF myErr = noErr THEN
      myErr := AESend(myShutDown, nilReply, kAENoReply +
                  kAECanSwitchLayer + kAEAlwaysInteract,
                  kAENormalPriority, kAEDefaultTimeout, NIL, NIL);
   ShutDownSafely := myErr;
END;
```

To send a Shutdown or Restart event, you must call three Apple Event Manager functions. First, use the `AECreateDesc` function to create an address descriptor record that specifies the address of the Finder. You can specify the address of the Finder by its signature, `'FNDR'`. Next, call `AECreateAppleEvent` to create the Apple event you want to send. Finally, call the `AESend` function. Use the Apple event returned in the

myShutDown variable of the AECreateAppleEvent function as the Apple event to send in AESend.

After sending the event, remember to dispose of the descriptor record and Apple event at some point, by calling the AEDisposeDesc function. For complete details about this function and the ones used in Listing 8-1, see the chapter "Apple Event Manager" in *Inside Macintosh: Interapplication Communication*.

Note
Applications running under system software version 6.0.x cannot send Apple events to MultiFinder because it cannot process them. As a result, your application cannot request that open applications be notified to quit before it calls ShutDwnPower and ShutDwnStart. Therefore, you should avoid calling these procedures unless absolutely necessary. u

Installing a Custom Shutdown Procedure

If you write a shutdown procedure, you can install a pointer to it in the Shutdown Manager's queue by calling the ShutDwnInstall procedure. You're most likely to need to use a custom shutdown procedure if you are writing a device driver or a system extension. For example, drivers for early hard disk drives that use stepper motors usually need to park the drive heads in a safe zone before the power is turned off. Similarly, drivers for floppy disks and CD-ROM discs use a shutdown procedure that ejects the disks so that they don't remain in the drives when the computer shuts down.

If you are developing an application, you can also make use of shutdown procedures. For example, a remote backup application might install a shutdown procedure that reminds a user about scheduled backups. If the user attempts to shut down the computer before the application has backed up the disk, the shutdown procedure could display an alert box asking whether the user wants to back up the disk before the computer shuts down.

Remember that the Process Manager frees all application heaps before the Finder calls ShutDwnPower. For this reason, you can't rely on your heap being intact. You should load your shutdown procedure into the system heap and specify the constants sdOnDrivers or sdOnUnmount to ensure that the procedure is executed while the system heap and any necessary system software components are still available.

The ShutDwnInstall procedure accepts a number of constants that specify when during the shutdown process ShutDwnPower or ShutDwnStart should execute your custom procedure. You can specify more than one constant to have your procedure executed at different phases of the process. The points indicated by these constants are: before the drivers receive good-bye messages, before volumes are unmounted, or before the computer is restarted or the power supply is switched off. However, Apple Computer, Inc., cannot guarantee the state of the computer after volumes are unmounted. Accordingly, if you plan to use the system heap or call Toolbox or Operating System routines to open a file, display a dialog box, play a sound, and so forth, be sure to specify the sdOnDrivers or sdOnUnmount constants. For more information about these constants, see the description of the ShutDwnInstall routine on page 8-13.

Listing 8-2 illustrates a sample custom shutdown procedure that ejects a CD-ROM disc just before the Macintosh computer shuts down or restarts.

Listing 8-2 A sample custom shutdown procedure

```
PROCEDURE MyShutDownProc;
CONST
   kMaxScsiID = 7;
VAR
   MyDCEHandle:      DCtlHandle;
   CDRefNum:         Integer;       {driver reference number}
   myID:             Integer;       {SCSI ID}
   MyDevStatHandle:  DevStatHandle; {handle to driver's array}
BEGIN
   {Read driver reference number from the unit table.}
   CDRefNum := GetMyRefNum;
   IF CDRefNum = 0 THEN
      Exit(MyShutDownProc);

   {Get handle to driver's device control entry.}
   MyDCEHandle := GetDCtlEntry(CDRefNum);

   {If handle is NIL, couldn't get device control entry.}
   IF MyDCEHandle = NIL THEN
      Exit(MyShutDownProc);

   {Eject all mounted CD-ROM discs.}
   MyDevStatHandle := DevStatHandle(MyDCEHandle^^.dCtlStorage);
   FOR myID := 0 to kMaxScsiID DO
   IF (MyDevStatHandle^^[myID].isMyCDDrive = TRUE) AND
      (MyDevStatHandle^^[myID].mounted = TRUE) THEN
      MyEjectCDProc(myID);          {your routine to eject CDs}
END;
```

In Listing 8-2, `GetMyRefNum` returns the reference number for a CD device driver from the Device Manager's unit table. A value of 0 indicates that `GetMyRefNum` did not find a CD device driver. Next, `MyShutDownProc` calls `GetDCtlEntry` to retrieve the handle to the CD driver's device control entry record. Both the handle (of type `DCtlHandle`) and the device control entry record (of type `DCE`) are described in the chapter "Device Manager" in *Inside Macintosh: Devices*. If the value returned is `NIL`, then `GetDCtlEntry` did not find the device control entry, and `MyShutDownProc` returns. The `MyShutDownProc` procedure next loops through the driver's device status array looking for and ejecting mounted compact discs. The `MyDevStatHandle` handle points

to the driver's array. After checking all elements in the array, `MyShutDownProc` returns control to the Shutdown Manager.

Note
Listing 8-2 does not define the device driver's array or the `GetMyRefNum` and `MyEjectCDProc` routines. These items are internal to the CD-ROM driver. u

Once you have finished writing your shutdown procedure, you can install it by calling `ShutDwnInstall`. Your call to `ShutDwnInstall` should specify a pointer to `MyShutDownProc`, as follows.

```
ShutDwnInstall(@MyShutDwnProc, sdOnDrivers);
```

Assembly-Language Note
When the Shutdown Manager calls a shutdown procedure, it sets a bit in the D0 register indicating the current phase of the shutdown process. The values 0, 1, 2, and 3 represent the constants `sdOnPowerOff`, `sdOnRestart`, `sdOnUnmount`, and `sdOnDrivers`, respectively. You can have your shutdown procedure read D0 if it needs to keep track of the shutdown process. u

The Shutdown Manager follows the standard conventions for saving registers specified in *Inside Macintosh: Overview*.

You can remove your shutdown procedure at any time by calling `ShutDwnRemove`.

Shutdown Manager Reference

This section describes the routines and constants that are specific to the Shutdown Manager. For a description of the constants provided by the Shutdown Manager, see the description of the `ShutDwnInstall` procedure on page 8-13. The section "Summary of the Shutdown Manager" lists these routines and constants for your reference.

Shutdown Manager Routines

This section first describes the routines for shutting down or restarting a Macintosh computer. It then describes the routines for installing or removing a custom shutdown procedure.

Shutting Down or Restarting a Macintosh Computer

The Shutdown Manager provides the routines `ShutDwnPower` and `ShutDwnStart` to shut down or restart the computer.

s **WARNING**
The `ShutDwnPower` and `ShutDwnStart` procedures are used by the Finder and other system software. You usually do not need to call these two routines. s

ShutDwnPower

The system software calls the `ShutDwnPower` procedure to shut down a Macintosh computer.

```
PROCEDURE ShutDwnPower;
```

DESCRIPTION

The `ShutDwnPower` procedure initiates the final stage of the system shutdown process. It performs system housekeeping, executes any custom shutdown procedures installed by calls to `ShutDwnInstall`, and, if possible, turns the computer off. (The Shutdown Manager displays the Shutdown alert box if the user has to turn the computer off manually.) The system housekeeping functions consist of a five-step process, described in full in "The Shutdown Process" on page 8-4.

You should always call `ShutDwnPower` indirectly, through the Finder, to give any other applications running at the time a chance to exit gracefully. "Sending a Shutdown or Restart Event" on page 8-7 describes the correct way to shut down a Macintosh computer.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `ShutDwnPower` procedure are

Trap macro	Selector
`_Shutdown`	$0001

ShutDwnStart

The system software calls the `ShutDwnStart` procedure to restart a Macintosh computer.

```
PROCEDURE ShutDwnStart;
```

DESCRIPTION

The `ShutDwnStart` procedure initiates the final stage of restarting the system. It performs system housekeeping, executes any custom shutdown procedures installed with `ShutDwnInstall`, and restarts the computer. The system housekeeping functions consist of a five-step process, described in full in "The Shutdown Process" on page 8-4.

You should always call `ShutDwnStart` indirectly, through the Finder, to give any other applications running at the time a chance to exit gracefully. "Sending a Shutdown or Restart Event" on page 8-7 describes the correct way to restart a Macintosh computer.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `ShutDwnStart` procedure are

Trap macro	Selector
`_Shutdown`	$0002

Installing or Removing a Shutdown Procedure

The Shutdown Manager provides the routines `ShutDwnInstall` and `ShutDwnRemove` to install and remove custom shutdown procedures.

ShutDwnInstall

You can use the `ShutDwnInstall` procedure to install a custom shutdown procedure that performs a certain task before the computer shuts down or restarts.

```
PROCEDURE ShutDwnInstall (shutDownProc: ProcPtr; flags: Integer);
```

`shutDownProc`
A pointer to your shutdown procedure.

`flags` An integer that indicates when during the shutdown process to execute your shutdown procedure.

DESCRIPTION

The `ShutDwnInstall` procedure installs the custom shutdown procedure pointed to by the `shutDownProc` parameter. You can install more than one custom procedure; simply call `ShutDwnInstall` for each one. For complete information on using a shutdown procedure, see "Installing a Custom Shutdown Procedure" on page 8-9.

The `flags` parameter indicates when during the shutdown process `ShutDwnPower` or `ShutDwnStart` executes your shutdown procedure. The following constants serve as masks for setting the bits in the `flags` parameter. Set the appropriate bits to have your procedure executed at different points during shutdown.

```
CONST
   sdOnPowerOff      = 1;  {call procedure before power off}
   sdOnRestart       = 2;  {call procedure before restart}
   sdRestartOrPower  = 3;  {call procedure before power off }
                           { or restart}
   sdOnUnmount       = 4;  {call procedure before unmounting }
                           { volumes}
   sdOnDrivers       = 8;  {call procedure before checking for }
                           { open drivers}
```

The following list indicates when `ShutDwnPower` or `ShutDwnStart` executes your procedure and summarizes the known state of the computer at the point specified by each constant:

Constant	Description
`sdOnDrivers`	The Shutdown Manager executes your procedure before checking the Device Manager's unit table for open drivers. All Toolbox and Operating System managers are available. The system heap is available. It is safe to open files, display dialog boxes, play sounds, or perform similar tasks.
`sdOnUnmount`	The Shutdown Manager executes your procedure before unmounting volumes. All Toolbox and Operating System managers are available. The system heap is available. It is safe to open files, display dialog boxes, play sounds, or perform similar tasks.
`sdOnRestart`	The Shutdown Manager executes your procedure before restarting the computer. The system heap is still available. However, in other respects, the state of the computer is indeterminate.
`sdOnPowerOff`	The Shutdown Manager executes your procedure before switching off the power supply or displaying the shutdown alert box. The system heap is still available. However, in other respects, the state of the computer is indeterminate.
`sdRestartOrPower`	The Shutdown Manager executes your procedure before restarting the computer or before switching off the power supply or displaying the shutdown alert box. The system heap is still available. However, in other respects, the state of the computer is indeterminate.

You can also combine these constants in the following ways:

Expression	Description
`sdOnPowerOff + sdOnDrivers`	When the computer is shutting down, `ShutDwnPower` calls your shutdown procedure before checking for open drivers.
`sdOnPowerOff + sdOnUnmount`	When the computer is shutting down, `ShutDwnPower` calls your shutdown procedure before unmounting volumes.
`sdOnRestart + sdOnDrivers`	When the computer is to be restarted, `ShutDwnStart` calls your shutdown procedure before checking for open drivers.
`sdOnRestart + sdOnUnmount`	When the computer is to be restarted, `ShutDwnStart` calls your shutdown procedure before unmounting volumes.

Note

These combinations of constants are recognized by the Shutdown Manager only in system software versions 7.0 and later. u

The Shutdown Manager executes a custom shutdown procedure just once. As soon as a custom procedure returns, the Shutdown Manager removes the address and flag entries for that procedure from its shutdown queue. As a result, the combination `sdOnDrivers + sdOnUnmount` does not work.

If your driver or system extension remains resident in memory after the boot process, be sure to load your shutdown procedure into the system heap because the Process Manager frees all application and other temporary heaps before calling the Shutdown Manager.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `ShutDwnInstall` procedure are

Trap macro	Selector
`_Shutdown`	$0003

ShutDwnRemove

The `ShutDwnRemove` procedure removes a shutdown procedure that you have previously installed by calling `ShutDwnInstall`.

```
PROCEDURE ShutDwnRemove (shutDownProc: ProcPtr);
```

`shutDownProc`
A pointer to your shutdown procedure.

DESCRIPTION

The `ShutDwnRemove` procedure removes the shutdown procedure pointed to by the `shutDownProc` parameter. The `ShutDwnRemove` procedure is useful for removing a shutdown procedure when the device controlled by the driver that installed the shutdown procedure is not operating.

If you have specified that your procedure should be executed at several points during the shutdown process (for instance, before unmounting at restart and before unmounting at power off), `ShutDwnRemove` removes it at all points.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `ShutDwnRemove` procedure are

Trap macro	Selector
`_Shutdown`	$0004

Application-Defined Routine

While the computer is restarting or shutting down, you can provide a custom shutdown procedure to perform any housekeeping tasks that your device driver or system extension requires. For example, your device driver might have to park the drive head or your system extension might have to write some statistics to a log. However, under normal circumstances, applications don't need to use a custom shutdown procedure.

Shutdown Procedures

A shutdown procedure performs any last-minute tasks required to put a device driver (or, in rare cases, an application) in a stable state before the computer restarts or shuts down.

Note

Applications can usually perform housekeeping tasks before they quit. If your application requires that some action be taken after it quits, such as having the system display a dialog box, you should use a system extension whenever possible. u

MyShutDownProc

A typical shutdown procedure has the following form:

```
PROCEDURE MyShutDownProc;
```

DESCRIPTION

You can install the address of your shutdown procedure in the Shutdown Manager's queue by calling the `ShutDwnInstall` procedure. When the computer restarts or shuts down, the Shutdown Manager searches its queue, reads the addresses it finds there, and executes the procedures at the points specified by the shutdown flag or flags that you passed with `ShutDwnInstall`. See the description of `ShutDwnInstall` on page 8-13 for details.

To remove your shutdown procedure, call the `ShutDwnRemove` routine, passing it a pointer to your procedure.

Be sure to install your shutdown procedure in the system heap. Because the Process Manager deallocates memory for all application heaps before the Finder calls `ShutDwnPower`, you can't rely on the application heap being intact. In addition, if you need to use Toolbox managers, specify the constants `sdOnDrivers` or `sdOnUnmount` to ensure that your procedure is executed while these managers are still available.

ASSEMBLY-LANGUAGE INFORMATION

The Shutdown Manager conforms with the standard assembly-language conventions for saving registers. The Shutdown Manager does not preserve the contents of these registers. Therefore, be sure to save and restore the data and address registers before issuing a jump instruction to your shutdown procedure.

Summary of the Shutdown Manager

Pascal Summary

Constants

```
CONST
   {masks for ShutDwnInstall flags}
   sdOnPowerOff      = 1;  {call procedure before power off}
   sdOnRestart       = 2;  {call procedure before restart}
   sdRestartOrPower  = 3;  {call procedure before power off or restart}
   sdOnUnmount       = 4;  {call procedure before unmounting volumes}
   sdOnDrivers       = 8;  {call procedure before checking for open drivers}
```

Shutdown Manager Routines

Shutting Down or Restarting the Computer

```
PROCEDURE ShutDwnPower;
PROCEDURE ShutDwnStart;
```

Installing or Removing a Shutdown Procedure

```
PROCEDURE ShutDwnInstall    (shutDownProc: ProcPtr; flags: Integer);
PROCEDURE ShutDwnRemove     (shutDownProc: ProcPtr);
```

Application-Defined Routine

Shutdown Procedures

```
PROCEDURE MyShutDownProc;
```

C Summary

Constants

```
/*masks for ShutDwnInstall flags*/
enum {
   sdOnPowerOff      = 1,  /*call procedure before power off*/
   sdOnRestart       = 2,  /*call procedure before restart*/
   sdRestartOrPower  = 3,  /*call procedure before power off or restart*/
   sdOnUnmount       = 4,  /*call procedure before unmounting volumes*/
   sdOnDrivers       = 8   /*call procedure before checking for open */
};                         /* drivers*/
```

Data Types

```
typedef pascal void (*ShutDwnProcPtr)(void)
```

Shutdown Manager Routines

Shutting Down or Restarting the Computer

```
pascal void ShutDwnPower    (void);
pascal void ShutDwnStart    (void);
```

Installing or Removing a Shutdown Procedure

```
pascal void ShutDwnInstall  (ShutDwnProcPtr shutDownProc, short flags);
pascal void ShutDwnRemove   (ShutDwnProcPtr shutDownProc);
```

Application-Defined Routine

Shutdown Procedures

```
pascal void MyShutDownProc  (void);
```

Assembly-Language Summary

Constants

```
sdPowerOff      EQU     1       ;selector for ShutDwnPower
sdRestart       EQU     2       ;selector for ShutDwnStart
sdInstall       EQU     3       ;selector for ShutDwnInstall
sdRemove        EQU     4       ;selector for ShutDwnRemove
```

Trap Macros Requiring Routine Selectors

_Shutdown

Selector	Routine
$0001	ShutDwnPower
$0002	ShutDwnStart
$0003	ShutDwnInstall
$0004	ShutDwnRemove

Glossary

A5 world An area of memory in an application's partition that contains the QuickDraw global variables, the application global variables, the application parameters, and the jump table—all of which are accessed through the A5 register.

active application The application currently interacting with the user. Its icon appears on the right side of the menu bar. See also **current process, foreground process.**

alert notification A notification in which an alert box containing a short message appears on the screen.

Apple events High-level events whose structure and interpretation are determined by the Apple Event Interprocess Messaging Protocol.

application heap An area of memory in the application heap zone in which memory is dynamically allocated and released on demand. The heap contains the application's `'CODE'` segment 1, data structures, resources, and other code segments as needed.

application partition A partition of memory reserved for use by an application. The application partition consists of free space, the application heap, the application's stack, and the application's A5 world.

asynchronous execution A mode of invoking a routine. During the asynchronous execution of a routine, an application is free to perform other tasks. See also **completion routine.**

audible notification A notification in which the Sound Manager plays the system alert sound or a sound contained in an `'snd '` resource.

background-only application An application that does not have a user interface.

background process A process that isn't currently interacting with the user. See also **foreground process.**

completion routine A routine that is executed when an asynchronous call to some other routine is completed.

context Information the Process Manager maintains about a process. This information includes the current state of the process, the address and size of its partition, its type, its creator, a copy of its low-memory global variables, information about its `'SIZE'` resource, and its process serial number.

context switch A major or minor switch.

cooperative multitasking environment A multitasking environment in which applications explicitly cooperate to share the available system resources. See also **multitasking environment.**

current process The process that is currently executing and whose A5 world is valid. This process can be in the background or foreground.

defer To postpone the execution of an interrupt task until all interrupts have been reenabled.

deferred task An interrupt task whose execution has been postponed until interrupts have been reenabled.

Deferred Task Manager The part of the Macintosh Operating System that allows you to defer the execution of lengthy interrupt tasks until interrupts have been reenabled.

deferred task queue An operating-system queue that contains deferred task records.

deferred task record A record that contains information about a deferred task. Defined by the `DeferredTask` data type.

desk accessory A "mini-application" that is available from the Apple menu regardless of which application you're using—for example, the Calculator, Note Pad, Alarm Clock, Puzzle, Scrapbook, Key Caps, and Chooser.

desktop The working environment on the computer—the menu bar and the gray area on the screen. The user can have a number of documents on the desktop at the same time. At the Finder level, the desktop displays the Trash icon and the icons (and windows) of volumes that have been mounted.

device A part of a computer, or a piece of external equipment, that can transfer data into or out of the computer.

device driver A program that controls the exchange of information between an application and a device.

Device Manager The part of the Macintosh Operating System that supports device I/O.

disabled interrupt An interrupt whose priority level is lower than or the same as that of an interrupt that is currently being serviced.

drift To deviate or vary from scheduled execution.

drift-free Executed precisely as scheduled, without drifting.

event The means by which the Event Manager communicates information about user actions, changes in the processing status of the application, and other occurrences that require a response from the application.

Event Manager The collection of routines that an application can use to receive information about actions performed by the user, to receive notice of changes in the processing status of the application, and to communicate with other applications.

exception An error or other special condition detected by the microprocessor in the course of program execution.

external reference A reference to a routine or variable defined in another code segment.

fixed-frequency Of constant frequency.

foreground process The process that is currently interacting with the user; it appears to the user as the active application. The foreground process displays its menu bar, and its windows are in front of the windows of all other applications. See also **background process.**

frequency The number of times per second that an action (such as the issuance of an interrupt) occurs. An action's frequency is measured in cycles per second, or hertz. See also **period.**

good-bye message A message sent by the Operating System to notify device drivers when an application quits or the system shuts down. To receive a good-bye message, drivers must set the `dNeedGoodBye` bit in the `drvrFlags` word.

hertz (Hz) A unit of frequency, equal to one cycle per second.

interrupt An exception signaled by a device to the processor, notifying it of a change in the condition of the device, such as the completion of an I/O request.

interrupt handle A routine that services interrupts.

interrupt priority level A number that identifies the importance of an interrupt. It indicates which device is interrupting, and which interrupt handler should be executed in response to the interrupt.

interrupt table A list (stored in low memory) of interrupt vectors.

interrupt task A routine executed as the result of an interrupt.

interrupt vector The address of an interrupt handler.

jump table An area of memory in an application's A5 world that contains one entry for every externally referenced routine in every code segment of the application. The jump table is the means by which segments are loaded and unloaded.

jump table entry A single entry in a jump table.

kill To cause a process or task to stop executing.

load To move a segment into RAM.

Macintosh Operating System The part of Macintosh system software that manages basic low-level operations such as file reading and writing, memory allocation and deallocation, process execution, and interrupt handling.

main segment The segment that contains the main program.

major switch The process of switching the context of the foreground process with the context of a background process (including the A5 worlds and application-specific system global variables) and bringing the background process to the front, sending the previous foreground process to the background. See also **context, minor switch.**

microsecond A unit of time equal to one millionth of a second. Abbreviated µsec.

millisecond A unit of time equal to one thousandth of a second. Abbreviated msec.

minor switch The process of switching the context of a process to give time to a background process without bringing the background process to the front. See also **context, major switch.**

multitasking environment An environment in which several independent applications or other processes can be open at once. See also **cooperative multitasking environment.**

notification An audible or visible indication that your application (or other piece of software) requires the user's attention. See also **alert notification, audible notification,** and **polite notification.**

Notification Manager The part of the Macintosh Operating System that allows you to inform users of significant occurrences in applications that are running in the background or in software that is largely invisible to the user.

notification queue The Notification Manager's list of pending notification requests.

notification record The internal representation of a notification request, through which you specify how a notification is to occur. Defined by the `NMRec` data type.

notification request A request to the Notification Manager to create a notification.

notification response procedure A procedure that the Notification Manager can execute as the final step in a notification.

null event An event signaling that there are no more events to report.

open application An application that is loaded into memory.

Operating System See **Macintosh Operating System.**

operating-system event An event returned by the Event Manager to communicate changes in the operating status of applications (suspend and resume events) and movement of the mouse outside of an area defined by the application (mouse-moved events).

operating-system queue See **queue.**

partition A contiguous block of memory reserved for use by the Operating System or by an application. See also **application partition** and **system partition.**

period The time elapsed during one complete cycle. See also **frequency.**

persistent VBL task A VBL task that is executed as scheduled, even when the application that installed it is switched out and is no longer in control of the CPU.

polite notification A notification in which a small icon blinks in the menu bar at the location of the Apple menu icon (the Apple logo) or the Application menu icon.

prime To activate a Time Manager task that is already installed in the Time Manager queue.

process An open application or, in some cases, an open desk accessory. (Only desk accessories that are not opened in the context of another application are considered processes.)

Process Manager The part of the Macintosh Operating System that provides a cooperative multitasking environment by controlling access to shared resources and managing the scheduling, execution, and termination of applications.

processor priority Bits in the status register of the CPU that indicate which interrupts are to be processed and which are to be ignored.

process serial number A number assigned by the Process Manager that identifies a particular instance of an application; this number is unique during a single boot of the local machine. Defined by the `ProcessSerialNumber` data type.

queue A list of identically structured entries linked together by pointers.

resume event An operating-system event that indicates that the execution of your application is about to be resumed. See also **suspend event.**

segment One of several logical divisions of the code of an application. Not all segments need to be in memory at the same time.

segment header A 4-byte area at the beginning of a segment that contains the offset of the first routine's entry from the beginning of the jump table (2 bytes) and the number of entries for the segment (2 bytes).

Segment Manager The part of the Macintosh Operating System that loads and unloads your application's code segments into and out of memory.

service To handle an interrupt by executing its interrupt handler.

Shutdown Manager The part of the Macintosh Operating System that manages the final stages of shutting down or restarting a Macintosh computer.

shutdown procedure An custom procedure installed by calling the `ShutDwnInstall` procedure and executed by the Shutdown Manager before the computer restarts or shuts down.

slot-based VBL task A VBL task that is linked to an external video monitor.

slot-card interrupt An interrupt sent by a slot device.

stack An area of memory in the application partition that is used to store temporary variables.

suspend event An operating-system event that indicates that the execution of your application is about to be suspended. See also **resume event.**

switch See **major switch** and **minor switch.**

synchronous execution A mode of invoking a routine. After calling a routine synchronously, an application cannot perform other tasks until the routine is completed.

system-based VBL task A VBL task that is not linked to an external video monitor.

system partition A partition of memory reserved for use by the Operating System.

terminate To end the execution of a process. A process can terminate by crashing, by quitting, or by being killed by some other process.

Time Manager The part of the Macintosh Operating System that lets you schedule the execution of a routine after a certain time has elapsed.

Time Manager queue A list of all installed Time Manager tasks.

Time Manager task record A data structure that contains information about a Time Manager task. Defined by the `TMTask` data type.

unload To unlock a segment. By unlocking unneeded segments, you allow them to be relocated or purged if necessary to accommodate a later memory-allocation request.

VBL See **vertical retrace interrupt.**

VBL task A task executed during a vertical retrace interrupt. See also **slot-based VBL task** and **system-based VBL task**.

VBL task record A data structure that contains information about a VBL task. Defined by the `VBLTask` data type.

vertical blanking interrupt (VBL) See **vertical retrace interrupt.**

vertical retrace interrupt An interrupt generated by the video circuitry each time the electron beam of a monitor's display tube returns from the lower-right corner of the screen to the upper-left corner.

Vertical Retrace Manager The part of the Operating System that schedules and executes tasks during a vertical retrace interrupt.

vertical retrace queue A list of the tasks to be executed during a vertical retrace interrupt.

virtual memory Addressable memory beyond the limits of the available physical RAM. The Operating System extends the logical address space by allowing unused applications and data to be stored on a secondary storage device instead of in physical RAM.

wake up To make a previously suspended process eligible to receive CPU time.

Index

A

B

C

D

E

F

G, H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W, X, Y, Z

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Proof pages were created on an Apple LaserWriter IINTX printer. Final page negatives were output directly from text files on an AGFA ProSet 9800 imagesetter. Line art was created using Adobe™ Illustrator. PostScript™, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.

WRITER
Tim Monroe

DEVELOPMENTAL EDITORS
Sean Cotter, Antonio Padial

ILLUSTRATOR
Peggy Kunz

PRODUCTION EDITOR
Josephine Manuele

ON-LINE PRODUCTION EDITOR
Gerri Gray

PROJECT MANAGER
Patricia Eastman

COVER DESIGNER
Barbara Smyth

Special thanks to David Harrison and Mike Puckett.

Acknowledgments to Michael Abramowicz, Scott Boyd, Sharon Everson, Sanborn Hodgkins, Jim Luther, Jim Reekes, Keith Rollin, and the entire *Inside Macintosh* team.