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.**

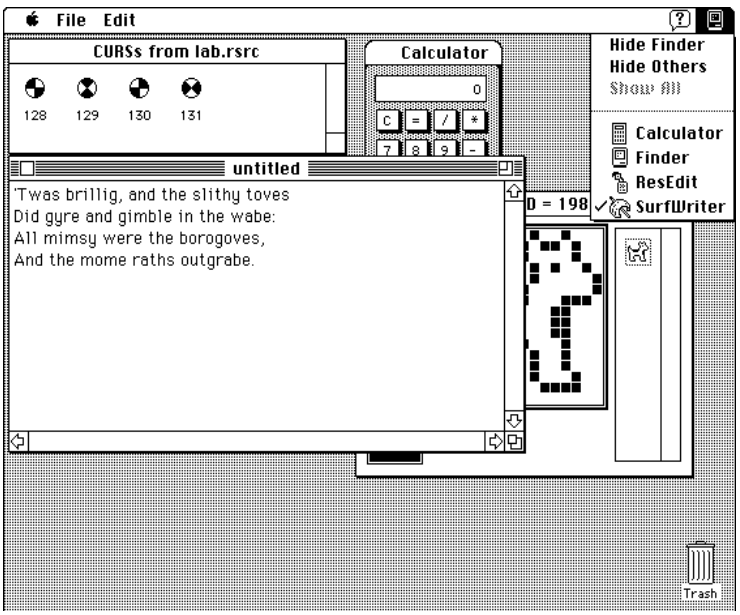
Note
The cooperative multitasking environment is available in system software versions 7.0 and later, and when the MultiFinder option is enabled in earlier system software versions. ◆

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. ◆

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. ◆

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. ◆

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. ◆

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.

▲ **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. ▲

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.