INSIDE MACINTOSH

Microkernel and Core System Services

WWDC Release

May 1996

Contents

About Mac OS 8

Contents

CHAPTER 1

About Mac OS 8

This chapter introduces Mac OS 8 with an emphasis on the microkernel and core operating system services. This chapter is intended primarily for experienced programmers who are interested in writing programs to run under Mac OS 8, but will also be of interest to anyone who wants an architectural overview of Mac OS 8. To read this chapter you should understand what a computer operating system is and be familiar with fundamental computer-related concepts such as memory and I/O.

This chapter starts with a high-level architectural overview of Mac OS 8; defines and discusses several key concepts such as tasks, processes, program scheduling, and memory protection; and introduces each of the components of the microkernel and core system services. The remaining chapters in this part describe the structure of a program in Mac OS 8 and introduce the chapters in the rest of the book.

Draft Release Note
The information in this chapter is preliminary and subject to change. Not all of the chapters planned for this book are included in this release. ◆

Mac OS 8 Architecture

Mac OS 8 is a software package developed by Apple Computer, Inc. Mac OS 8 includes a variety of low-level services to control hardware and to execute software, many services that simplify your job of providing the graphical interface and ease of use that users expect from a Mac-compatible computer, and user interface components such as the Finder.

This book describes those portions of Mac OS 8 that constitute the *microkernel*, plus some low-level operating system services referred to here as the *core system services*. An operating system **kernel** is the heart of the operating system. Typically, a kernel manages all or most of the operating system services necessary to control the computer. In a UNIX®-based operating system, for example, the kernel supervises task and file management, device input and output, and memory allocation. A **microkernel**, by contrast, is a kernel that operates only a small but critical subset of the computer's operating system services.

An operating system architecture based on a microkernel is highly modular. The microkernel handles all or most of the operations that must be customized for a particular microprocessor or hardware architecture. All other operating system services, including fundamental services such as I/O and graphics, are implemented as modules outside the microkernel. In a microkernel-based operating system, it should be possible to modify or replace any of these modules without affecting the others. For example, Apple could implement a new I/O subsystem without affecting graphics or file handling.

The Mac OS 8 microkernel provides basic task and process control and management of operating system resources associated with tasks and processes, such as memory. It also includes basic services such as synchronization, timing, and low-level messaging. It does not include other services—such as file management, I/O management, and support for the Mac OS human interface—that are part of Mac OS 8.

The elements of the Mac OS 8 microkernel occupy memory protected from most other executing code. Because the microkernel routines operate very efficiently, calling the microkernel directly is usually the fastest way to perform an operation. However, the microkernel interface includes only very basic operations. Higher-level operating system services are generally more powerful and easier to use. For example, whereas the Microkernel Messaging Service allows you to send data from one task to another, if you use that service you must establish your own protocols for exchanging addresses and interpreting the data. Apple events, by contrast, have established protocols for addressing and for data interpretation, can send messages over a network, and are used extensively by other system services. Apple events in Mac OS 8 are much faster than they were in previous versions of Mac OS. You should use microkernel services only if you have a specific need that can’t be fulfilled by higher-level system services.

This book documents the services provided by the microkernel plus several other low-level operating system services referred to, for the sake of convenience, as the **core system services**. The chapter “Introduction to Mac OS 8 Microkernel and Core System Services” (To be provided in a later release)contains a full list of services offered by the microkernel and the core system services.

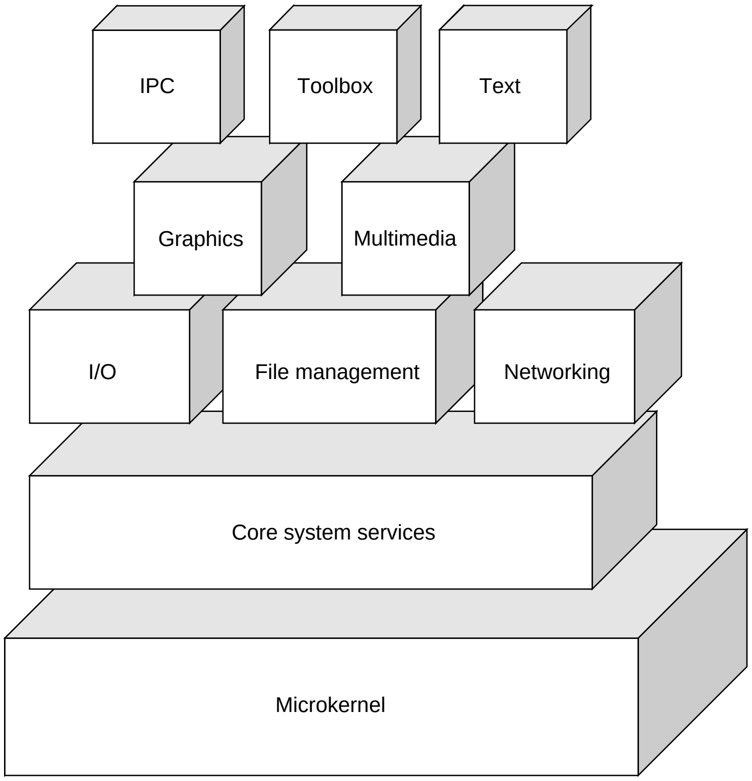
Figure 1-1 shows the major services offered by Mac OS 8 for programmers. In general, the services at any given level in the figure are, or can be, clients of the services below them in the figure. In many cases, it is also possible for a given service or program to skip intervening layers and call low-level services directly. For example, an application can call the interprocess communications

(IPC) services near the top of the figure to communicate with other entities in the system, or it can call the messaging service provided by the microkernel.

Constituents of Mac OS 8 include the following:

- The microkernel, which provides fundamental services that are used directly or indirectly by all other parts of Mac OS 8. Examples include creation and deletion of tasks and processes, task scheduling, low-level memory management, and low-level messaging services.
- The core system services, which are clients of the microkernel and which provide a variety of services for programs and for higher levels of the Mac OS. Core system services include, among others, the Dynamic Memory Allocation Service, the Code Fragment Manager, the Process Manager, System Notification Service, the Mixed Mode Manager, the Cooperative Thread Manager, the Patch Manager, and a variety of utilities.
- The I/O subsystem, which includes such constituents as the Device Manager, the NuBus™ and PCI bus APIs, the SCSI Manager, and the Display Manager. Also included in the I/O architecture, but treated as separate topics in the *Inside Macintosh* book suite, are the File System and the Open Transport networking services.
- The graphics subsystem, which includes QuickDraw, QuickDraw GX, and QuickDraw 3D. All printing in Mac OS 8 uses QuickDraw GX, and all the graphics subsystems in Mac OS 8 use a common base of code.
- The multimedia services, including QuickTime and QuickTime VR.
- The human interface toolbox, which includes system services used by applications to implement the Mac OS user interface.
- The text-handling services, which include the functions that applications can use to display and manipulate text and other services to help you create world-ready software. Mac OS 8 text-handling services are a client of the graphics services.
- The interprocess communication (IPC) services, which include Apple events and AppleScript. These services are described in more detail in "Synchronization of Tasks and Coordination of Processes" (page 1-20).

Figure 1-1 Constituents of Mac OS 8



Tasks, Processes, and Multithreading

The Mac OS 8 microkernel includes functions to create and delete tasks and processes. A **task** is the basic unit of program execution in Mac OS 8. Every task has its own stack and set of register values. The actual code executed by a task might be in memory or on disk. The code might be provided by the program itself, or it might be part of a shared library used by the program. The microkernel keeps track of the location of the code. The microkernel schedules tasks, determining when one should stop execution and another begin. The microkernel services include routines to create and terminate tasks, and to assign an execution priority to a task. The execution priority is one of the criteria the microkernel uses to determine which task should be executing.

A **process** comprises one or more tasks and the memory and other operating system resources allocated to those tasks. The microkernel uses processes to track the resources required by tasks and to recover those resources when a process terminates. Every process has at least one associated task and can have several tasks. A given task can be associated with only one process, however.

When you write a program, you can write it to execute as one or more tasks. When Mac OS 8 launches the program, the microkernel and Process Manager prepare the code to run and associate it with a process. Mac OS 8 uses the process to allocate, track, and deallocate the system resources needed to run the program. When first prepared to run, a process has a single task, called the **main task.** The main task can create other tasks, those tasks can also create tasks, and so forth. All the tasks created in this way belong to the same process as the main task.

You can also write a software product that executes as more than one process. For example, you could write a software product that consists of two programs: a newswatcher program that monitors a newswire service for articles that contain specific keywords, and a newsreader program that can be used to read an article. The newswatcher program might have no human interface, and would execute as one or more tasks assigned to a single process. When the newswatcher program found an appropriate article, it could send a message to the news reader program. The newsreader program would provide a user interface and could execute as one or more tasks assigned to their own process. Thus, the newswatcher and newsreader programs could be launched and run independently; at any given time, either or both programs could be running on a given computer.

A program can follow a single path of execution through the code, or it can be written so that more than one path of execution can be running at a time. Each such path of execution is known as a **thread.** For example, the newsreader program could contain one thread to handle user interactions and another to read a saved news article from disk. By running these threads concurrently, Mac OS 8 increases the efficiency of the program. User interactions are normally sporadic; whenever the user is not actually using a pointing device or pressing keys, the program could be reading in data. The user perceives the program as reading the data in the background without affecting the responsiveness of the program. A program that has more than one concurrent thread is said to be **multithreaded.**

In Mac OS 8, you can use any combination of three techniques to multithread your software: you can create more than one task, you can use the Cooperative Thread Manager to thread a single task, and you can write your software to execute as more than one process.

The most efficient means of multithreading is often multitasking. In **multitasking,** several tasks are in progress at the same time, in the sense that they all have begun execution. Although (on a single-processor system) only one task can be executing at a time, Mac OS 8 switches from one task to another with such rapidity that the user has the impression that the tasks are executing simultaneously. The microkernel keeps track of the state of the CPU, including the location of the executing task's variables, for each task. This information is known as the task's **context.** The microkernel switches the context each time it preempts one task and gives control to another.

In **cooperative multithreading,** a single task is divided into more than one thread of execution by calls to the Cooperative Thread Manager. The use of cooperative multithreading is restricted to a certain type of task: the main task of a cooperative program. Cooperative programs are discussed in "Cooperative Scheduling" (page 1-10).

Writing your software as more than one program, so that each program is executed as a separate process, is especially useful when the different parts of your product do not always have to be available for execution at the same time. In the newswatcher and newsreader example discussed earlier in this section, for instance, the newswatcher program could run in the background whenever the computer was turned on, even if the newsreader application had not been prepared for execution. The newswatcher program could display a dialog box whenever an article of interest came in, and the user could then start up the newsreader application to read it.

Terminology note
Although the terms *thread* and *multithreading* are in wide use throughout the computer industry to refer to paths of execution in general, in Mac OS 8 and the *Inside Macintosh* suite of books these terms are normally used only to refer to threads created through the Cooperative Thread Manager. Furthermore, in Mac OS 8, the terms *process*, *task*, and *thread* imply a strict hierarchy: processes include tasks, which can include threads. This usage might not conform to the use of these terms in other operating systems or in books other than *Inside Macintosh*. ◆

Program Scheduling and Preemption

The microkernel schedules all tasks within the system preemptively according to their priority and eligibility to execute. If the program is an application—that is, a program that has a human interface component—the Process Manager schedules the main task cooperatively. This section discusses the concepts of preemptive and cooperative scheduling.

Preemptive Multitasking

The Mac OS 8 microkernel schedules all tasks preemptively. In **preemptive multitasking**, the microkernel can interrupt, or preempt, the execution of a task at any time. Preemptive multitasking allows Mac OS 8 to maximize system performance and responsiveness to the user. The microkernel can preempt a running task at any time to allow another task to execute. For example, if a task is reading in data from a disk and the user presses a key, the microkernel might preempt the task that's reading from disk and give control to the task that has to respond to the user. To the user, the disk would appear to be read in the background, without degrading the application's responsiveness.

The exact algorithm used by the microkernel to determine which task should execute is proprietary and subject to change. In general, however, if a low-priority task is executing and another task of higher priority becomes eligible, the microkernel preempts the executing task and passes control to the higher-priority task. If there are two or more eligible tasks with similar priority, the microkernel might give each task a specific amount of time (referred to as a

time slice) to execute. When the time slice expires, the microkernel preempts the running task and passes control to another eligible task. The microkernel also has special rules to handle tasks whose execution time should not be restricted to a time slice, such as real-time tasks (for example, those used for playing movies and animations) and some operating system tasks.

If a task is waiting for some event to occur, such as a keystroke from the user or an interrupt from an I/O device, then the microkernel considers that task no longer eligible for execution. The microkernel suspends that task and passes control to another eligible task. While a task is waiting for an event to occur, the task is said to be **blocked.** After that event occurs, the blocked task again becomes eligible for execution.

Cooperative Scheduling

Preemptive multitasking requires no special programming techniques for the vast majority of code. However, in order for preemptive multitasking to work, it must be possible for the microkernel to interrupt a task at any time. Therefore, special provisions must be made for tasks that make calls to non-reentrant system services. A **reentrant** service is one that can be interrupted and then resume execution where it left off, and that can be used concurrently by several tasks. If a task calls a reentrant service and is preempted, the service can resume execution when the task resumes control as if the interruption never happened—even if that service has been called by other tasks in the meantime. However, if a task calls a non-reentrant service, that service must not be called again until the current call has completed execution.

To make tasks that call non-reentrant system services work in the preemptive multitasking environment of Mac OS 8, the Process Manager schedules all such tasks cooperatively. In **cooperative scheduling,** the Process Manager blocks the main tasks of all but one cooperative program from execution at any given time. A **cooperative program** is one whose main task calls non-reentrant system services. Only when that main task voluntarily relinquishes control does the Process Manager block that task and make another such task eligible for execution. (These programs are called cooperative because they must cooperate by voluntarily relinquishing control.) The microkernel can preempt the main task of a cooperative program, but the Process Manager sees to it that no other such task is eligible for execution until the one that was executing relinquishes control.

Because major portions of the human interface toolbox are not reentrant, any task that makes use of Macintosh human interface elements, such as windows and menus, has to be cooperatively scheduled by the Process Manager. You can use the Cooperative Thread Manager to thread a cooperatively scheduled task. The threads created by the Cooperative Thread Manager for a particular task can execute only when that task has been made eligible to execute by the Process Manager and has been selected for execution by the microkernel. You can also thread a cooperative program by creating additional tasks; however, only the main task of such a program can call non-reentrant system services.

Applications and Server Programs

Most of the non-reentrant system services in Mac OS 8 are part of the human interface toolbox. The human interface toolbox is used to provide a program with the familiar Macintosh graphical user interface (GUI). Any program with a GUI must therefore be scheduled cooperatively. In this book and other books of the *Inside Macintosh* suite, the term **application** refers specifically to a program that calls the non-reentrant system services; that is, any program with a GUI.

Note
Because only cooperatively scheduled tasks can call the non-reentrant Mac OS 8 services, these services are sometimes referred to as **cooperative system services.** ◆

As for all programs, when Mac OS 8 prepares an application for execution, it creates a single task: the application's main task. The Process Manager schedules an application's main task cooperatively. You must make sure that your application's main task strictly follows all programming guidelines for cooperative software. An application's main task can create other tasks, but these tasks can call only reentrant system services; all calls to non-reentrant system services must be in an application's main task. Every function description in *Inside Macintosh* specifies whether the function is reentrant.

You can also write programs that do not call any non-reentrant services; that is, that do not have any graphical user interface. None of the tasks in the process, including the main task, can call non-reentrant services. A program that doesn't call non-reentrant services (and that therefore has no graphical user interface) is called a **server program.** Server programs can be network or file servers but can also be any other program that has no GUI. Server programs can provide services to applications or background services for a user.

Virtual Memory

Mac OS 8 uses a virtual memory system to provide large amounts of memory for use by programs. Virtual memory is always enabled. Most kinds of software need never make a call to the virtual memory system in order to work with Mac OS 8.

Logical memory in Mac OS 8 is organized into multiple address spaces of 2^{32} bytes (4 GB) each. At least 1 GB of logical addresses are available for program use in each address space. Because few, if any, Mac-compatible computers contain several GB of physical RAM, the virtual memory system maps between the logical addresses assigned to code and data, the physical RAM available in a particular computer, and locations on disk (or some other storage device).

The Mac OS 8 virtual memory system is demand paged. In **demand paging,** a page of code or data is read from disk only when it is actually needed by software running on the computer. When an executing task needs data or code that is not in RAM, the hardware generates a **page fault,** an exception for which the microkernel provides a handler. The handler resolves the page fault by calling a **backing provider,** which makes space in RAM and reads in a set amount of data (referred to as a **page**) from a storage device. Mac OS 8 provides a backing provider that uses a hard disk as the storage device. Other backing providers can be written to use other devices, such as compressed RAM disk.

There are two ways the backing provider can make space in RAM for the new page: by writing the data in RAM out to storage, or by simply writing over the page in RAM.

The area of disk or other storage used to store data paged into and out of RAM is referred to as **backing store**. To use as little disk space as possible for backing store, the Mac OS 8 virtual memory system memory-maps code files on disk. A **memory mapped file** is a disk file associated with a memory area so that the virtual memory system maps logical addresses directly to the file and reads the information on disk into RAM only as it is needed. Code files are always read-only; when the backing provider needs the space that program code occupies in RAM, it simply reads the new code or data from disk and writes over the page in RAM. Because the program file cannot be modified and is always read directly from disk (rather than from a scratch file), there is no need to write it back to disk.

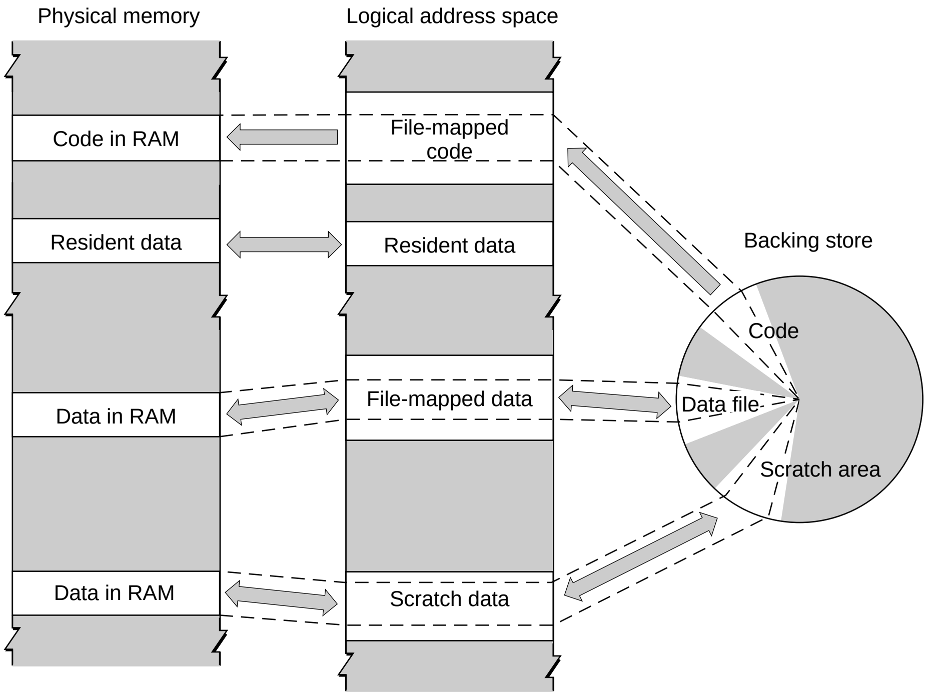
Because data can be modified, it is not automatically memory mapped. However, you can create a memory area that is memory mapped. In that case, rather than allocating a separate portion of the disk for scratch files, the virtual memory system reads a page of data directly from a disk file into RAM as it does for code. Unlike code, however, if data is modified, it cannot simply be purged from memory. Instead, when it pages memory-mapped data back to disk, the virtual memory system writes it back directly into the same file from which it was read.

If you have data that you know won't change, you can create a memory area that is memory mapped and has read-only access. In that case, the virtual memory system treats the data like it treats code: it purges the data from memory rather than write it back to disk when it needs the memory for something else.

Figure 1-2 illustrates the relationships among physical memory, logical memory, and backing store. Because code files are memory mapped, a file on disk is mapped to a specific location in logical memory and read into RAM as needed. The code is never written back out to backing store. Code and data are read into RAM as needed. For a memory-mapped file, the file on disk is mapped to a specific area in logical memory. All of the data on disk has corresponding locations in logical memory, but not all the data need be in physical memory at any one time.When the space in RAM is needed for something else, the data is written back to the file on disk. For data that is not memory mapped, the data in RAM is mapped to logical address space. When the space in RAM is needed for something else, the data is written to a scratch area on backing store. It can be read back in from the scratch area when it is needed. If you want to save the data, you must write it to a file on disk as a separate operation.

Note
Figure 1-2 is conceptual only and does not show the structure of the memory. For example, the data in RAM need not be contiguous or all in the same portion of memory. The sequence of memory areas in physical and logical memory need not be the same. Files on disk need not be contiguous or all stored in the same area of the disk. ◆

Figure 1-2 Virtual Memory



When a task triggers a page fault, that task becomes blocked while it waits for the data to be paged from disk. The microkernel then passes control to another task until the page fault is resolved.

Some programs, such as device drivers and games doing real-time animation, cannot tolerate page faults. Mac OS 8 provides routines that allow such software to create memory areas that remain resident in physical memory. As shown in Figure 1-2 a resident memory area has corresponding locations in physical and logical memory, but it has no backing store. Mac OS 8 also provides routines that allow you to temporarily lock memory pages as necessary. A locked memory page has corresponding backing store, but cannot be paged out of RAM as long as it is locked.

Compatibility note
Although virtual memory is always enabled in Mac OS 8, the Gestalt Manager always tells you that virtual memory is off. If you make any calls to the System 7 Virtual Memory Manager, Mac OS 8 does nothing except to return the `noErr` result code. Because System 7 applications use the result returned by the Gestalt Manager to determine whether to make calls to the System 7 Virtual Memory Manager, and because in Mac OS 8 those calls do nothing, a negative result from the Gestalt Manager prevents such programs from making unnecessary calls. ◆

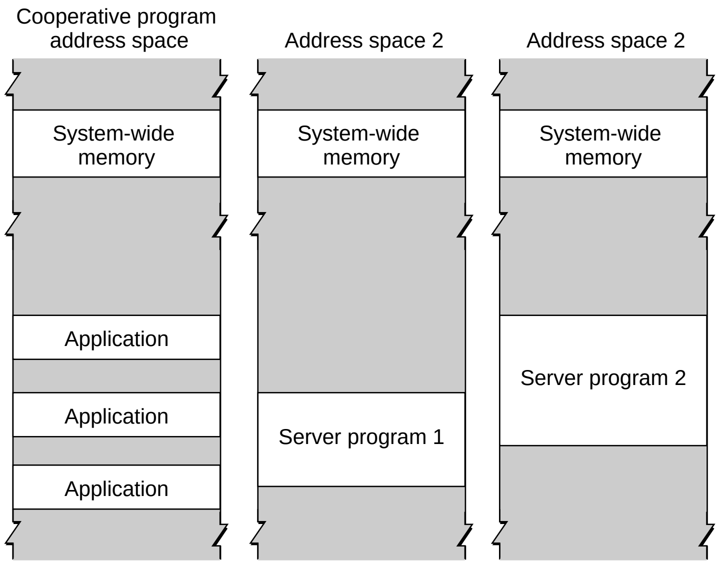
Memory Organization and Protection

The Mac OS 8 logical memory is divided into address spaces of 4 GB each. At any given time the system can access data in only one address space. Therefore, a task running in one address space cannot read from or write to memory in any other address space. This arrangement protects code and data in one address space from corruption by tasks running in other address spaces, but it also makes it harder for programs and Mac OS 8 to communicate and to share data. Several features of the memory organization in Mac OS 8 address this problem and enhance memory protection.

Multiple Address Spaces

To protect programs from corruption by other software, every server program runs in its own address space. Because the Process Manager must coordinate the operation of all cooperative programs, which share low memory and a common system heap, all cooperative programs share a single address space. Figure 1-3 shows a highly simplified address map illustrating these principles.

Figure 1-3 Multiple address spaces



Each address space includes some areas that are also mapped into other address spaces, such as code, and some data that is mapped to that address space only, such as the stack used by a server program. When Mac OS 8 preempts a task and switches to one running in another address space, it must perform an address space switch. By contrast, when Mac OS 8 runs microkernel code, it does not have to perform an address space switch, because the microkernel code and data are mapped into all address spaces.

Memory Efficiency

Mac OS 8 makes efficient use of physical memory. For example, when the Process Manager launches an application, it allocates space in the cooperative program logical address space for all the code and data needed by that application. However, not all of that code and data are read into physical memory at once. Instead, Mac OS 8 reads pages of code and data into physical memory only as they are needed. When that code or data is no longer needed, the OS can page it out or overwrite it.

Mac OS 8 dynamically increases and reduces backing store as it launches and terminates programs. In addition, when Mac OS 8 launches programs that are written to take maximum advantage of Mac OS 8 memory management, it allocates only as much backing store as it needs for those programs. This combination of demand-based loading of physical memory and minimal allocation of backing store results in a highly efficient use of the computer's resources.

Memory Areas

Each address space can contain both code and data. To minimize address-space switching, code or data can be mapped into more than one address space. For example, the code and data of the microkernel are mapped into every address space, and all code for all processes is generally mapped into every address space. To control access to specific portions of memory, each address space is subdivided into memory areas. A **memory area** is a range of logical addresses within an address space to which the creator of the area can assign certain attributes. Memory area attributes specify who (all code or privileged code only) can read from and write to the area, whether the data in the area can be paged out or must remain in physical memory, whether the area should be globally mapped to all address spaces, and other characteristics of the area. The microkernel code area, for example, is globally mapped but always read-only. The microkernel data area is globally mapped and can be written to only by privileged code.

Privileged code

Privileged code is code that is executed while the CPU is in supervisor mode. **Supervisor mode** is a state of operation of the PowerPC processor, enforced by the internal logic of the processor, that allows access to certain critical resources, such as all processor instructions and tables that control memory protection. Certain processor instructions can be executed only by privileged code, and memory areas can have different read/write access for privileged and nonprivileged code. In Mac OS 8, only the microkernel, portions of device drivers, and certain other portions of the operating system are privileged. ◆

When Mac OS 8 launches a program, it allocates memory areas for the program's code and data. The code memory area is shared across all address spaces but is read-only. The data memory areas are read/write and are local to a single address space. Each program gets memory areas for its static data and for dynamic storage, and each task gets its own stack, used during the execution of certain routines. For a cooperative program not written to take advantage of Mac OS 8 memory management, the heap is a set size and cannot grow. For a program that takes advantage of Mac OS 8 memory management, the heap is the minimum size needed and grows dynamically as the program allocates memory. You are not restricted to the heap provided at launch time; any program can request additional memory areas to use for dynamic storage.

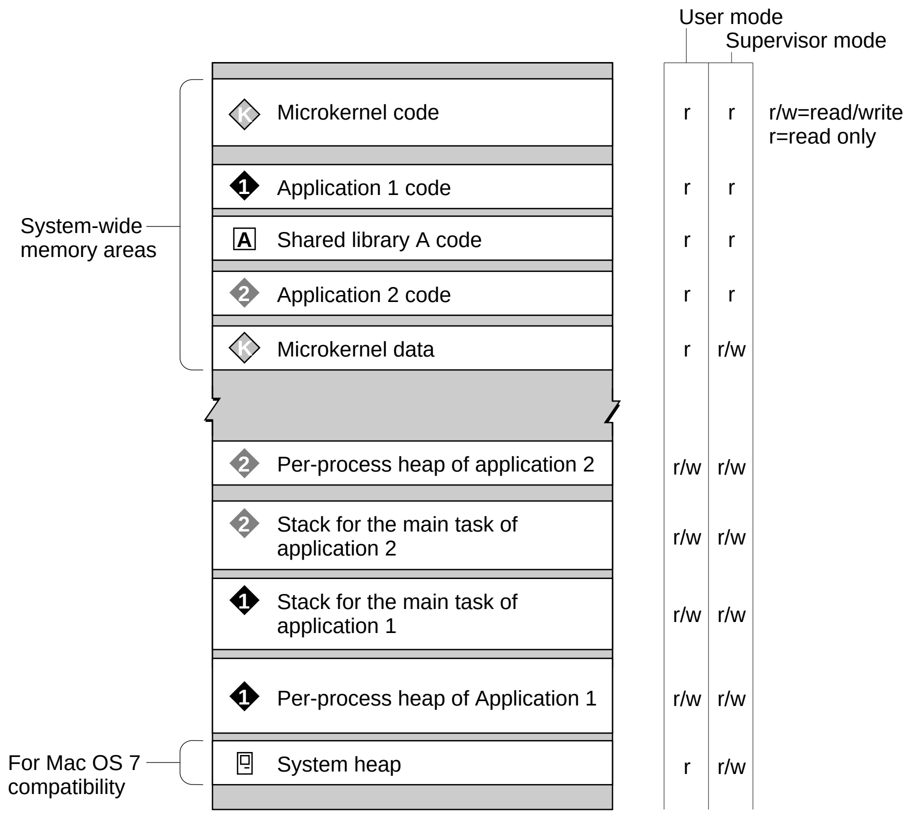
If you want to share data with another program, you can request a shared memory area and specify into which address spaces this area should be mapped. For a shared memory area, you can specify different access permissions for each address space into which that area is mapped. For example, if you want to make data available to a program running in another address space but don't want that program to be able to corrupt the data, you can specify that the memory area be mapped into that program's address space but that the area be read-only in that address space.

Figure 1-4 illustrates some of the memory areas in the cooperative address space. As shown in the figure, a program's code, heap, and stack are in separate memory areas and there is no requirement that these memory areas be adjacent or even in the same region of memory. The microkernel's code and data are mapped into every address space. The address space for cooperative programs also has a system heap, used to maintain compatibility of programs written for System 7.

All PowerPC native software executing in Mac OS 8 is packaged as code fragments. A **code fragment** is a block of executable code and its data, structured in such a way that the Code Fragment Manager can dynamically link all references to other code fragments when they prepare the fragment for execution. Every code fragment is potentially shareable by other code fragments. A code fragment that is shared by other fragments is called a **shared library**. Because shared libraries are dynamically linked to other code at runtime, they are also referred to as **dynamically linked libraries**.

If any program in a given address space is linked to a shared library, the library's code and data are mapped into that address space. Because the code is shared, only one copy of the code is mapped into memory, no matter how many programs call the code. Depending on how the shared library uses its variables, Mac OS 8 might maintain a separate data area for the shared library for each program that calls that library, or it might allocate a system-wide shared memory area for the library's data.

Figure 1-4 Memory areas in the cooperative program address space

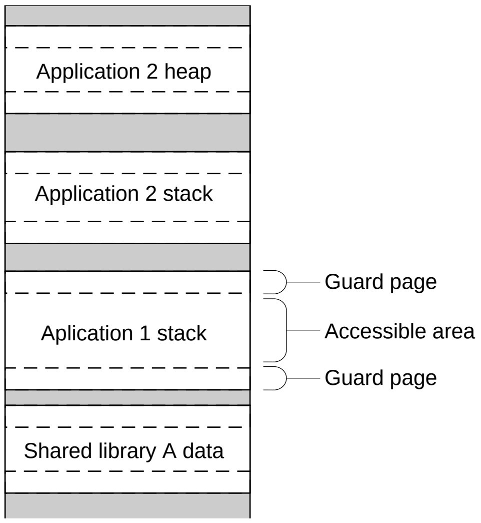


Guard Pages

Because all cooperative programs share a single address space, some additional protection is needed for these programs. Mac OS 8 provides a guard page mechanism to protect specific memory areas. A **guard page** is a page of logical memory addresses located just at the beginning or end of a memory area. A guard page has excluded permission; that is, no software, including privileged software, can read from or write to the guard page. If a task attempts to access a location within a guard page, the microprocessor generates an exception. The guard space at the beginning or end of a memory area can consist of one or more guard pages.

When the Process Manager launches a cooperative program, it allocates guard pages around that application's memory areas. The guard pages prevent one application's data from overflowing and overwriting data belonging to another application. They cannot, however, prevent an error in one application from causing it to write data directly into the memory area of another application. Figure 1-5 illustrates a portion of the cooperative program address space, showing guard pages.

Figure 1-5 Guard pages



Synchronization of Tasks and Coordination of Processes

In a preemptively multitasking environment, it is possible for more than one task to seek access concurrently to the same data or code. The tasks can belong to the same process, or can be in different processes and different address spaces, as long as the code or data they share is mapped into their respective address spaces. Because a task can start to read or write data or execute code and be preempted before it finishes, it is possible for a second task to start writing to the same data or executing the same code before the first task finishes. To prevent such events from causing problems, Mac OS 8 provides several mechanisms for synchronizing tasks.

If your code uses only local variables and parameters that do not point to shared data, and calls only reentrant services, then it is reentrant and does not need to use these synchronization mechanisms. If your task uses data that can be accessed by other tasks, then to make it reentrant you must call only reentrant services and you must use the Mac OS 8 synchronization services. Even if you call non-reentrant system services, it is recommended that you synchronize access to data so that your code can more easily make the transition to future, fully reentrant versions of Mac OS. Synchronization mechanisms that you can use to prevent concurrent access to code or data include atomic operations, locks, and counting semaphores.

Other types of coordination or synchronization among tasks and processes might be necessary as well. For example, an application might activate a user notification or enable some menu items only when a server program notifies it that some operation is complete; or one task might not be able to start execution until another task completes execution. Mac OS 8 provides a variety of services that you can use to coordinate the execution of tasks, including event flags and groups, microkernel queues, and software interrupts. To send more complex types of information between tasks, you can use interprocess communication (IPC) services, including microkernel queues, microkernel messages, and Apple events, as discussed in “Notification Services” (page 1-24).

Synchronizing Access to Code or Data

Atomic operations are the simplest way to synchronize access to specific memory locations, such as a single word in a data structure. Atomic operations perform specific, limited operations, such as adding a value to a value in memory or performing an OR operation on a value you specify and a value in memory. No other task can write to or read from the memory location being acted on until the atomic operation is complete. Because atomic operations are small and very efficient, they execute extremely quickly.

If you need to control access to a block of code—perhaps because it is not reentrant—or to a block of data larger than can be protected by atomic operations, you can use locks. **Locks** are a set of functions that enable you to implement a synchronization protocol. It is important to understand that

Mac OS 8 does not enforce a lock in any way. To lock access to a data structure, for example, you call a lock function before writing to or reading the data structure, and call another function to unlock the data when you are through. Any other task following this synchronization protocol also calls a lock function before trying to access the data. If one task has already called such a function ("acquired a lock"), the second task is blocked until the first task completes (alternatively, you can call a function that returns an error rather than blocking). However, nothing in the lock services prevents a task from reading or writing to the data without first calling a lock function, thus violating the protocol. Such a programming error could be fatal to a program.

You can use locks to prevent all access to a block of code or data, and you can use read/write locks to allow one task to read from and write to a block of data while multiple tasks can read it but not write to it.

Counting semaphores is another service you can use to implement a synchronization protocol. Counting semaphores are especially useful when one or more tasks (called *producer tasks*) are producing data or performing services that are consumed by other tasks (called *consumer tasks*). You call the semaphore service to create a semaphore, specifying an initial count (which is usually, but need not be, 0). The producer task or tasks then signal the semaphore each time they have performed a specific operation, such as making data available for the consumer tasks. If no consumer tasks are waiting for the semaphore, then the semaphore service increments the semaphore count.

From the consumer point of view, a consumer task calls the counting semaphores service to find out if a semaphore has been signaled. If the semaphore count is 1 or greater, the service decrements the count and the task continues to execute. If the semaphore count is 0 or negative, the task blocks until the producing tasks have incremented the semaphore to 1. If two or more consumer tasks are waiting for the semaphore, the task that has been waiting longest is unblocked.

The counting semaphores service does not pass any data. You must implement your own protocol to obtain the data or interpret the meaning of a semaphore signal.

Coordinating Tasks and Processes

The microkernel provides a fundamental mechanism for coordinating tasks, based on sets of flags called **event flags.** To use the event-flag mechanism, one task calls a microkernel function to create a set of 32 event flags, called an **event**

group. The microkernel returns an event-group ID. It is up to you to decide the meaning of each flag for your program. Any task can clear or set any of the flags in an event group, and any task can wait for an event or set of events to occur. A task that is waiting for events is blocked. The task can specify whether one or some combination of event flags must be set before that task becomes unblocked. You could use event flags, for example, to implement a state machine involving several tasks. Each task would become eligible for execution only when certain other tasks had set the appropriate flag.

Event flags are useful when a task must wait until one or some combination of events occurs before the task executes, and the task does not need to receive any data with notification of the occurrence of the events. Event flags do not distinguish between one occurrence of an event and several; thus if you must queue several similar events and act on each one in turn, or if you must pass data with an event, you should use one of the other coordination mechanisms, such as microkernel queues.

Like event flags, **microkernel queues** also allow a task to wait for an event; however, in the case of microkernel queues the event is the placing of an entry in the queue. When a task places into a queue an entry for which one or more tasks are waiting, the microkernel unblocks the task that has been waiting longest. The queue entry can contain three words of data; it is up to you to decide the meaning of that data. For example, the first word could specify the type of event that occurred and the other two words could convey specific information about the event. Because notifications are queued, your task can distinguish among multiple occurrences of the same event. Unlike event flags, however, which can unblock any number of tasks, the arrival of an entry in a microkernel queue unblocks only the single task that has been waiting longest.

Microkernel queues are slower and require more resources than event flags. However, whereas event flags are limited to 32 distinguishing events, the data passed by microkernel queues allows them to distinguish among an unlimited number of types of events, or among multiple occurrences of the same event. Microkernel queues provide an efficient, low-level mechanism for coordinating and ordering tasks, and allow you to pass limited amounts of data.

Both event flags and microkernel queues block a task until an event occurs. If you do not want to create a separate task to wait for an event, you can use a software interrupt instead. A **software interrupt,** as its name implies, is an interrupt generated by software and used to interrupt a task. A software interrupt can be originated by any task. To receive software interrupts, you write a software interrupt handler and call a microkernel function to specify the task with which it is to be associated; the microkernel returns a software interrupt ID. You can pass that ID to any task in any address space, and that task can then send a software interrupt to your task at any time.

A software interrupt executes in the context of the task to which it is sent. When a task receives a software interrupt, the task's execution is interrupted and the software interrupt handler executes. When the handler completes, the task resumes. Receiving a software interrupt does not change a task's execution priority. However, if the task is waiting for execution, or if the task has disabled its ability to receive software interrupts, the software interrupt is queued and executes as soon as the task can receive it. Because software interrupts execute in a task's context, they have access to the task's data (which you should access only with care). Like other interrupt handlers, however, a software interrupt handler must never call non-reentrant system services, even if it is okay for the interrupted task to do so.

Notification Services

Mac OS 8 provides several ways for the system to communicate with tasks and for tasks to communicate with each other or with the user. When the communication is simply an indication that some event has occurred, the service is called a *notification*. When the purpose of the communication is to pass information between tasks or from the system to a task, the service is called *interprocess communication* (IPC).

You can use notification services to coordinate the activities of tasks and processes, to provide feedback to users when some event occurs, and to coordinate your program with events that have taken place elsewhere in the system. Notification services include microkernel notification, the System Notification Service, and the Notification Manager. You can use IPC services to send information from one task to another. IPC services include microkernel queues, microkernel messages, and Apple events. In many cases, you might wish to send data along with notification that an event has occurred; therefore, the functions and purposes of the various notification and IPC services overlap somewhat.

Microkernel Notification and Asynchronous Execution

The Mac OS 8 APIs include many functions that you can call asynchronously; that is, they return control to your task immediately and then send notification to the calling task when they complete. When you call an asynchronous function, you provide a microkernel notification structure. This structure lets you specify that you want to use any combination of three notification mechanisms: microkernel queues, event flags, or software interrupts. In addition, you can use a special microkernel notification structure to use Apple events as the notification mechanism.

The preferred notification and interprocess communication mechanism in Mac OS 8 is Apple events. To use Apple events with asynchronous functions, you call an Apple event routine that creates a microkernel notification structure for you. When you pass that microkernel notification structure to an asynchronous function, Mac OS 8 sends an Apple event to notify you when the routine completes. Apple events are discussed further in “IPC Services” (page 1-26). If you have specific needs not fulfilled by Apple events, you can use one of the other, lower-level notification services.

Microkernel queues and event flags are simple and efficient. Both of these mechanisms require that you either check periodically to see if the event has occurred, or wait for the event, causing your task to block while waiting. By contrast, you can use software interrupts to receive notifications without polling for events or causing your task to block. To use software interrupts, however, you have to provide a software interrupt handler. Both microkernel queues and software interrupts pass a limited amount of data along with notification and include the function result of the routine you called asynchronously. Event flags don’t pass any data beyond the values of the flags themselves. All three of these notification mechanisms are discussed in “Coordinating Tasks and Processes” (page 1-22).

System Notification Service

The System Notification Service provides a mechanism for Mac OS 8 and other service providers, such as third-party server programs, to broadcast to tasks notification of specific events. Any task can register to receive notification for specific types of events. The service that provides notification is called a **notification provider**, and the task that receives the notification is called a **consumer**. To receive events, you must know beforehand for what types of events notification is available. Types of events currently available include the death of a process or task and changes in files and file directories.

The System Notification Service provides synchronous and asynchronous functions that you can use to receive and process (“consume”) a notification. These functions return the type of event that occurred and data associated with the event, such as the old and new names of a file in the case of a filename change. Because the System Notification Service informs you when an event has occurred, you do not have to poll to get this information. For example, if you need to know whether the user has changed the name of a file that your application has open, you can register with the System Notification Service and call the consume function to receive that information. Without the System Notification Service, you would have to poll the File Manager periodically to get the current name of the file.

User Notification Services

Whereas you can use the System Notification Service to receive notification of system events, you can use the Notification Manager to inform users of events of interest to them. The Notification Manager is a reentrant service that can be used both by applications and server programs to inform the user that something has happened. You can cause an icon to blink in the menu bar, the Sound Manager to play an alert sound, an alert box to appear on the screen, or any combination of these. If you use the Notification Manager to display an alert box, you can include a brief message informing the user of the event that has occurred or asking the use to perform some action, such as bringing your application to the front. The Notification Manager does not return any information to the task that called it, except that you can determine whether the user has responded to the alert box by clicking the OK button.

IPC Services

There are several ways that tasks can communicate with other tasks both in the same process and in other processes. Any task can send or receive microkernel queues and software interrupts, and both carry limited amounts of data. They are described in “Coordinating Tasks and Processes” (page 1-22). There are two primary methods of sending larger amounts of data: microkernel messages and Apple events. Microkernel messages are a low-level service provided by the microkernel. Apple events are a high-level service widely used by applications and server programs for a variety of purposes.

Microkernel Messages

The Microkernel Messaging Service provides for delivery of data from one task to another, or to a special function called an *accept function*. An accept function is privileged software. Messages sent to accept functions never cause an address-space switch and never cause data to be copied from one address space to another. Therefore, such messages are very efficient. However, because an accept function executes in supervisor mode and has access to the operating system's data, it can potentially corrupt critical resources and crash the system. You should use privileged software only when absolutely necessary. The sending task does not know or need to know whether the entity receiving the message is a task or an accept function.

The receiving task or accept function must respond to each message it receives; the message and its reply form a transaction between the sender and receiver. The messaging service does not interpret the message data. It is entirely up to the sending task and the receiving task or accept function to determine the protocol by which the message is interpreted. If you are writing a shared software library or a server program for which you are providing an API, you should not expose the Microkernel Messaging Service to your clients. If you use microkernel messages, you should keep the details private and provide a high-level API for your clients.

You send a message to a message object rather than to the actual task or accept function that will process the message. A message object is an address associated with a message port. A message port is a message destination maintained by the Microkernel Messaging Service. A port can have multiple objects associated with it. When you send a message to a message object, the messaging service delivers the message to the port associated with that object. The receiving task or accept function retrieves the message from the port; the object to which the message was sent is included in the information the receiver gets. Message objects can be used in any way that makes sense to the program that creates them. For example, a file server can create a message object for each file that it opens. Then the sending task indicates which file it wants to read from or write to by sending the message to the message object for that file. In this example, the message's data would indicate the action to take (read or write) and would include the data to write or the location in the file from which to read.

The reply to a message can also carry data. In the file server example, the reply to a request to read data from a file could include the data that was read.

You can send the data in a message by reference or by value. If you send the data by reference, the data is mapped into the address space of the receiver. Then, if the receiver modifies the data, the sender's data is modified as well. You can use this method when it is necessary for the data used by the receiver and the sender to remain identical, especially if the receiver needs to examine or modify only a few values in a large block of data. By contrast, if you send the data by value, the data is copied into the address space of the receiver. In this case, if the receiver modifies the data, it modifies its copy only. You can use this method when you do not want the sender's data modified or corrupted by the receiver. You can also let the messaging service choose the fastest method.

The Microkernel Messaging Service is very fast and efficient. However, it is a very low-level operating system service; it has no standard data format. You can't use it to send data across a network from one computer to another. Because the programs using the messaging service must devise their own protocol for exchanging object IDs and other information needed to interpret the message, the messaging service is not suitable for sending messages from one program to another, unrelated program. If you need any of these high-level capabilities in an IPC messaging service, you should use Apple events.

Apple Events

Apple events are a high-level messaging service that allow a task to send messages to any other task, whether on the same computer or over a network. In addition, Mac OS 8 uses Apple events to send messages and notifications to tasks. Programs can use Apple events to send messages to themselves; in fact, doing so is essential if you want to make your application scriptable.

There is a standard set of Apple events that every application should handle so that it can respond to messages sent by other applications and by Mac OS 8. You can also define your own Apple events.

Apple events have a well-documented format; they're standardized and easy to incorporate in your code. They can be used both by applications and server programs, and they are used widely by Mac OS 8 system services. Apple events in Mac OS 8 are much faster than they were in previous versions of Mac OS. For these reasons, Apple events are the standard preferred method of interprocess communication in Mac OS 8.

Because Apple events are a high-level service, they are not described in this book. See *Apple Events in Mac OS 8* for more information about Apple events.

Glossary

accept function A special function that receives microkernel messages. An accept function is privileged software and runs in the privileged software memory area. There can be only one accept function for each microkernel messaging port.

address space The maximum number of address locations that can be physically addressed by the CPU. At any given time the system can access data in only one address space. Mac OS 8 provides multiple address spaces, each 4 GB in size. See also **memory area**.

application A stand-alone program with a graphical user interface. See also **cooperative program, part editor**.

atomic operations A set of specific, limited functions performed by the microkernel, such as adding a value to a value in memory or performing an OR operation on a value you specify and a value in memory. No other task can write to or read from the memory location being acted on until the atomic operation is complete. Atomic operations are small and very efficient.

backing provider Code responsible for managing pages of physical memory and transferring data (typically between backing store and physical memory) in response to page faults.

backing store A repository—typically a file on a paging device such as a hard disk—for pages of memory that aren't currently in physical memory.

blocked task A task that is not eligible for execution until a certain event occurs, such as the completion of a synchronous I/O operation.

code fragment A block of executable code and its data, structured in such a way that the Code Fragment Manager and Process Manager can dynamically link all references to other code fragments when they load and launch the fragment. See also **shared library**.

consumer A task that has registered with the System Notification Service to receive information about changes in the state of the system. See also **notification provider**.

context The state of the CPU while a specific task is executing, including the location of the executing task's variables. The microkernel switches the context each time it preempts one task and gives control to another.

context switch The suspension of a currently executing task and resumption of a different task from the point at which it was blocked. During a context switch, the microkernel saves the context of the suspended task and restores the context of the task about to resume execution.

cooperative multithreading A form of multithreading in which a single task is divided into more than one thread of execution. The use of cooperative multithreading is restricted to a certain type of task: the main task of a cooperative program. You must use the Cooperative Thread Manager to incorporate cooperative multithreading in your program. Compare **preemptive multitasking**.

cooperative program A program whose main task calls non-reentrant system services to present a graphical user interface. A cooperative program cooperates with other programs by voluntarily relinquishing control of the CPU. Cooperative programs are cooperatively scheduled by the Process Manager. See also **application**, **part editor**. Compare **server program.**

cooperative scheduling A policy for scheduling access to Mac OS 8 non-reentrant services implemented by the Process Manager. The Process Manager blocks the main tasks of all but one cooperative program from execution at any given time, thus ensuring that each call to a non-reentrant function can execute to completion without being preempted. See also **cooperative program, main task.**

cooperative system services Non-reentrant system services. These services support the Macintosh graphical user interface and can be called only by cooperative programs.

core system services Low-level system services outside of the microkernel. The core system services include some of the memory management services, the Process Manager, the Code Fragment Manager, the Cooperative Thread Manager, the Mixed Mode Manager, debugger services, the Component Manager, and a variety of utilities and other services.

counting semaphores A service used to implement a synchronization protocol. Counting semaphores are most useful when one or more tasks are producing data or performing services that are consumed by other tasks. The producer tasks signal the semaphore each time they have performed a specific operation, such as making data available for the consumer tasks. If no consumer tasks are waiting for the semaphore, then the semaphore service increments the semaphore count. However, if one or more consumer tasks are waiting for the semaphore, the task that has been waiting longest is unblocked instead. If a consumer task calls a semaphore and the semaphore count is 1 or greater, the service decrements the count and the task continues to execute. If a consumer task calls a semaphore and the semaphore count is 0 or negative, the task blocks until the producing tasks have incremented the semaphore to 1.

demand paging A type of virtual memory in which a page of code or data is read from backing store into memory only when actually needed by software running on the computer.

dynamically linked library (DLL) See **shared library.**

event flags The individual bits in an event group.

event group A set of 32 event flags that can be used as a fundamental mechanism for coordinating tasks. Any task can clear or set any of the flags in an event group, and any task can block until one or more flags are set. It is up to the participating tasks to agree on the meanings of the flags in an event group.

file mapping See **memory-mapped file.**

guard page A page of memory given excluded permission, so that no tasks can read from or write to the page. Guard pages can be placed at the beginning and end of a memory area to protect it from corruption. If a programming error causes a task to attempt to reference a guard page, the CPU generates an exception. The guard space at the beginning or end of a memory area can consist of one or more guard pages.

kernel A program that manages all or most of the operating system services necessary to control a computer. In a UNIX-based operating system, for example, the kernel is a program that supervises task and file management, device input and output, and memory allocation. Compare **microkernel.**

locks A service used to implement a synchronization protocol. To lock access to a data structure, you call a lock function before writing to or reading the data structure, and call another function to unlock the data when you are through. Any other task following this synchronization protocol also calls a lock function before trying to access the data. If one task has already called such a function (“acquired a lock”), the second task is blocked until the first task completes (alternatively, you can call a function that returns an error rather than blocking).

Mac OS 8 system services All of the services provided by Mac OS 8, including the operating system services, the I/O system, the graphics system, text, and the human interface toolbox.

main task The first task created by Mac OS 8 for a process. The main task can create other tasks, those tasks can also create tasks, and so forth. The main tasks for cooperative programs can safely use Mac OS 8 non-reentrant services, whereas all other tasks in Mac OS 8 must use only reentrant services.

memory area A range of logical addresses within an address space to which the creator of the area can assign certain attributes.

memory mapped file The association of a disk file with a memory area so that the file's data is paged between physical memory and the file's permanent location on disk. Thus, the disk version of the file (instead of a separate scratch file) serves as backing store for the file's representation in memory.

message See **microkernel message**.

message object An address associated with a message port. You send a microkernel message to a message object rather than to the actual task or accept function that will process the message.

message port A message destination maintained by the Microkernel Messaging Service, but belonging to a specific process. When you send a message to a message object, the messaging service delivers the message to the port associated with that object. The receiving task or accept function retrieves the message from the port; the object to which the message was sent is included in the information the receiver gets. See also **message port**.

microkernel A program that manages a small but critical subset of the operating services necessary to control a computer. The Mac OS 8 microkernel, for instance, manages processes, their attendant tasks, and other operating system resources associated with tasks, such as memory, synchronization, timing, and messaging. Other operating system services, such as the file system, the I/O system, and the human interface toolbox, are implemented separately from the microkernel. Compare **kernel.**

microkernel message A message sent by the Microkernel Messaging Service.

Microkernel Messaging Service A microkernel service that provides for delivery of a string of bytes from one task to another, or to a special function called an accept function. The receiving task or accept function must reply to a message.

microkernel queue A mechanism by which one or more tasks notify another task of some event. The task waiting for notification blocks until the event occurs. When a task places an entry into a queue for which one or more tasks are waiting, the microkernel unblocks the task that has been waiting longest for the queue. The queue entry can contain three words of data; it is up to the communicating tasks to decide the meaning of that data.

multitasking The situation in which several tasks are in progress at the same time, in the sense that they all have begun execution. Although (on a single-processor system) only one task can be executing at a time, Mac OS 8 switches from one task to another with such rapidity that the user has the impression that the tasks are executing simultaneously. See also **multithreaded.**

multithreaded Having more than one concurrent path of execution. For instance, one thread in a multithreaded program might handle user interactions, another thread might perform calculations, and yet a third might perform I/O. See also **thread.**

notification provider A service that uses the System Notification Service to provide notification of changes in the state of the system. See also **consumer**.

OpenDoc A multiplatform technology, implemented as a set of shared libraries, that facilitates the construction and sharing of compound documents. See also **part.**

operating system services Any services provided by Mac OS 8. Compare **core system services**, **microkernel**.

page (1) A unit, measured in bytes, of the information that may be read from and written to an I/O device. (2) To transfer pages between physical memory and backing store.

page fault An exception that causes a page of data or code needed by a program to be read from backing store into physical memory.

part A portion of an OpenDoc compound document. A part consists of document content, plus—at execution time—a part editor that manipulates that content. The document content is data of a given structure or type, such as text, graphics, or video. In programming terms, a part is an object, an instantiation of a subclass of the class ODPart. To a user, a part is a single set of information displayed and manipulated in one or more frames or windows.

part editor An OpenDoc component that can display and change the data of a part. It is the executable code that provides the behavior for the part. An OpenDoc part editor's main task can call non-reentrant system services to present a graphical user interface. Like other cooperative programs, an OpenDoc part editor that calls non-reentrant services must cooperate with other programs by voluntarily relinquishing control of the CPU. Cooperative programs are cooperatively scheduled by the Process Manager. Compare **application, server program.**

preemptive multitasking A policy for allocating access to the CPU and other operating system services among multiple tasks. The preemptive multitasking environment of Mac OS 8 uses a set of well-defined rules to determine which task should execute. Following these rules, the microkernel can interrupt, or preempt, the execution of a task at any time and resume the execution of another. Compare **cooperative multithreading.**

privileged code is code that is executed while the CPU is in supervisor mode. Certain processor instructions can be executed only by privileged code, and memory areas can have different read/write access for privileged and nonprivileged code. In Mac OS 8, only the microkernel, portions of device drivers, and certain other portions of the operating system are privileged.

process An instance of a program at execution time. A process comprises one or more tasks and the memory and other operating system resources allocated to those tasks. The microkernel uses processes to track the resources required by tasks, and to recover those resources when a task completes execution. Every process has at least one associated task, and can have several tasks. A given task can be associated with only one process, however.

Process Manager A Mac OS 8 service that launches, manages, and terminates processes. On behalf of programs using non-reentrant system services, the Process Manager also synchronizes use of these services.

reentrant Code that can interleave multiple requests for service and process these requests in any order. For example, a reentrant function can begin responding to one call, become interrupted by other calls, and complete them all with the same results as if the function had received and completed each call serially.

server program In Mac OS 8, a program that calls no non-reentrant services (and that therefore has no graphical user interface). A server program runs in its own protected address space. Server programs typically provide services to other programs. Compare **cooperative program.**

shared library A code fragment exporting a set of routines that can be called by multiple programs. Because they are prepared for use dynamically—that is, at program execution time instead of at program generation time—shared libraries are also called dynamically linked libraries.

software interrupt An interrupt generated by software and used to interrupt a task. A software interrupt can be originated by any task. To receive software interrupts, you write a software interrupt handler routine and call a microkernel function to specify the task with which it is to be associated; the microkernel returns a software interrupt ID. You can pass that ID to any task in any address space, and that task can then send a software interrupt to your task at any time.

supervisor mode A state of operation of the PowerPC processor, enforced by the internal logic of the processor, that allows access to certain critical resources, such as all processor instructions and tables that control memory protection. See also **privileged code, user mode.**

System Notification Service A microkernel service that provides a mechanism for Mac OS 8 and other service providers, such as device drivers, to broadcast notification of specific events to tasks. Any task can register to receive notification for specific types of events. See also **consumer**, **notification provider**.

task The basic unit of program execution in Mac OS 8. Preemptively scheduled and assigned a priority by the microkernel, every task has its own stack and set of registers. The microkernel uses processes to track the resources required by tasks, so that every process is associated with at least one task, and several tasks can be associated with a single process. See also **main task.**

thread (1) A path of execution through a program. For example, one thread in a program might handle user interactions, another might perform calculations, and a third might perform I/O. (2) To design software with more than one path of execution. Mac OS 8 developers can thread products using one or a combination of three different approaches. That is, developers can divide operations so that they are performed by more than one process, by more than one task in a single process, or by more than one cooperatively scheduled thread within a single task.

time slice An interval of time during which a task is given access to the CPU. Under certain circumstances, if there are two or more tasks eligible to run, the microkernel gives each task a specific amount of time (referred to as a time slice) to execute. When the time slice expires, the microkernel preempts the running task and passes control to another eligible task.

C H A P T E R 2

Tasks Reference

Contents

Tasks Constants and Data Types

Task Priority

The microkernel uses task priorities to schedule the execution of tasks. When multiple tasks are eligible to execute, the microkernel gives preference to those with higher priorities. Once a task with a real time priority (page 2-8) begins executing, the microkernel allows that task to continue executing until it blocks or until a task with a higher real time priority becomes eligible to execute. Once a task with a lower than real time priority begins executing, the microkernel may preempt it after it has executed for a specific amount of time or when a higher priority task becomes eligible to execute.

When you create a task using the `CreateTask` function (page 2-18), you specify its priority using either a numeric value or a category that represents the intended purpose of the task (the microkernel determines the appropriate numeric value corresponding to each category). In general, you should specify the priority of a task using a category because it ensures that your task will be scheduled appropriately relative to other tasks in various categories. However, if you need to specify a numeric priority for a task, you must specify a value between 1 and 30 (the microkernel reserves priorities 0 and 31 for its own use). Once you create a task, you can use the `SetTaskPriority` function (page 2-31) anytime you want to increase or decrease its priority.

When you use the `CreateTask` or `SetTaskPriority` functions, you should always specify the lowest task priority that is practical for your purposes. Smaller values indicate lower priorities. The categories defined in the task priority data type are listed, from top to bottom, in order of increasing priority.

Although you can choose the priority of your tasks, you should not rely on the relative priorities of tasks to provide implicit synchronization. For example, when a higher priority task page faults, it may enable a lower priority task to execute.

You use task priority constants to set the desired bits in the `options` parameters you pass to the `CreateTask` or `SetTaskPriority` functions. Also, you can use any of the constants to interpret values supplied on output in the task information structure (page 2-14) parameter of the `GetTaskInformation` function (page 2-26).

The constants for these task priorities are defined in the `TaskPriority` data type. The constant `kTaskPriorityMask` is used to mask a priority value. The constants `kTaskPriorityIsAbsolute` and `kTaskPriorityIsSymbolic` are used in combination with a priority value. The remaining constants represent categories and are listed in order from the lowest priority (`kTaskBackgroundPriority`) to the highest priority (`kTaskRealTimePriority16`).

```
typedef OptionBits TaskPriority;                /* task priorities */
enum {
    kTaskPriorityMask           = 0x0000001F,   /* priority value */

    kTaskPriorityIsAbsolute     = 0x00000100,   /* is numeric */
    kTaskPriorityIsSymbolic     = 0x00002000,   /* is a category */

    kTaskBackgroundPriority     = 0x00002001,   /* lowest category */
    kTaskAppCPUBoundPriority    = 0x00002002,   /* CPU-bound*/
    kTaskAppNonUIPriority       = 0x00002003,   /* secondary to UI */
    kTaskAppPriority            = 0x00002004,   /* user inteface * /
    kTaskUIHelperPriority       = 0x00002005,   /* toolbox helpers */
    kTaskLowServerPriority      = 0x00002006,   /* lowest servers */
    kTaskServerPriority         = 0x00002007,   /* servers */
    kTaskHighServerPriority     = 0x00002008,   /* highest servers */
    kTaskLowDriverPriority      = 0x00002009,   /* lowest drivers*/
    kTaskDriverPriority         = 0x0000200A,   /* drivers */
    kTaskHighDriverPriority     = 0x0000200B,   /* highest drivers */
    kTaskRealTimePriority1      = 0x0000200C,   /* lowest real time */
    kTaskRealTimePriority2      = 0x0000200D,   /* real time */
    kTaskRealTimePriority3      = 0x0000200E,   /* real time */
    kTaskRealTimePriority4      = 0x0000200F,   /* real time */
    kTaskRealTimePriority5      = 0x00002010,   /* real time */
    kTaskRealTimePriority6      = 0x00002011,   /* real time */
    kTaskRealTimePriority7      = 0x00002012,   /* real time */
    kTaskRealTimePriority8      = 0x00002013,   /* real time */
    kTaskRealTimePriority9      = 0x00002014,   /* real time */
    kTaskRealTimePriority10     = 0x00002015,   /* real time */
    kTaskRealTimePriority11     = 0x00002016,   /* real time */
    kTaskRealTimePriority12     = 0x00002017,   /* real time */
    kTaskRealTimePriority13     = 0x00002018,   /* real time */
    kTaskRealTimePriority14     = 0x00002019,   /* real time */
    kTaskRealTimePriority15     = 0x0000201A,   /* real time */
    kTaskRealTimePriority16     = 0x0000201B    /* highest category */
};
```

Field descriptions

`kTaskPriorityMask` A bit mask for determining the value of a numeric priority or priority category. When you wish to determine the current numeric priority of an existing task, call the `GetTaskInformation` function, inspect the `priority` field of the `taskInfo` parameter that is supplied on output, and mask the bits indicated by this constant. When you wish to determine the options used to create an existing task, call the `GetTaskInformation` function, inspect the `options` field of the `taskInfo` parameter that is supplied on output, and mask the bits indicated by this constant, the `kTaskPriorityIsAbsolute` constant, and the `kTaskPriorityIsSymbolic` constant. The resulting priority value should be interpreted as a numeric priority if the `kTaskPriorityIsAbsolute` bit of the `options` field is also set or it should be interpreted as a category if the `kTaskPriorityIsSymbolic` bit of the `options` field is also set.

`kTaskPriorityIsAbsolute`

The bit indicating whether the priority value should be treated as a numeric priority. If this bit is set, the priority of a task is specified in numeric terms. When you supply a numeric priority value in the `options` parameter of the `CreateTask` or `SetTaskPriority` functions, you must set this bit as well. In general, you should supply a priority category rather than a numeric priority.

When you call the `GetTaskInformation` function for an existing task and you wish to determine if the task was created using a numeric priority, inspect the value supplied on output in the `options` field of the `taskInfo` parameter and examine the bit indicated by this constant. If this bit is set, a numeric priority (the value of which can be obtained by using the `kTaskPriorityMask` constant) was supplied for this task.

`kTaskPriorityIsSymbolic`

The bit indicating whether the priority value should be treated as a priority category. If this bit is set, the priority of a task is specified in terms of the intended purpose of the task. This bit is set implicitly when you supply a category value in the `options` parameter of the `CreateTask` or `SetTaskPriority` functions.

When you call the `GetTaskInformation` function for an existing task and you wish to determine if the task was created using a priority category, inspect the value supplied on output in the `options` field of the `taskInfo` parameter and examine the bit indicated by this constant. If this bit is set, a priority category (the value of which can be obtained by using the `kTaskPriorityMask` constant) was supplied for this task.

`kTaskBackgroundPriority`
The value indicating the category for tasks that should execute only when the CPU is idle. This is the lowest priority category. For example, if your task's primary purpose is to compress files or perform full text indexing, set this value in the `options` parameter you pass to the `CreateTask` or `SetTaskPriority` functions.

`kTaskAppCPUBoundPriority`
The value indicating the category for CPU-bound tasks. If your task's primary purpose is to perform operations that do not block implicitly (such as arithmetic operations), set this value in the `options` parameter you pass to the `CreateTask` or `SetTaskPriority` functions.

`kTaskAppNonUIPriority`
The value indicating the category for tasks of a slightly lower priority than the task handling user interaction. For example, if your task's primary purpose is to repaginate changes to a document or recalculate changes to a spreadsheet, set this value in the `options` parameter you pass to the `CreateTask` or `SetTaskPriority` functions.

`kTaskAppPriority` The value indicating the category for tasks handling the user interaction aspects of an application. In general, this category applies to application tasks that are not intended for execution in the background. If your task's primary purpose is to handle the user interaction aspects of an application, set this value in the `options` parameter you pass to the `CreateTask` or `SetTaskPriority` functions.

`kTaskUIHelperPriority`
: The value indicating the category for I/O-bound tasks and tasks that the toolbox depends upon to implement the user interface of the system. If your task is I/O-bound or its primary purpose is to help implement the user interface of the system, set this value in the `options` parameter you pass to the `CreateTask` or `SetTaskPriority` functions.

`kTaskLowServerPriority`
: The value indicating the category for server tasks. The `kTaskLowServerPriority` constant represents a slightly lower priority than the `kTaskServerPriority` constant.

`kTaskServerPriority`
: The value indicating the category for server tasks. This category applies to most system services. If your task's primary purpose is to perform system services, you can set this value in the `options` parameter you pass to the `CreateTask` or `SetTaskPriority` functions.

`kTaskHighServerPriority`
: The value indicating the category for server tasks. The `kTaskHighServerPriority` constant represents a slightly higher priority than the `kTaskServerPriority` constant.

`kTaskLowDriverPriority`
: The value indicating the category for device drivers. The `kTaskLowDriverPriority` constant represents a slightly lower priority than the `kTaskDriverPriority` constant.

`kTaskDriverPriority`
: The value indicating the category for device drivers. If your task's primary purpose is to function as a device driver, set this value in the `options` parameter you pass to the `CreateTask` or `SetTaskPriority` functions.

  The priority represented by the device driver category is higher than that of any other category except the real time task categories. This is because a device driver needs to be highly responsive when an I/O request occurs.

`kTaskHighDriverPriority`
: The value indicating the category for device drivers. The `kTaskHighDriverPriority` constant represents a slightly higher priority than the `kTaskDriverPriority` constant.

`kTaskRealTimePriority1—kTaskRealTimePriority16`

The values indicating the categories for real time tasks. There are 16 levels of real time categories. Real time categories should be used for tasks that perform real time operations such as digital signal processing for telephony, audio, and video. If your task's primary purpose is to perform real time operations, set one of these values in the `options` parameter you pass to the `CreateTask` or `SetTaskPriority` functions.

A higher real time priority task that becomes eligible to execute always preempts the currently executing task if the currently executing task has a lower real time or other type of priority. Unless a higher real time priority task becomes eligible to execute, a task with a real time priority executes until it blocks.

Set Task Priority Options

You can change the priority of an existing task by calling the `SetTaskPriority` function (page 2-31) and specifying either a particular task priority (page 2-3) or a relative change to the priority. You can set the priority of a task no higher than 30 and no lower than 1 (the microkernel reserves priorities 0 and 31 for its own use). When you use the set task priority options to specify a relative change to an existing task, you use them in conjunction with the task priority constants to specify the amount of the relative change as a numeric value or as a priority category. In general, you should specify the change using a priority category (which is possible only when you use the `kTaskRaisePriorityToAtLeast` or `kTaskLowerPriorityToAtMost` constants). You use the set task priority options only when you are changing the priority of an existing task, not when you are creating a new task.

The constants for all set task priority options are defined in the `SetTaskPriorityOptions` data type.

```
typedef OptionBits SetTaskPriorityOptions;          /* priority change */
enum(
    kTaskRaisePriorityBy            = 0x00000200,   /* raise priority *(/
    kTaskLowerPriorityBy            = 0x00000400,   /* lower priority */
    kTaskRaisePriorityToAtLeast     = 0x00000800,   /* maximum */
    kTaskLowerPriorityToAtMost      = 0x00001000,   /* minimum */
);
```

Field descriptions

`kTaskRaisePriorityBy`

The bit indicating that the current priority of the task should be raised. When you call the `SetTaskPriority` function to increment the current priority of a task, set this bit in the `options` parameter, specify the numeric increase in the bits of the `options` parameter indicated by the `kTaskPriorityMask` constant, and set the bit in the `options` parameter indicated by the `kTaskPriorityIsAbsolute` constant. Even if the new value computed by adding the specified amount to the current priority exceeds 30, the microkernel will set the new priority no higher than 30. In general, you should use the `kTaskRaisePriorityToAtLeast` constant with a priority category instead of using the `kTaskRaisePriorityBy` constant.

The microkernel always saves the new computed value so that it can compute subsequent increases or decreases relative to this value. For example, if the current priority is 29 and you attempt to raise the priority by 2, the microkernel will save 31 but raise the actual priority to only 30. If you subsequently decrease the priority by 2, the microkernel will apply the decrease to the saved value for a result of 29 and set the new priority of the task to 29.

`kTaskLowerPriorityBy`

The bit indicating that the current priority of the task should be lowered. When you call the `SetTaskPriority` function to decrement the current priority of a task, set this bit in the `options` parameter, specify the numeric decrease in the bits of the `options` parameter indicated by the `kTaskPriorityMask` constant, and set the bit in the `options` parameter indicated by the `kTaskPriorityIsAbsolute` constant. Even if the new value computed by subtracting the specified amount from the current priority is less than 1, the microkernel will set the new priority no lower than 1. In general, you should use the `kTaskLowerPriorityToAtMost` constant with a priority category instead of using the `kTaskLowerPriorityBy` constant.

The microkernel always saves the new computed value so that it can compute subsequent increases or decreases relative to this value. For example, if the current priority is 2 and you attempt to decrease the priority by 2, the microkernel will save 0 but lower the actual priority to only 1. If you subsequently increase the priority by 2, the microkernel will apply the increase to the saved value for a result of 2 and set the new priority of the task to 2.

`kTaskRaisePriorityToAtLeast`

The bit indicating that the new priority should be increased to the greater of the specified priority and the current priority. The microkernel will set the new priority no higher than 30. If the current priority of the task is greater than the specified priority, the priority remains unchanged. You may specify a numeric priority, but in general, you should specify a priority category instead. When you call the `SetTaskPriority` function to conditionally increase the priority of a task, set this bit in the `options` parameter, specify the numeric value or category in the bits of the `options` parameter indicated by the `kTaskPriorityMask` constant, and, if you are specifying a numeric value, set the bit in the `options` parameter indicated by the `kTaskPriorityIsAbsolute` constant.

`kTaskLowerPriorityToAtMost`

The bit indicating that the new priority should be decreased to the lower of the specified priority and the current priority. The microkernel will set the new priority no lower than 1. If the current priority of the task is lower than the specified priority, the priority remains unchanged. You may specify a numeric priority, but in general, you should specify a priority category instead. When you call the `SetTaskPriority` function to conditionally decrease the priority of a task, set this bit in the `options` parameter, specify the numeric value or category in the bits of the `options` parameter indicated by the `kTaskPriorityMask` constant, and, if you are specifying a numeric value, set the bit in the `options` parameter indicated by the `kTaskPriorityIsAbsolute` constant.

Task Relationship

The tasks created within a single process (described in the chapter 'Process Manager Reference' to be provided at a later date) are arranged in one or more tree structures that define their relationships. The task that resides at the root of a task tree is an orphan task. The child tasks of that orphan task reside in the same task tree.

You pass a task relationship to the `SetTaskPriority` function (page 2-31) and the `TerminateTask` function (page 2-24) to specify whether those functions should act upon a single task or a family of tasks. The microkernel does not define the order in which these functions will act on the tasks within a family.

The constants for all task relationships are defined in the `TaskRelationship` data type.

```
typedef UInt32 TaskRelationship;    /* scope of operation*/
enum {
    kTaskOnly           = 0,        /* specified task, only */
    kTaskAndChildren    = 1,        /* task and all descendents *
    kTaskFamily         = 2,        /* entire task tree */
    kTaskKernelProcess  = 3         /* all tasks within process */
};
```

Field descriptions

`kTaskOnly`	The value indicating that the operation should apply only to the specified task. When you want the `SetTaskPriority` function or the `TerminateTask` function to operate only on the specified task, pass this value in the `scope` parameter of those functions.
`kTaskAndChildren`	The value indicating that the operation should apply to the specified task as well as its direct and indirect descendents. When you want the `SetTaskPriority` function or the `TerminateTask` function to operate on the specified task and its descendents, pass this value in the `scope` parameter of those functions.
`kTaskFamily`	The value indicating that the operation should apply to the entire family of the specified task. The microkernel identifies the root task of the specified task's tree, and then it applies the operation to the root of the task tree as well as the root's direct and indirect descendents. When you

want the `SetTaskPriority` function or the `TerminateTask` function to operate on the entire task tree that includes the specified task, pass this value in the `scope` parameter of those functions.

`kTaskKernelProcess`
The value indicating that the operation should apply to all tasks within the same process as the specified task. When you want the `SetTaskPriority` function or the `TerminateTask` function to operate on all tasks within the same process as the specified task, pass this value in the `scope` parameter of those functions.

Task Options

When you use the `CreateTask` function, you can specify various task options. One task option allows you to specify that the newly created task should be an orphan. Unless you specify this option, a task created within the process of its creator will be a child of its creator (a task created within a different process than that of its creator will be an orphan implicitly). The second task option lets you specify whether the microkernel should create a stack for the task that cannot be paged out of memory (but you should avoid specifying this option).

Creating an orphan task affects the conditions under which the orphan terminates. For example, if the `TerminateTask` function (page 2-24) is applied to a specified task and its descendents, the orphans of the specified task will not terminate unless the entire process terminates.

In general, you should not force the microkernel to create a stack that cannot be paged out of memory. However, it may be necessary for you to request this if, for example, you have to ensure that a task's stack will be accessible to hardware interrupt handlers and secondary interrupt handlers.

The constants for all task options are defined in the `TaskOptions` data type.

```
typedef OptionBits TaskOptions;             /* orphans and stacks */
enum (
    kTaskIsOrphan           = 0x00400000,   /* orphan */
    kTaskIsResident         = 0x00004000,   /* resident */
);
```

Field descriptions

`kTaskIsOrphan`	The value indicating that a task created within the process of its creator should be an orphan. When a task is creating another task within the same process and it wants the newly created task to be an orphan, set this value in the `options` parameter of the `CreateTask` function.
`kTaskIsResident`	The value indicating that the microkernel should create a stack for the task that cannot be paged out of memory. In general, you should not specify this option. When you must force the creation of a stack that cannot be paged out of memory, set this value in the `options` parameter. Setting this value does not affect the code, heap, or static data of the task. For related information, see the `ControlPagingForRange` function described in the chapter 'Virtual Memory Services Reference.'

Task Name

A task name is a 4-character value that you can use for debugging. You associate a task name with a task by specifying the name when you call the `CreateTask` function. The microkernel stores the name you specify, but it does not use the name for any purpose. You can obtain the task name for an existing task by calling the `GetTaskInformation` function (page 2-26).

The `TaskName` data type defines a task name.

```
typedef OSType TaskName;    /* 4-character task name */
```

Task ID

You use a task ID to refer to a task. You must use an ID to refer to a task because you cannot refer directly to the underlying data structure of a task.

When you create a task using the `CreateTask` function, it generates a task ID for the newly created task. An existing task can call the `CurrentTaskID` function (page 2-30) to get its own task ID. You must supply this ID when you wish to operate upon this task using the other functions described in this chapter.

The `TaskID` data type defines a task ID.

```
typedef struct OpaqueTaskID* TaskID;
```

The Task Information Structure

The task information structure contains information about a task. You obtain a task information structure by calling the `GetTaskInformation` function (page 2-26). The task information structure you obtain reflects the state of the task when you called the `GetTaskInformation` function. Because the microkernel uses a preemptive scheduling mechanism, the information supplied by the task information structure may be obsolete by the time the `GetTaskInformation` function returns.

The `TaskInformation` data type defines a task information structure.

```
struct TaskInformation {                /* task information structure */
    TaskName            name;                   /* task name*/
    KernelProcessID     owningKernelProcess;    /* task's process */
    TaskOptions         options;                /* creation options */
    TaskPriority        priority;               /* current priority */
    SchedulerState      taskState;              /* task state */
    SchedulerState      swiState;               /* software interrupts */
    Boolean             isTerminating;          /* is terminating */
    Boolean             reserved2[3];           /* reserved */
    ItemCount           softwareInterrupts;     /* quantity processed */
    LogicalAddress      stackLimit;             /* stack address */
    ByteCount           stackSize;              /* stack size */
    AbsoluteTime        creationTime;           /* task creation time */
    AbsoluteTime        cpuTime;                /* CPU time consumed */
    void                *reserved;              /* reserved */
};
typedef struct TaskInformation TaskInformation, *TaskInformationPtr;
```

Field descriptions

`name`	The 4-character task name (page 2-13) passed in the `name` parameter of the `CreateTask` function (page 2-18) when this task was created.
`owningKernelProcess`	The process ID (described in the chapter 'Process Manager Reference' to be provided at a later date) passed in the

	`owningKernelProcess` parameter of the `CreateTask` function when this task was created.
`options`	The combination of task priority (page 2-3) and task options (page 2-12) set as a result of calling the `CreateTask` function.
`priority`	The current numeric task priority. This field specifies a numeric priority even if a priority category was specified in the `options` parameter passed to the `CreateTask` function.
`taskState`	A 4-character abbreviation of the scheduler state of the task. This field is for debugging purposes. See the chapter 'Debugging Services Reference' for information on the scheduler state enumerators.
`swiState`	A 4-character abbreviation of the scheduler state of the software interrupts (described in the chapter 'Interrupt Services Reference') associated with the task. This field is for debugging purposes. See the chapter 'Debugging Services Reference' for information on the scheduler state enumerators.
`isTerminating`	A Boolean value indicating whether the task is in the process of terminating. This field is true if the task is terminating.
`softwareInterrupts`	The number of software interrupts sent to this task that have been processed. The value in this field does not include any software interrupts that are pending execution by the task.
`stackLimit`	The logical address of the base of the stack for this task. If the creator of the task supplied a stack for this task, then the value in this field is the address the creator specified in the `stackBase` parameter of the `CreateTask` function. If the creator of the task did not supply a stack, then the value in this field is the base of the stack the microkernel created for this task.
`stackSize`	The size of the stack for this task. If the creator of the task supplied a stack size for this task, then the value in this field is the number of bytes the creator specified in the `stackSize` parameter of the `CreateTask` function. If the creator of the task did not supply a stack and stack size,

	then the value in this field is the size of the stack the microkernel created for this task.
`creationTime`	The time at which the task was created. You can compute the time that has passed since the creation of the task in absolute time units by subtracting the value in this field from the value returned by the `UpTime` function (described in the chapter 'Timing Services Reference').
`cpuTime`	The amount of CPU time the task has consumed. The value in this field includes time consumed by microkernel execution, by processing software interrupts, and by processing hardware and secondary interrupts that occurred while the task was executing.

When you call the `GetTaskInformation` function, you must supply a version for the task information structure you pass in the `taskInfo` parameter. The task information version enables your software to remain compatible with future releases of the microkernel. The task information version is defined by the `kTaskInformationVersion` constant. It represents the only version of the task information structure available at this time.

```
enum {
    kTaskInformationVersion = 1
};
```

Task Main Entry Point

When you call the `CreateTask` function (page 2-18) to create a task, you pass a pointer to the main entry point at which the task will begin executing. If it does not terminate explicitly using the `ExitTask` function (page 2-22) or the `TerminateTask` function (page 2-24), the task terminates implicitly when the main entry point function returns.

The main entry point you specify must conform to the declaration of a task main entry point. The declaration of a task main entry point requires you to specify a main entry point function that takes a single parameter and returns a result code. The parameter allows you to pass any type of initial data to the main entry point when you create the task. The result code represents the completion status of the main entry point function upon its return. This completion status is returned to the creator of the task if the creator requested notification of the task's termination.

The TaskProc data type defines a pointer to a task main entry point.

```
typedef OSStatus (*TaskProc)(void *parameter);  /* main entry point */
```

Task Storage Index

Each task within a process has access to the local data on its stack and to static data that is shared globally among all tasks within the same process. Sharing static data can be useful in many cases. For example, a function could use a static lock to synchronize access to a resource. Because all tasks calling that function would attempt to acquire the same static lock, the function could guarantee that only one task would be modifying the resource at any time.

In some cases, however, you may wish to provide static data on a per-task rather than a shared basis. For example, a shared library might need to preserve unique state information for each task using the shared library. To support the per-task static data mechanism, the microkernel provides a task storage index, a task storage value (page 2-17), and several functions that operate upon these data types.

A task storage index is an identifier for a particular per-task static variable. Typically, a shared library calls the AllocateTaskStorageIndex function (page 2-33) once to allocate a unique per-task static variable for every task within a process. The AllocateTaskStorageIndex function returns a single task storage index that identifies a unique per-task static variable for every task within the process (as well as any tasks created subsequently within the process). Each task has its own per-task static variable and can assign a value to it by passing the task storage index and a task storage value (page 2-17) to the SetTaskStorageValue function (page 2-35). A task retrieves the value assigned to its per-task static variable by passing the task storage index to the GetTaskStorageValue function (page 2-37).

The TaskStorageIndex data type defines a task storage index.

```
typedef UInt32 TaskStorageIndex;    /* task storage index */
```

Task Storage Value

A task uses a task storage value to assign a unique value to its per-task static variable. Typically, a task storage value represents a reference to a block of data.

A task assigns a task storage value to its per-task static variable by calling the `SetTaskStorageValue` function (page 2-35) and passing the value along with the task storage index (page 2-17) that identifies the per-task static variable. A task gets the task storage value assigned to its per-task static variable by calling the `GetTaskStorageValue` (page 2-37) function.

The `TaskStorageValue` data type defines a task storage value.

```
typedef void *TaskStorageValue;     /* task storage value */
```

Tasks Functions

Creating and Terminating Tasks

In general, you can rely on high level services to create and terminate tasks for you. However, when you need to create and terminate tasks yourself, you can use the `CreateTask` function (page 2-18), the `ExitTask` function (page 2-22), and the `TerminateTask` function (page 2-24).

CreateTask

Creates a task.

```
OSStatus CreateTask (TaskName name,
                     KernelProcessID owningKernelProcess,
                     TaskProc entryPoint,
                     void *parameter,
                     LogicalAddress stackBase,
                     ByteCount stackSize,
                     const KernelNotification *notification,
                     TaskOptions options,
                     TaskID *theTask);
```

`name`	A 4-character task name (page 2-13). You supply a name to use for debugging purposes. You can obtain this task name by calling `GetTaskInformation` function (page 2-26) once the task is created.
`owningKernelProcess`	A process ID. You supply the process ID to which the newly created task should belong. You can obtain the process ID of the current task by calling the `CurrentKernelProcessID` function (described in the chapter 'Process Manager Reference' to be provided at a later date). If the current task supplies its own process ID in this parameter and the bit indicated by the `kTaskIsOrphan` constant is not set in the `options` parameter, the newly created task will be a child of the current task. If the current task specifies a different process ID than its own, the newly created task will be an orphan task within the specified process. Note that the microkernel will not permit a nonprivileged task to create a task within a process that contains privileged tasks. You can obtain this process ID by calling `GetTaskInformation` function (page 2-26) once the task is created. See the chapter 'Process Manager Reference' (to be provided at a later date) for more detailed information on processes.
`entryPoint`	A pointer to the main entry point at which you want the task to begin executing. The address of the main entry point must be within the address space of the process you supply in the `owningKernelProcess` parameter. You must specify a pointer to a function that conforms to the declaration of a task main entry point (page 2-16). You specify the initial data you want to pass to this main entry point in `parameter`. The return of the main entry point function will cause the task to terminate with the completion status returned by the main entry point function.
`parameter`	A 32-bit word of data you want the microkernel to pass to the function you specified in the `entryPoint` parameter when the task begins executing. You can supply any type of initial data in this parameter. You can supply `null` if there is no information you wish to convey to the main entry point at which the task begins executing.

`stackBase`	The logical address of the base of the stack you are supplying for this task. In general, you should specify `null` and allow the microkernel to create the stack for you (in which case it also creates guard pages for your stack). If you are supplying a stack, you must specify the size of the stack you are supplying in the `stackSize` parameter. You can obtain the actual stack base by calling `GetTaskInformation` function (page 2-26) once the task is created. See the chapters 'Dynamic Storage Allocation Services Reference' and 'Virtual Memory Services Reference' for more detailed information on memory.
`stackSize`	The size you want the stack to be for this task. If you are providing a stack in the `stackBase` parameter, you must supply the size of the stack you are providing in this parameter. If you are allowing the microkernel to create the stack, you can either supply a particular size you want the stack to be or specify 0 if you want the stack to be the default size (in either case, the microkernel allocates additional space for guard pages). You can obtain the actual stack size by calling `GetTaskInformation` function (page 2-26) once the task is created. A task that needs to perform its own stack checking can use the `RemainingStackSpace` function (page 2-30).
`notification`	A pointer to a microkernel notification structure (described in the chapter 'Microkernel Notification Reference' to be provided at a later date) you supply to specify the mechanism by which the microkernel should notify the creator of the task when the task terminates. A task can terminate explicitly by calling the `ExitTask` function (page 2-22) or implicitly when the main entry point function passed in the `entryPoint` parameter returns. Specify `null` if you do not want delivery of notification to the creator of the task.
`options`	The options that specify the attributes of the task. You supply a value in which you have set the desired combination of bits indicated by the task priority constants (page 2-3) and task options constants (page 2-12) or you supply `null` if you want the new task to inherit the options used to create its parent. You can obtain the options you specified when you created the task as well as its actual current priority by calling `GetTaskInformation` function (page 2-26) once the task is created.

`theTask`	A pointer to a task ID (page 2-13). On output, `CreateTask` supplies an ID for the newly created task. You use this ID to refer to this task in other task functions. Once a task has been created, it can call the `CurrentTaskID` function (page 2-30) to get its own task ID.
function result	A result code. The result code `noErr` indicates that `CreateTask` successfully created a task. The result code `kernelPrivilegeErr` indicates that a nonprivileged task attempted to create a task within a process containing privileged tasks. The `CreateTask` function returns result code `paramErr` if you attempt to set the priority of the task to 0 or 31. See “Tasks Result Codes” (page 2-38) for a description of other result codes that `CreateTask` may return.

DISCUSSION

In most cases, you can rely on high level services to create tasks for you. Launching an application automatically invokes the Process Manager (described in the chapter ‘Process Manager Reference’ to be provided at a later date), which creates the process and main task for the application. Launching a server automatically invokes the Server Manager (described in the chapter ‘Server Manager Reference’), which creates the process and main task for a server.

The tasks within a process are all privileged or they are all nonprivileged. Whether a task is privileged or nonprivileged depends upon the process in which it is created. A task is privileged when it is created within a process containing privileged tasks and it is nonprivileged when it is created within a process containing nonprivileged tasks. Although a task can create another task within a different process, the microkernel does not permit a nonprivileged task to create a task within a process containing privileged tasks. All applications, as well as most other types of software, should use nonprivileged tasks.

When you create a task, you should specify its priority using a category and you should always specify the lowest priority that is practical for your purposes.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

In most cases, you can rely on the Process Manager and Server Manager to terminate tasks and processes for you. However, if you need to terminate a task directly, see the descriptions for the `TerminateTask` function (page 2-24) and the `ExitTask` function (page 2-22), both of which force a task to terminate before its main entry point function returns.

ExitTask

Terminates the calling task.

```
void ExitTask       ( TerminateOptions options,
                      OSStatus exitStatus);
```

`options`	The termination options. You must specify `kNilOptions` for this parameter.
`exitStatus`	The completion status you are supplying for your termination. If the creator of the task requested notification (described in the chapter 'Microkernel Notification Reference' to be provided at a later date) of your termination and it specified a notification mechanism that would convey the completion status, the microkernel will include the completion status you supply when it notifies the creator.

DISCUSSION

Although you can rely on the Process Manager and Server Manager to terminate most tasks and processes for you, a task can use `ExitTask`, which terminates a task before its main entry point function returns, to terminate itself directly.

A task terminating itself using `ExitTask` will discontinue execution as soon as termination begins. If a terminated task has children, the microkernel delays reclaiming its stack until its children have terminated. This allows any child task that references its parent's stack to continue normal execution.

Once a task terminates, the microkernel delivers termination notification to the task's creator if the creator requested termination notification when it created the task. If no more tasks exist within the process that contained the calling task or the calling task is the main task within the process, the microkernel calls the `ExitKernelProcess` function (described in the chapter 'Process Manager Reference' to be provided at a later date) to terminate the task's process, which allows the Code Fragment Manager (described in the chapter 'Code Fragment Manager Reference' to be provided at a later date) to execute any termination routines for shared libraries the process was using.

Although the `TerminateTask` function (page 2-24) is similar to `ExitTask`, its effect on processes is different and it does not allow the Code Fragment Manager to execute termination routines. Using `ExitTask` is usually preferable to using the `TerminateTask` function.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

TerminateTask

Terminates the specified task.

```
OSStatus TerminateTask (
                TaskID theTask,
                TaskRelationship scope,
                TerminateOptions options,
                OSStatus status);
```

`theTask`	The task ID (page 2-13) for the task you want to terminate. You supply the task ID that the `CreateTask` function (page 2-18) generated. A task can call the `CurrentTaskID` function (page 2-30) to get its own task ID
`scope`	The task relationship constant (page 2-11) you supply to specify which other related tasks (if any) this operation should affect.
`options`	The termination options. You must specify `kNilOptions` for this parameter.
`status`	The completion status you are supplying for this termination. If the creator of the task (or creators, if you specified additional tasks to terminate in the `scope` parameter) requested notification (described in the chapter 'Microkernel Notification Reference' to be provided at a later date) of this task's termination and it specified a notification mechanism that would convey the completion status, the microkernel will include the completion status you supply when it notifies the creator.
function result	A result code. The result code `noErr` indicates that `TerminateTask` successfully terminated the specified tasks. The `TerminateTask` function returns no result code if it terminated the calling task. See "Tasks Result Codes" (page 2-38) for a description of other result codes that `TerminateTask` may return.

DISCUSSION

Although you can rely on the Process Manager and Server Manager to terminate most tasks and processes for you, you can use `TerminateTask`, which terminates a task before its main entry point function returns, to terminate any specified task directly.

A task you terminate using `TerminateTask` will discontinue execution as soon as termination begins. If you terminate a blocked task, the microkernel will force it to unblock (this implies that you must be prepared to handle the unexpected termination of tasks).

If a terminated task has children and you did not specify that they should terminate as well, the microkernel delays reclaiming its stack until its children have terminated. This allows any child task that references its parent's stack to continue normal execution.

Once a task terminates, the microkernel delivers termination notification to the task's creator if the creator requested termination notification when it created the task. If no more tasks exist within the process that contained the terminated task, the microkernel calls the `DeleteKernelProcess` function (described in the chapter 'Process Manager Reference' to be provided at a later date) to delete the process. Because it calls the `DeleteKernelProcess` function, the Code Fragment Manager (described in the chapter 'Code Fragment Manager Reference' to be provided at a later date) is not allowed to execute termination routines.

The `ExitTask` function (page 2-22), which a task uses to terminate itself, is similar to `TerminateTask`, but it has a different effect on processes and it allows the Code Fragment Manager to execute termination routines when a process terminates. When the `ExitTask` function results in the termination of a process, the `ExitKernelProcess` function (described in the chapter 'Process Manager Reference' to be provided at a later date) is called. Furthermore, the `ExitTask` function causes the task's process to terminate if the calling task is the main task within a process. You should use `TerminateTask` if you do not care about the execution of termination routines and you want to terminate the main task within a process without causing the process to terminate as well.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Getting Information About Tasks

You can obtain information about any existing task by calling the `GetTaskInformation` function (page 2-26). You can obtain the task IDs of all tasks within a process by calling the `GetTasksInKernelProcess` function (page 2-28).

A task can obtain its own task ID by calling the `CurrentTaskID` function (page 2-30) and it can determine how much space remains on its stack by calling the `RemainingStackSpace` function (page 2-30).

GetTaskInformation

Obtains information about the specified task.

```
OSStatus GetTaskInformation (
                     TaskID theTask,
                     PBVersion version,
                     TaskInformation *taskInfo);
```

`theTask`	A task ID (page 2-13). You supply the task ID that the `CreateTask` function (page 2-18) generated. A task can call the `CurrentTaskID` function (page 2-30) to get its own task ID
`version`	The version (described in the chapter ‘Systemwide Constants and Data Types Reference’ to be provided at a later date) of the task information structure (page 2-14) you are supplying. This

parameter enables your software to remain compatible with future releases of the microkernel. You must specify `kTaskInformationVersion` (page 2-16) for this parameter.

`taskInfo` A pointer to a task information structure (page 2-14). On output, `GetTaskInformation` supplies this structure with information about the task that you specify in the `theTask` parameter. When finished with this structure, you are responsible for releasing its memory.

function result

A result code. The result code `noErr` indicates that `GetTaskInformation` successfully supplied information for the specified task. See "Tasks Result Codes" (page 2-38) for a description of other result codes that `GetTaskInformation` may return.

DISCUSSION

The task information you obtain reflects the state of the task at the time you called `GetTaskInformation`. Because the microkernel uses a preemptive scheduling mechanism, the information supplied by the task information structure may be obsolete by the time the `GetTaskInformation` function returns.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

GetTasksInKernelProcess

Provides the task IDs of all tasks within the specified process.

```
OSStatus GetTasksInKernelProcess (
                        KernelProcessID kernelProcess,
                        ItemCount requestedTasks,
                        ItemCount *totalTasks,
                        TaskID *theTasks);
```

`kernelProcess`
: A process ID that you supply. You can obtain the process ID of the current task by calling the `CurrentKernelProcessID` function (described in the chapter 'Process Manager Reference' to be provided at a later date). You can obtain the process ID of any existing task by calling the `GetTaskInformation` function (page 2-26).

`requestedTasks`
: The maximum number of IDs that you are prepared to receive from `GetTasksInKernelProcess`. You supply in this parameter the number of entries available at the location specified by the `theTasks` parameter.

`totalTasks`
: A pointer to the total number of tasks within the specified process. On output, `GetTasksInKernelProcess` supplies this value. If this value is less than or equal to the value you specified in the `requestedTasks` parameter, then task IDs for all tasks in the specified process were supplied on output at the location specified by the `theTasks` parameter. If this value is greater than the value you specified in the `requestedTasks` parameter, then you did not provide enough entries to accommodate all task IDs. If you did not provide enough entries, you should allocate the number of entries specified by the `totalTasks` parameter and call `GetTasksInKernelProcess` again.

`theTasks`
: A pointer to a memory area into which `GetTasksInKernelProcess` should copy the task IDs of the tasks in the specified process. You are responsible for allocating enough memory to hold the number of task IDs you requested in the `requestedTasks` parameter. Specify `null` in this parameter

and specify 0 in the `requestedTasks` parameter if you want `GetTasksInKernelProcess` to supply a value on output for the `totalTasks` parameter, but you do not wish to obtain the task IDs. On output, `GetTasksInKernelProcess` copies the requested number of task IDs into the specified memory area until it runs out of entries or copies all the task IDs.

function result

A result code. The result code `noErr` indicates that the specified process exists, but it does not indicate whether all task IDs were supplied on output. See "Tasks Result Codes" (page 2-38) for a description of other result codes that `GetTasksInKernelProcess` may return.

DISCUSSION

The information you obtain from this function reflects which tasks existed within the specified process at the time you called `GetTasksInKernelProcess`. Because tasks can be created or terminated at any time, this information may be obsolete by the time `GetTasksInKernelProcess` returns.

In general, you should call `GetTasksInKernelProcess` in a loop until you have provided enough entries to accommodate all the task IDs. Because new tasks may be created within the specified process while you are calling `GetTasksInKernelProcess`, you may have to call it and allocate additional entries multiple times.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

CurrentTaskID

Returns the ID of the calling task.

```
TaskID CurrentTaskID(void );
```

function result
The task ID (page 2-13) of the calling task.

DISCUSSION

The `CurrentTaskID` function returns the task ID that the `CreateTask` function (page 2-18) generated for the current task.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

RemainingStackSpace

Returns the amount of space remaining in the current stack for the calling task.

```
ByteCount RemainingStackSpace  (void);
```

function result
The amount of space in bytes remaining in the current stack for the calling task.

DISCUSSION

A stack overflow causes an exception if the microkernel created guard pages for the stack. The microkernel creates the stack (and its guard pages) for a task if you pass `null` in the `stackBase` parameter of the `CreateTask` function (page 2-18).

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

This function has no calling restrictions.

Changing the Priority of a Task

You can change the priority of any existing task or set of related tasks within a process by calling the `SetTaskPriority` function (page 2-31).

SetTaskPriority

Sets the priority of a task.

```
OSStatus SetTaskPriority (
                    TaskID theTask,
                    TaskRelationship scope,
                    SetTaskPriorityOptions options);
```

`theTask`	The task ID (page 2-13) for the task whose priority you are changing. You supply the task ID that the `CreateTask` function (page 2-18) generated. A task can call the `CurrentTaskID` function (page 2-30) to get its own task ID

`scope`	The task relationship constant (page 2-11) you supply to specify which other related tasks (if any) this operation should affect.
`options`	The options that specify the new priority of the task or tasks. You supply a value in which you have set the desired combination of bits indicated by the task priority constants (page 2-3) and set task priority constants (page 2-8).
function result	A result code. The result code `noErr` indicates that `SetTaskPriority` successfully set the specified priority in the specified task or tasks. The `SetTaskPriority` returns result code paramErr if you attempt to set the priority of the task to 0 or 31. See "Tasks Result Codes" (page 2-38) for a description of other result codes that `TerminateTask` may return.

DISCUSSION

When you set the priority of a task, you should use a priority category and you should always specify the lowest priority that is practical for your purposes.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Working With Per-Task Static Data

In general, static data is shared globally among all tasks within a process. However, it is useful in some cases to provide separate static data for each task in a process. The functions described in this section support the microkernel's per-task static data mechanism.

AllocateTaskStorageIndex

Allocates a task storage index that identifies a different per-task static variable for every task within a process.

```
OSStatus AllocateTaskStorageIndex (TaskStorageIndex *theIndex);
```

`theIndex`	A pointer to a task storage index (page 2-17). On output, `AllocateTaskStorageIndex` supplies a newly created index that identifies a separate static variable for each task that uses it. Typically, you assign this index to a global static variable that all tasks using the shared library can access. To assign a value to its per-task static variable, a task passes this index and a task storage value (page 2-17) to the `SetTaskStorageValue` function (page 2-35). To read the value of its per-task static variable, a task passes this index to the `GetTaskStorageValue` function (page 2-37).
function result	A result code. The result code `noErr` indicates that `AllocateTaskStorageIndex` successfully created a task storage index. See “Tasks Result Codes” (page 2-38) for a description of other result codes that `AllocateTaskStorageIndex` may return.

DISCUSSION

Calling `AllocateTaskStorageIndex` generates a task storage index and allocates a unique per-task static variable for every task within a process. A unique per-task static variable is allocated as well for each task created subsequently within this process. This single task storage index identifies a different per-task static variable for every task.

During its initialization, a library should call `AllocateTaskStorageIndex` and store the newly created index in a global static variable. For each task in the process, an initial value of `null` is assigned to the per-task static variable identified by this index.

Because the microkernel supplies only a small quantity of task storage indices, each library should allocate no more than one. Typically, each task uses that one per-task static variable to refer to a larger block of data allocated in a memory area.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

See the `DeallocateTaskStorageIndex` function (page 2-34) for information on destroying a task storage index.

DeallocateTaskStorageIndex

Deallocates the specified task storage index.

```
OSStatus DeallocateTaskStorageIndex(TaskStorageIndex theIndex);
```

`theIndex`	A task storage index (page 2-17). You supply the task storage index that the `AllocateTaskStorageIndex` function (page 2-33) generated. Once deallocated, a subsequent call to the `AllocateTaskStorageIndex` function may reassign this task storage index to a new set of per-task static variables.
function result	A result code. The result code `noErr` indicates that `DeallocateTaskStorageIndex` successfully deleted the specified task storage index. See "Tasks Result Codes" (page 2-38) for a description of other result codes that `DeallocateTaskStorageIndex` may return.

DISCUSSION

A library should call `DeallocateTaskStorageIndex` when it will not result in synchronization problems.**(Russell, care to elaborate?)**.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

See the `SetTaskStorageValue` function (page 2-35) and the `GetTaskStorageValue` function (page 2-37) for information on setting and getting the value of the per-task static variable for a particular task.

SetTaskStorageValue

Assigns the specified task storage value to a per-task static variable for the calling task.

```
OSStatus SetTaskStorageValue (
                    TaskStorageIndex theIndex,
                    TaskStorageValue newValue);
```

`theIndex`	A task storage index (page 2-17). The calling task supplies the task storage index that the `AllocateTaskStorageIndex` function (page 2-33) generated. Typically, the library that allocated the task storage index assigned it to a global static variable. The task storage index identifies a separate static variable for each calling task.

`newValue`	A task storage value (page 2-17). The calling task supplies the value it wants to store in the per-task static variable identified by the `theIndex` parameter. The calling task can read the value by calling the `GetTaskStorageValue` function (page 2-37).
function result	A result code. The result code `noErr` indicates that `SetTaskStorageValue` successfully set the specified task storage value in the per-task static variable. See "Tasks Result Codes" (page 2-38) for a description of other result codes that `SetTaskStorageValue` may return.

DISCUSSION

Typically, a task uses its per-task static variable to store a reference to a larger block of data allocated in a memory area.

The `SetTaskStorageValue` function is less expensive to use than a microkernel call.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

See the `DeallocateTaskStorageIndex` function (page 2-34) for information on destroying a task storage index.

GetTaskStorageValue

Reads the task storage value assigned to a per-task static variable for the calling task.

```
TaskStorageValue GetTaskStorageValue (TaskStorageIndex theIndex);
```

`theIndex` A task storage index (page 2-17). The calling task supplies the task storage index that the `AllocateTaskStorageIndex` function (page 2-33) generated. Typically, the library that allocated the task storage index assigned it to a global static variable. The task storage index identifies a separate static variable for each calling task.

function result
A task storage value (page 2-17). This function returns the value stored in the per-task static variable identified by `theIndex` for the calling task. This value will be `null` or it will be the value assigned when the task most recently called the `SetTaskStorageValue` function with the same task storage index. A `null` result indicates that the calling task has not yet called `SetTaskStorageValue` to initialize its per-task static variable.

DISCUSSION

Before calling the `SetTaskStorageValue` function, a task should call `GetTaskStorageValue` and test the result. If the result is `null`, the task has not yet initialized its per-task static variable. To initialize its per-task static variable, the task should allocate a block of data and pass the address of the block to the `SetTaskStorageValue` function.

The `GetTaskStorageValue` function is less expensive to use than a microkernel call.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

See the `DeallocateTaskStorageIndex` function (page 2-34) for information on destroying a task storage index.

Tasks Result Codes

`noErr`	0	No error
`kernelPrivilegeErr`	-2404	Illegal attempt to create privileged task
`memFullErr`	–108	Not enough memory
`kernelIDErr`	–2419	Process ID or task ID does not exist

Dynamic Storage Allocation Service Reference

Contents

Dynamic Storage Allocation (DSA) service provides functions that you can use to allocate fixed or variable sized memory blocks that are referenced by pointers and variable-sized memory blocks that are referenced by handles. DSA service guarantees at least 4-byte alignment for the allocated blocks.

Each function that you use to allocate memory requires that you specify a memory allocator. Your choice of allocator determines where the memory is allocated (global area or calling process' area) and whether it is resident.

Constants and Data Types

This section describes your choice of memory allocators, which you use to specify whether the memory you allocate is resident, global, or local.

Memory Allocators

When you call one of the DSA functions to allocate a block of memory and initialize a pointer or handle that references this block, you must specify one of the following memory allocators: `xResidentAllocator`, `xDefaultGlobalAllocator`, or `xDefaultAllocator`. Each of these allocators is defined to allocate memory that is characterized by different attributes:

`xResidentAllocator`	Allocate memory that is resident. Use this type of allocator if you need storage for data that must not be paged out of memory. This type can be used only by privileged code like a driver or by code that can be called at interrupt time.
`xDefaultGlobalAllocator`	Allocate memory from the system-wide global area. Use this type of allocator if you need to store data that is visible to all processes.
`xDefaultAllocator`	Allocate memory from your process' area. Use this type of allocator if you need to store data that is visible only to your process.

Functions

You use the functions described in the following sections

- to allocate a block of memory and to initialize a handle or pointer that references these blocks
- to resize memory blocks that are referenced by variable pointers or by handles
- to dispose of the handles or pointers you have created

Allocating Fixed-Size Pointers

DSA Service offers two sets of functions that you can use to create and manipulate pointers. You use the set of functions described in this section to create and dispose of fixed size pointers. **Fixed-size pointers** reference blocks of memory whose size you cannot change. If you used a fixed-size pointer to allocate memory, you must specify the size of this memory block when you dispose of the pointer referencing it. A **variable-size pointer** (page 3-8) references a block of memory whose size you can change. You do not have to specify the size of the referenced memory when you dispose of a variable-sized pointer. Unless you need to resize blocks of memory or the size of the block is not known at disposal time, it is recommended that you use fixed-size pointers.

MemNewFixed

Allocates a block of memory referenced by the specified pointer.

```
extern OSStatus MemNewFixed(MemAllocatorRef inAllocator,
                    ByteCount inSize,
                    void **outMemory);
```

`inAllocator`	The name of the allocator (page 3-3) to be used in creating the pointer.
`inSize`	The size of the block of memory referenced by the pointer.
`outMemory`	On input, a pointer to a block of memory whose size is specified by the `inSize` parameter. If the function fails, this is set to `NULL`.

function result If there is not enough memory to allocate the requested amount, the function returns the result `kFullMemError`.

DISCUSSION

This function allocates a fixed-size block of memory in the calling program's area or in the global area, depending on the memory allocator specified. You cannot change the size of a fixed-size pointer. Unless you need to resize blocks of memory or the size of the block is not known at disposal time, it is recommended that you use fixed-size pointers.

SPECIAL CONSIDERATIONS

When you dispose of the pointer, you must specify the size of memory which it references (specified by the `inSize` parameter).

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

Use the `MemDisposeFixed` function (page 3-7) to dispose of memory allocated with the `MemNewFixed` function.

Use the `MemNewVariable` function (page 3-9), to allocate a block of memory whose size you can change.

MemNewFixedClear

Allocates a block of memory referenced by the specified pointer and initializes this block to 0.

```
extern OSStatus MemNewFixedClear(MemAllocatorRef inAllocator,
                    ByteCount inSize,
                    void **outMemory);
```

`inAllocator`	The name of the allocator (page 3-3) to be used in creating the pointer.
`inSize`	The size of the block of memory referenced by the pointer.
`outMemory`	On output, a pointer to a block of memory whose size is specified by the `inSize` parameter. If the function fails, this is set to `NULL`.
function result	If there is not enough memory to allocate the requested amount of memory, the function returns the result `kFullMemError`.

DISCUSSION

This function allocates a fixed-size block of memory in the calling program's area or in the global area, depending on the memory allocator specified. You cannot change the size of a fixed-size pointer. Unless you need to resize blocks of memory or the size of the block is not known at disposal time, it is recommended that you use fixed-size pointers.

SPECIAL CONSIDERATIONS

You must specify the size of this memory block when you dispose of the pointer referencing it.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

Use the `MemDisposeFixed` function (page 3-7) to dispose of a block of memory created with the `MemNewFixedClear` function.

Use the `MemNewVariable` function (page 3-9), to allocate a block of memory whose size you can change.

MemDisposeFixed

Frees a fixed-size block of memory and sets the pointer referencing it to `NULL`.

```
extern OSStatus MemDisposeFixed(
                        MemAllocatorRef inAllocator,
                        ByteCount inSize,
                        void **ioMemory);
```

`inAllocator`	The name of the allocator (page 3-3) that was used to create the pointer.
`inSize`	The size of the block of memory to be disposed. This must be the same size as that specified with the `inSize` parameter to the `MemNewFixed` function or `MemNewFixedClear` function. If it is not, the behavior of the function is undefined.
`ioMemory`	On input, the pointer referencing the memory to be disposed. On output, `*ioMemory` is set to `NULL`.

function result If an error occurs, the block is still disposed and the pointer referencing it is set to `NULL`.

DISCUSSION

Use this function to dispose of a block of memory allocated with the `MemNewFixed` function or `MemNewFixedClear` function.

If the calling program terminates without explicitly disposing of the pointers it has allocated, the following happens:

- If the pointer is global, it is not disposed.
- If the pointer is local, it is disposed as part of process termination.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To allocate a new block of memory that is fixed in size, use the `MemNewFixed` function (page 3-4) or the `MemNewFixedClear` function (page 3-6).

Allocating Variable-Sized Pointers

You use the functions described in this section to allocate, manipulate, and dispose of variable sized pointers. These are pointers that reference a block of memory whose size you can change during program execution.

Unless you need to resize blocks of memory or the size of the block is not known at disposal time, it is recommended that you use fixed-size pointers, which are described in "Allocating Fixed-Size Pointers" (page 3-4).

MemNewVariable

Allocates a block of memory that you can resize and returns a pointer that references it.

```
extern OSStatus MemNewVariable(MemAllocatorRef inAllocator,
                        ByteCount inSize,
                        void **outMemory);
```

`inAllocator`	The name of the allocator (page 3-3) to be used in creating the pointer.
`inSize`	The initial size of the memory block.
`outMemory`	The pointer referencing the block of memory whose size is specified by the `inSize` parameter. If the function fails, this pointer is set to `NULL`.
function result	If there is not enough memory to allocate the requested amount, the function returns the result `kFullMemError`.

DISCUSSION

When you dispose of a block of memory allocated with this function, you do not have to specify its size. You can change the size of the allocated memory with the `MemSizeVariable` function.

Unless you need to resize blocks of memory or the size of the block is not known at disposal time, it is recommended that you use fixed-size pointers

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

Use the `MemSizeVariable` function (page 3-11) to change the size of a block of memory.

Use the `MemDisposeVariable` function (page 3-13) to dispose of a block of memory allocated with the `MemNewVariable` function.

Use the `MemNewFixed` function (page 3-4) to allocate a fixed-size block of memory.

MemNewVariableClear

Allocates a block of memory that you can resize, initializes it to 0, and returns a pointer that references it.

```
extern OSStatus MemNewVariableClear(MemAllocatorRef inAllocator,
                     ByteCount inSize,
                     void **outMemory);
```

`inAllocator`	The name of the allocator (page 3-3) to be used in creating the pointer.
`inSize`	The initial size of the memory block.
`outMemory`	The pointer referencing the block of memory whose size is specified by the `inSize` parameter. If the function fails, this pointer is set to `NULL`.
function result	If there is not enough memory to allocate the requested amount, the function returns the result `kFullMemError`.

DISCUSSION

When you dispose of a block of memory allocated with this function, you do not have to specify its size. You can change the size of the allocated memory with the `MemSizeVariable` function.

Unless you need to resize blocks of memory or the size of the block is not known at disposal time, it is recommended that you use fixed-size pointers

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

Use the `MemSizeVariable` function (page 3-11) to change the size of a block of memory.

Use the `MemDisposeVariable` function (page 3-13) to dispose of a block of memory allocated with the `MemNewVariable` function.

Use the `MemNewFixed` function (page 3-4) to allocate a fixed-size block of memory.

MemSizeVariable

Resizes a variable-sized block of memory.

```
extern OSStatus MemSizeVariable(MemAllocatorRef inAllocator,
                     ByteCount inSize,
                     void **ioMemory);
```

`inAllocator`	The name of the allocator (page 3-3) used to create the pointer.
`inSize`	The new size of the block of memory.
`ioMemory`	On input, a pointer to the block of memory whose size you want to change. On output, a pointer to the resized block of memory. If the function fails, this pointer is left unchanged. The block returned might be different than the block passed in.
function result	If there is not enough memory to allocate a larger block of memory, the function returns the result `kMemFullErr`.

DISCUSSION

If you make the block of memory larger, the data that is added is undefined. If you make the block of memory smaller, the data that lies beyond the boundary specified by the `inSize` parameter is also undefined. Resizing the block might cause it to be moved.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

Use the `MemGetVariableSize` function (page 3-11) to obtain the current size of a block of memory.

Use the `MemNewFixed` function (page 3-4) to allocate a fixed-size block of memory.

MemGetVariableSize

Returns the current size of a block of memory allocated using the `MemNewVariable` function.

```
extern OSStatus MemGetVariableSize(MemAllocatorRef inAllocator,
                    void *inMemory,
                    ByteCount *outSize);
```

`inAllocator`	The name of the allocator (page 3-3) used to create the pointer.
`inMemory`	The pointer referencing the block of memory whose size you want to obtain.

`outSize`	On return, the size of the block of memory referenced by the `inMemory` parameter. If the function fails, this is set to 0.
function result	If the function succeeds, it returns the result `NoErr`.

DISCUSSION

If the block whose size is sought is not a valid block—that is, if it is not a variable block allocated using the allocator specified by the `inAllocator` parameter, the behavior of this function is undefined.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

Use the `MemSizeVariable` function (page 3-11) to change the size of a block allocated with the `MemNewVariable` function

Use the `MemDisposeVariable` function (page 3-13) to dispose of a block of memory allocated with the `MemNewVariable` function.

MemDisposeVariable

Disposes of a pointer that references a block of memory you have allocated using the `MemNewVariable` function.

```
extern OSStatus MemDisposeVariable(MemAllocatorRef inAllocator,
                      void **ioMemory);
```

`inAllocator`	The name of the allocator (page 3-3) used to create the pointer
`ioMemory`	A pointer to the block of memory you want to dispose of. On output, whether or not the function succeeds, the pointer is set to `NULL` and the block of memory is no longer considered valid.
function result	To be provided.

DISCUSSION

When the `MemDisposeVariable` function returns, the contents of the block are undefined. Note that memory referenced by the pointer you are disposing might be reallocated before the function returns.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

Use the `MemNewVariable` function (page 3-9) or `MemNewVariableClear` function (page 3-10) to allocate a block of memory that you can resize.

Allocating Handles

You can use the functions described in this section to create, resize, and dispose of handles. You can use the functions provided to manipulate handles whether these handles were created using the System 7 Memory Manager or whether they were created using the `MemNewHandle` function. Note, that the opposite is not true: you cannot use a handle created by the `MemNewHandle` function to call the System 7 Memory Manager.

Handles are provided for the convenience of developers who are moving existing code to applications that use the Dynamic Storage Allocation services. In general, handle use is not recommended in Mac OS 8.

MemNewHandle

Allocates a block of memory and returns a handle that references that block.

```
extern OSStatus MemNewHandle(MemAllocatorRef inAllocator,
                    ByteCount inSize,
                    Handle *outHandle);
```

`inAllocator`	The name of an allocator (page 3-3) to be used in creating the handle.
`inSize`	The size in bytes of the memory to be allocated.
`outHandle`	On return, the handle referencing the new block of memory.
function result	If there is not enough memory available to be allocated, the function returns the result `kMemFullErr`. If the function fails for any reason, the `outHandle` parameter is set to `NULL`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

Use the `MemSizeHandle` function (page 3-17) to grow or shrink the block of memory referenced by this handle.

Use the `MemDisposeHandle` function (page 3-19) to dispose of a handle.

Use the `MemNewHandleClear` function to allocate a block, initialize it to 0, and create a handle that references it.

MemNewHandleClear

Allocates a block of memory, initializes it to 0, and returns a handle that references that block.

```
extern OSStatus MemNewHandleClear(MemAllocatorRef inAllocator,
                          ByteCount inSize,
                          Handle *outHandle);
```

`inAllocator`	The name of an allocator (page 3-3) to be used in creating the handle.
`inSize`	The size in bytes of the memory to be allocated.
`outHandle`	On return, the handle referencing the new block of memory.
function result	If there is not enough memory available to be allocated, the function returns the result `kMemFullErr`. If the function fails for any reason, the `outHandle` parameter is set to `NULL`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

Use the `MemSizeHandle` function (page 3-17) to grow or shrink the block of memory referenced by this handle.

Use the `MemDisposeHandle` function (page 3-19) to dispose of a handle.

MemSizeHandle

Increases or decreases the size of a memory block referenced by the specified handle.

```
extern OSStatus MemSizeHandle (ByteCount inSize,
                               Handle inHandle);
```

`inSize`	The new size in bytes of the block of memory referenced by the handle specified by the `inHandle` parameter.
`inHandle`	The handle to the block of memory to be resized.
function result	The result `kMemFullErr` indicates that there is not enough memory to resize the handle.

DISCUSSION

You use the `MemSizeHandle` function to resize a block of memory you allocated using the `MemNewHandle` function or the `MemNewHandleClear` function.

If you make the block of memory larger, the data that is added is undefined. If you make the block of memory smaller, the data that lies beyond the boundary specified by the `inSize` parameter is also undefined.

It is possible that the block of memory referenced by `inHandle` is moved as a result of its being resized.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

Use the `MemGetHandleSize` function (page 3-18) to obtain the current size of a block of memory to which you have a handle.

Use the `MemDisposeHandle` function (page 3-19) to dispose of a handle.

MemGetHandleSize

Returns the size of a block of memory referenced by the specified handle.

```
extern OSStatus MemGetHandleSize (Handle inHandle,
                        ByteCount *outSize);
```

`inHandle`	The handle referencing the block of memory whose size you want to determine.
`outSize`	The size in bytes of the block of memory referenced by the handle specified by the `inHandle` parameter. If the function fails it sets the `outSize` parameter to 0
function result	To be provided.

DISCUSSION

If the value of the `inHandle` parameter is invalid, the behavior of the `MemGetHandleSize` function is undefined.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

Use the `MemSizeHandle` function (page 3-17) to resize the block of memory referenced by a handle.

Use the `MemDisposeHandle` function (page 3-19) to dispose of a handle.

MemDisposeHandle

Disposes of an existing handle.

```
extern OSStatus MemDisposeHandle (Handle *ioHandle);
```

`ioHandle`	On input, the name of the handle that you want to dispose of. On output, the address stored in `ioHandle` is set to `NULL`.
function result	To be provided.

DISCUSSION

If the handle specified by `ioHandle` is invalid, the behavior of `MemDisposeHandle` is undefined.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

Use the `MemNewHandle` function (page 3-15) or the `MemNewHandleClear` function (page 3-16) to create a new handle.

Glossary

Fixed-size pointer A pointer that references a block of memory whose size you cannot change. If you use a fixed-size pointer to allocate memory, you must specify the size of this memory block when you create or dispose of the pointer.

Variable-size pointer A pointer that references a block of memory whose size you can change. You do not have to specify the size of the referenced memory when you dispose of a variable-sized pointer.

Virtual Memory Services Reference

Contents

Mac OS 8 provides a virtual memory system that associates logical addresses in an address space to physical addresses in ROM or RAM and further extends this mapping to physical locations in secondary storage. Virtual memory is permanently enabled and its operations are transparent to most software.

However, because debuggers, client-server applications, and drivers often have specialized memory usage needs, an API to the virtual memory services is provided. This section describes these services, which enable you to

- create and destroy an address space and get information about address spaces
- create areas, delete areas, obtain information about areas, and change an area's access level
- map selected portions of a large backing store to logical memory
- obtain information about the use of select memory ranges (pages) and control the way in which select memory ranges are used
- share memory across addressing spaces
- work with processor caches
- prepare memory for I/O operations

Because the virtual memory services API is fairly large and serves diverse needs, this chapter organizes descriptions of data types and functions based on the entities you are interested in manipulating (address spaces, areas, pages) and on the operations you are interested in performing (sharing memory, working with processor caches, preparing for I/O).

For information about functions used for dynamic storage allocation, see the chapter "Dynamic Storage Allocation Reference."

Memory Management Constants and Data Types

This section describes the constants and data types used by virtual memory services. These include that data types used to specify the basic elements of addressing as well as the more complex data structures used to return information about address spaces, areas, pages, memory reservations, and I/O preparation tables.

Addresses

The basic memory element is the address space, identified by the following opaque ID.

```
typedef struct OpaqueAddressSpaceID *AddressSpaceID;
```

An **address space** is the domain of addresses that can be directly referenced by the processor at any given moment. You can identify the current address space by using the `CurrentAddressSpaceID` function (page 4-30).

A **logical address** specifies location within an address space. Logical addresses are unsigned; the lower bound of a logical address is zero; the upper bound is the size of the address space minus one.

```
typedef void *LogicalAddress;
```

A **physical address s**pecifies a location in physical memory: RAM, or ROM. You specify a physical address when preparing to transfer data with the `PrepareMemoryforIO` function (page 4-67).

```
typedef void *PhysicalAddress;
```

Backing Object Types

Pages can be swapped out to a backing store, identified by a backing object ID.

```
typedef struct OpaqueBackingObjectID *BackingObjectID;
```

You create a backing object ID for a backing store by making the appropriate call to the file system. Specifying a value of `kNoBackingObjectID` when creating an area means that the area is a scratch area.

```
#define kNoBackingObjectID ((BackingObjectID) kInvalidID)
```

Backing addresses are 64-bit integer values in anticipation of file systems that provide support for files larger then 4 GB.

```
typedef UInt64 BackingAddress; /* offset within a backing object */
```

Address Space Management

The address space information structure returns the number and type of pages in a given address space.

The Address Space Information Structure

When you use the `GetSpaceInformation` function (page 4-29) to obtain information about a specified address space, the function returns the information in the address space information structure.

The address space information structure is defined by the `SpaceInformation` data type.

```
struct SpaceInformation {
    ItemCount   numLogicalPages;
    ItemCount   numInMemoryPages;
    ItemCount   numResidentPages;
};
typedef struct SpaceInformation SpaceInformation, *SpaceInformationPtr;
```

Field descriptions

`numLogicalPages`	The total number of logical pages in this address space, excluding pages in global areas and pages in guard ranges. This is the same as the sum of the size of all areas in the address space.
`numInMemoryPages`	The number of logical pages in this address space that are currently in physical memory, excluding pages in global areas.
`numResidentPages`	The number of pages in this address space that are locked in physical memory. This number includes resident areas and pages locked with the `ControlPagingForRange` function (page 4-49) and the `PrepareMemoryForIO` function (page 4-67).

Area Management

You use area management data types to specify how an area is created and used: its memory accessibility, pageability, sparseness, location, and addressability.

Memory Access Level Enumeration

A memory area's accessibility depends on whether it's being accessed by privileged or user code and on the type of operation performed be this code.

To determine an area's memory access level, you call the `GetAreaInformation` function (page 4-41). This function stores the required information in the area information structure (page 4-9). You use the `SetAreaAccess` function (page 4-37) to change the access levels of existing areas.

You must specify the memory access level for user-mode or privileged code when you create an area with the `CreateArea` function (page 4-33) or the `CreateAreaForRange` function (page 4-37).

The memory access level enumeration specifies possible values for an area's memory access level.

```
typedef UInt32 MemoryAccessLevel;
enum {                                          /* memory access levels */
    kMemoryExcluded             = 0,    /* no access allowed */
    kMemoryReadOnly             = 1,    /* read and fetch allowed */
    kMemoryReadWrite            = 2,    /* read, write, fetch allowed*
    kMemoryCopyOnWrite          = 3,    /* not supported */
    kInheritUserAccess          = 4,    /* inherits user rights */
    kInheritPrivilegedAccess    = 5     /* inherits privileged rights */
};
```

Enumerator descriptions

`kMemoryExcluded`	No accesses, including instruction fetches, are allowed.
`kMemoryReadOnly`	Read and instruction fetch operations are allowed.
`kMemoryReadWrite`	Read, write and instruction fetch operations are allowed.
`kInheritUserAccess`	When using use the `CreateAreaForRange` function to create an area, specifying this value means that the new area

	inherits the user mode access rights of the area being mapped.
`kInheritPrivilegedAccess`	When using use the `CreateAreaForRange` function to create an area, specifying this value means that the new area inherits the privileged mode access rights of the area being mapped.

Area Usage Enumeration

A given area can be used in various ways. An area's usage is set at boot time or when an I/O device is configured and cannot be changed. To determine how an area is being used, you must examine the `usage` field of the area information structure, which is returned to you by the `GetAreaInformation` function.

The area usage enumeration defines the values that can be returned to you in the `usage` field of the area information structure.

```
typedef UInt32 AreaUsage;
enum {
    kUsageUnknown       = 0,
    kUsageRAM           = 1,
    kUsageROM           = 2,
    kUsageIO            = 3,
    kUsageVideoRAM      = 4
};
```

Enumerator descriptions

`kUsageUnknown`	The area's usage is not known.
`kUsageRAM`	The area is mapped to RAM.
`kUsageROM`	The area is mapped to ROM.
`kUsageIO`	The area is used for I/O.
`kUsageVideoRAM`	The area is used for video RAM.

Area Options Enumeration

When you create an area, you use the `options` parameter to the `CreateArea` function (page 4-33) to specify additional attributes for the area. When you use the `CreateAreaForRange` function (page 4-37) to create a new area based on the current area, the new area inherits all the options defined for the current area except for the `kGlobalArea` and `kPlacedArea` options.

To obtain the current setting of an area's attributes, you examine the area information structure returned by the `GetAreaInformation` function (page 4-41).

Some options apply only to scratch areas or nonpageable areas. Scratch areas are areas for which you specified `kNoBackingObjectID` for the `backingObject` parameter of the `CreateArea` function (page 4-33). A nonpageable area is an area for which you also specify the `kResidentArea` option.

The area options enumeration defines the values you can specify for the `options` parameter of the `CreateArea` function.

```
typedef OptionBits AreaOptions;
enum {
    kZeroFill                   = 0x00000001,
    kResidentArea               = 0x00000002,
    kSparseArea                 = 0x00000004,
    kPlacedArea                 = 0x00000008,
    kGlobalArea                 = 0x00000010,
    kPhysicallyContiguousArea   = 0x00000020
};
```

Option descriptions

`kZeroFill`	Memory in this area is initialized to zero. You can specify this option only for scratch areas and nonpageable areas. The initialization occurs when you first access the area.
`kResidentArea`	Data for this area is always physically resident. Specifying this option makes an area nonpageable: for such areas the microkernel never pages between physical memory and backing storage.
`kSparseArea`	Resources for this area are allocated on-demand. This option applies only to areas that are mapped to scratch space and to areas that are resident.
	If an area is mapped to scratch space, sparseness means that disk space is allocated to the scratch file as needed when you call the `ControlPagingForRange` function, rather than all at once when you call the `CreateArea` function.

For resident areas, sparseness means that the mapping between logical and physical memory is made as needed rather than when the area is created.

`kPlacedArea` This area begins at the location referenced by the `areaBase` parameter of the `CreateArea` function.

▲ **WARNING**
Be careful when you use the `kPlacedArea` option because the specified location might be part of an unknown memory reservation. To be safe, before creating the area, first create a reservation for the range into which the area is to be placed. See the `CreateAreaReservation` function (page 4-52) for more information. ▲

`kGlobalArea` Data for this area is addressable from any address space. Code in any address space can access this area in accordance with the user and privileged access levels specified for the area. The created area appears at the address referenced by the parameter `areaBase` in every address space. When you specify this option, the `addressSpace` parameter has no effect; pass the current address space ID instead.

`kPhysicallyContiguousArea`
The physical pages that make up the area is contiguous. You can only specify this option if you also specify the `kResidentArea` option.

You should use this option only for drivers of devices that must perform multipage DMA transfers but do not handle scatter-gather operations.

Specifying this option for the `CreateArea` function might cause the function to fail with this option, it may fail even when there is a great deal of available memory.

The Area Information Structure

When you use the `GetAreaInformation` function (page 4-41) to obtain information about a specified area, the function returns the information in the area information structure.

The area information structure is defined by the `AreaInformation` data type.

```
struct AreaInformation {
    AddressSpaceID          addressSpace;
    LogicalAddress          base;
    ByteCount               length;
    MemoryAccessLevel       userAccessLevel;
    MemoryAccessLevel       privilegedAccessLevel;
    AreaUsage               usage;
    BackingObjectID         backingObject;
    BackingAddress          backingBase;
    AreaOptions             options;
    KernelProcessID         owningKernelProcess;
};
typedef struct AreaInformation AreaInformation;
typedef AreaInformation *AreaInformationPtr;
```

Field descriptions

`addressSpace`	The address space that contains the area. This value is set to the constant `kGlobalAddressSpaceID` if the area is global to all address spaces.
`base`	The logical address of the area.
`length`	The size, in bytes, of the area.
`userAccessLevel`	The kinds of references allowed by user mode access. Reference kinds are specified by a memory access level enumeration value (page 4-6).
`privilegedAccessLevel`	The kinds of references allowed by privileged mode access. Reference kinds are specified by a memory access level enumeration value (page 4-6).
`usage`	The area’s use, such as RAM, I/O, or onboard video. Usage values are specified by an area usage enumeration value (page 4-7).
`backingObject`	The ID of the object providing backing store for the area. The value `kNoBackingObjectID` is returned if there is no backing object or if the area is a scratch area.
`backingBase`	The area’s base address within the backing object.
`options`	The options that were specified when the area was created. Option values are specified by the area options enumeration (page 4-7).

`owningKernelProcess`
The ID of the kernel process that was specified when the area was created. The area is automatically deleted when this kernel process exists or when the address space is deleted.

Page Management

Page management data types allow you to get information about pages in the page information structure and to control how a page is mapped by backing object providers, whether backing store or physical memory is allocated for it, and whether it is eligible for replacement.

Page State Information Enumeration

When you use the `GetPageInformation` function (page 4-48) to get information about a specified page, the function returns the information in the page information structure. The `information` field in this structure contains page state information bits for each logical page.

```
typedef UInt32 PageStateInformation;
enum {
    kPageIsProtected           = 0x00000001,
    kPageIsProtectedPrivileged = 0x00000002,
    kPageIsModified            = 0x00000004,
    kPageIsReferenced          = 0x00000008,
    kPageIsLockedResident      = 0x00000010,
    kPageIsInMemory            = 0x00000020,
    kPageIsShared              = 0x00000040,
    kPageIsWriteThroughCached  = 0x00000080,
    kPageIsCopyBackCached      = 0x00000100
};
```

Enumerator descriptions

`kPageIsProtected`
The page is protected against being written to by user mode software.

`kPageIsProtectedPrivileged`
The page is protected against being written to by privileged software.

`kPageIsModified`	Data has been written to the page since the last time it was mapped in or its data has been released by the `ReleaseData` function (page 4-46).
`kPageIsReferenced`	The page has been referenced (either load or store) since the last time the kernel's aging operation checked the page.
`kPageIsLockedResident`	The page is ineligible for replacement (that is, it is nonpageable) because it is part of a resident area or because there is at least one outstanding call of the `ControlPagingForRange` function (page 4-49) with the `kPagingModeResident` option set or the `PrepareMemoryForIO` function against it (page 4-67).
`kPageIsInMemory`	The logical page is mapped to physical memory.
`kPageIsShared`	The page's underlying physical page is mapped into additional logical pages.
`kPageIsWriteThroughCached`	Modifications to the page are written through the processor cache to main memory.
`kPageIsCopyBackCached`	Modifications to the page may be cached by the processor and not immediately reflected in main memory.

Page Control Operation Enumeration

You can control paging operations for a specified range of addresses by supplying one of these constants to the `operation` parameter of the `ControlPagingForRange` function. For example, programs that make sequential references to an area of memory could use these constants to decrease the time they spend waiting for paging operations and the amount of memory they use. Such a program would use the `kControlPageTouch` operation for addresses it is about to reference and the `kControlPageReplace` operation for addresses it has finished referencing.

```
typedef UInt32 PageControlOperation;
enum {
    kControlPageMakePageable    = 1,
    kControlPageMakeResident    = 2,
    kControlPageCommit          = 3,
    kControlPageTouch           = 4,
    kControlPageReplace         = 5,
    kControlPageFlush           = 6,
    kControlPageFlushAsync      = 7
};
```

Enumerator descriptions

`kControlPageMakePageable`
: The range is made eligible for page replacement. This only works for a range that was originally pageable, and which was made resident with a call to the `ControlPagingForRange` function with the `kControlPageMakeResident` enumerator. This enumerator cannot make pageable a range that was originally resident. Originally resident ranges do not benefit from this operation although the function still completes successfully.

  In effect, this enumerator “undoes” a previous call to the `ControlPagingForRange` function making a pageable range into a resident range. When all such calls have been undone, the page is returned to the mode associated with the area containing the range (that is, pages in backed areas become eligible for replacement, but pages in resident areas do not).

`kControlPageMakeResident`
: The range is to be loaded and made ineligible for page replacement. The calling task is blocked until the operation is complete.

`kControlPageCommit`
: The range is to have backing store allocated for its pages in a sparse pageable area, or physical memory is to be allocated for its pages in a sparse resident area. Ranges in areas created without the `kSparseArea` option specified do not benefit from this operation although the function still completes successfully. The calling task is blocked until this operation is complete.

`kControlPageTouch`
: The pages in the range are to be brought into physical memory and the calling task is not blocked while this is being done. Specifying this option does not cause the range to be mapped into logical memory, it just loads it into a cache. You can use this option to optimize

	performance by causing the system to read in pages that will be needed in the future.
`kControlPageReplace`	The physical memory space occupied by the pages in the range are to be made available for other uses, after writing the page data to backing store if necessary. You are not blocked while this is being done. The data in the page is always preserved by this operation. You can use this to optimize performance by giving the system information about pages that will not be needed in the near future.
`kControlPageFlush`	The pages in the range are forced to be written to backing store and you are blocked while this is being done.
`kControlPageFlushAsync`	The pages in the range are forced to be written to backing store and you are not blocked while this is being done.

The Page Information Structure

When you use the `GetPageInformation` function (page 4-48) to obtain information about one or more logical pages in a specified range of addresses, it returns information about each page in the page information structure. Its buffer has room for multiple entries, one for each page.

The page information structure is defined by the `PageInformation` data type.

```
struct PageInformation {
    AreaID                  area;
    ItemCount               count;
    PageStateInformation    information [kVariableLengthArray];
};
typedef struct PageInformation PageInformation;
typedef PageInformation *PageInformationPtr;
```

Field descriptions

`area`	The ID of the area associated with the range.
`count`	The number of page state information entries (page 4-11) returned.
`information`	An array that contains one page state information entry (page 4-11) for each logical page.

Memory Sharing

This section describes the constants and data types you use with the `InterspaceBlockCopy` function (page 4-61) and with the functions used to reserve memory. Memory reservation data types allow you to specify the boundaries and scope of a memory reservation, and to get information about established reservations in the reservation information structure.

Interspace Copy Options Enumeration

When you use the `InterspaceBlockCopy` function (page 4-61), you use the `options` parameter to specify the access checks to apply to the copy operation. The interspace copy options enumeration supplies the values you can use for the `options` parameter.

```
typedef OptionBits InterspaceCopyOptions;
enum {
 kCheckSourceUserRights         = 0x00000001,
 kCheckDestinationUserRights    = 0x00000002
};
```

Option descriptions

`kCheckSourceUserRights`
: The user (non-privileged) access rights to the source address are to be checked. If not specified, the caller's execution mode's access rights are checked.

`kCheckDestinationUserRights`
: The user (non-privileged) access rights to the destination address are to be checked. If not specified, the caller's execution mode's access rights are checked.

Reservation Options Enumeration

When you use the `CreateAreaReservation` function (page 4-52), you can specify some of the characteristics of the reservation being created by using one of the following enumeration values for the function's `options` parameter.

```
typedef OptionBits ReservationOptions;
enum {
    kPlacedReservation      = 0x00000001,
    kGlobalReservation      = 0x00000002,
    kGlobalAreaReservation  = 0x00000004
};
```

Option descriptions

`kPlacedReservation`
: The reservation is to be placed at a location specified by the `base` parameter. The microkernel aligns both the `base` and `length` values to page boundaries, which means that the actual reservation may be larger than you specified.

`kGlobalReservation`
: The reservation is to apply across all existing and future address spaces. The reservation appears at the address specified by the `base` parameter in every address space. Note that although the reservation is available globally, when you create an area within a reservation, the area is actually located within the address space that you specify when creating the area unless you also specify the `kGLobalAreaReservation` option

  Global reservations, like global areas, are automatically added to new address spaces.

`kGlobalAreaReservation`
: The areas created in the reservation are to be available globally. When you specify this option, you must also specify the `kGlobalReservation` option.

The Reservation Information Structure

The `GetReservationInformation` function (page 4-54) returns information about the specified reservation in the reservation information structure.

The reservation information data structure is defined by the `ReservationInformation` data type.

```
struct ReservationInformation {
    AddressSpaceID  addressSpace;
    LogicalAddress  base;
    ByteCount  length;
    ReservationOptions options;
};
typedef struct ReservationInformation ReservationInformation;
typedef ReservationInformation *ReservationInformationPtr;
```

Field descriptions

`addressSpace`	The ID of the address space in which the reservation exists. This is the current address space ID if the reservation is global to all address spaces.
`base`	The logical base address of the reservation.
`length`	The size, in bytes, of the reservation.
`options`	The options specified when the reservation was created. Possible values are defined by the reservation options enumeration (page 4-15).

Processor Cache Mode Enumeration

You use the `SetProcessorCacheMode` function (page 4-65) to specify the processor cache mode for a given range of addresses in an address space. Use one of the following enumeration values for the function's `processorCacheMode` parameter to specify the cache mode.

```
typedef UInt32 ProcessorCacheMode;
enum {
    kProcessorCacheModeDefault          = 0,
    kProcessorCacheModeInhibited        = 1,
    kProcessorCacheModeWriteThrough     = 2,
    kProcessorCacheModeCopyBack         = 3
};
```

Enumerator descriptions

`kProcessorCacheModeDefault`
The cache mode defined for the range.

`kProcessorCacheModeInhibited`
Data and/or code caching are not available.

`kProcessorCacheModeWriteThrough`
Read operations use the cache, but write operations are immediately apparent in physical memory.

`kProcessorCacheModeCopyBack`
Both read and write operations use the cache, so write operations may not be immediately apparent in physical memory.

Memory Preparation For I/O

When you want to prepare memory for use in I/O transfers, you use the `PrepareMemoryForIO` and the `CheckpointIO` functions in conjunction with the I/O preparation table to define how and where the preparation is to take place. The table includes two address range structures, and there are several sets of options you can use to further define the data transfers for which you wish to prepare.

I/O Preparation Options Enumeration

When you use the `PrepareMemoryForIO` function (page 4-67), you define some of the characteristics of I/O preparation by specifying one or more of the I/O preparation options enumeration values for the `options` field of the I/O preparation table.

```
typedef OptionBits IOPreparationOptions;
enum {
    kIOMultipleRanges           = 0x00000001,
    kIOLogicalRanges            = 0x00000002,
    kIOMinimalLogicalMapping    = 0x00000004,
    kIOShareMappingTables       = 0x00000008,
    kIOIsInput                  = 0x00000010,
    kIOIsOutput                 = 0x00000020,
    kIOCoherentDataPath         = 0x00000040,
    kIOClientIsUserMode         = 0x00000080
};
```

Option descriptions

`kIOMultipleRanges`	The `rangeInfo` field in the I/O preparation table refers to a multiple address range, enabling a scatter-gather specification.

`kIOLogicalRanges`	The `base` fields of the address range structures are logical addresses. If you omit this option, the addresses are treated as physical addresses.
`kIOMinimalLogicalMapping`	The logical mapping table is to be filled in with just the first and last static logical mappings of each range, arranged as pairs, instead of allocating a full-sized logical mapping table.There are two entries per range although the value of the second entry of the pair is undefined if the range is contained within a single page. Minimal mapping is useful for transfers where physical addresses are used for the bulk of the transfer, but logical addresses must be used to handle unaligned portions at the beginning and end.
`kIOShareMappingTables`	The microkernel can use the caller's mapping tables rather than maintain its own copies of the tables. Normally the `PrepareMemoryForIO` function keeps its own copy of the mapping tables in addition to the tables the driver allocates. You can reduce memory use if the driver shares its mapping tables with the microkernel. Use this option to specify that you want to share the tables. You can only specify this option if the mapping tables are located in a global resident area or a locked portion of a pageable area, and remain in that location until the final `CheckpointIO` function completes. Furthermore, the mapping tables must remain allocated and the entries unaltered until after the final `CheckpointIO` function completes. It is not necessary for the driver to provide both logical and physical tables.
`kIOIsInput`	Data will be moved into main memory. You can specify this option independently from the `kIOIsOutput` option: You can specify either, both, or neither at preparation time. Specifying neither is useful when the preparation must be made long in advance of the transfer (that is, so the system resources are allocated). You can then call the `CheckPointIO` function just before the transfer to prepare the caches.
`kIOIsOutput`	Data will be moved out of main memory. You can specify this option independently from the `kIOIsInput` option: You can specify either, both, or neither at preparation time.

Specifying neither is useful when the preparation must be made long in advance of the transfer (that is, so the system resources are allocated). You can then call the `CheckPointIO` function just before the transfer to prepare the caches.

`kIOCoherentDataPath`

The data path used to access memory during the I/O operation is fully coherent with the main processor's data caches, obviating the need for data cache manipulations. Coherency with the instruction cache is not implied, however, so the appropriate instruction cache manipulations are performed regardless.

This option is useful when the overall hardware architecture is not coherent, but the driver knows that the transfer will occur on a particular hardware path that is coherent. (The `PrepareMemoryforIO` function operates according to the overall architecture and has no detailed knowledge of individual data paths.)

When in doubt, omit this option. Incorrectly omitting this option merely slows the operation of the computer, whereas incorrectly specifying this option can result in erroneous behavior and crashes.

`kIOClientIsUserMode`

The `PrepareMemoryForIO` function is being called on behalf of a user mode client. If you specify this option, the system checks the memory range or ranges for user mode accessibility. If you do not specify this option, the system checks the memory range or ranges for privileged mode accessibility. Drivers can obtain the client's execution mode through the Family Programming Interface (FPI). In general, however, this information is available to message recipients in the message header.

I/O Checkpoint Options Enumeration

When you use the `CheckpointIO` function (page 4-69), you can define subsequent transfers by specifying one or more of the following I/O checkpoint options enumeration values for the parameter `theOptions`.

```
typedef OptionBits IOCheckpointOptions;
enum {
    kNextIOIsInput  = 0x00000001,
    kNextIOIsOutput = 0x00000002,
    kMoreIOTransfers = 0x00000004
};
```

Option descriptions

`kNextIOIsInput`	In the next transfer, data will be copied into main memory. You can specify this option independently from the `kNextIOIsOutput` option: You can specify either, both, or neither at preparation time. Specifying neither is useful for finalizing the previous transfer when the next transfer is not immediately pending.
`kNextIOIsOutput`	In the next transfer, data will be copied out of main memory. You can specify this option independently from the `kNextIOIsInput` option: You can specify either, both, or neither at preparation time. Specifying neither is useful for finalizing the previous transfer when the next transfer is not immediately pending.
`kMoreIOTransfers`	Further I/O transfers are to occur to or from the buffer. This option is especially useful when the caller is unable to specify in which direction the next transfer will be (that is, when neither the `kNextIOIsInput` nor `kNextIOIsOutput` option is specified), and is required if the next transfer direction is specified. If you do not specify the `kMoreIOTransfers` option, all microkernel resources associated with preparation are reclaimed, including any kernel-allocated subsidiary structures and the `IOPreparationID` identifier.

The I/O Preparation Table Structure

The `PrepareMemoryForIO` function (page 4-67) uses the I/O preparation table to obtain the address ranges to be prepared and to return mapping information about the ranges it has prepared.

The logical and physical mapping tables are where the function returns the addresses that the driver can use to access the client's buffer. The first entry of a range's mappings is the exact mapping of the first prepared address in that range, regardless of page alignment, while the remaining entries are page aligned. If you specify multiple address ranges, the mapping table is a concatenation, in order, of the mappings for each range.

There are no explicit length fields in the mapping tables. Instead, entry lengths are implied by the entry's position in the table, the overall range length, and the page size. In the general case, the length of the first entry is to the next page boundary, the length of any intermediate entries is the page size, and the length of the last element is what remains by subtracting the previous lengths from the overall range length. If the prepared range fits within a single page, there is only one prepared entry and its length is equal to the value in the `lengthPrepared` field.

The I/O preparation table structure is defined by the `IOPreparationTable` data type.

```
struct IOPreparationTable {
    IOPreparationOptions        options;
    IOPreparationState          state;
    IOPreparationID             preparationID;
    AddressSpaceID              addressSpace;
    ByteCount                   granularity;
    ByteCount                   firstPrepared;
    ByteCount                   lengthPrepared;
    ItemCount                   mappingEntryCount;
    LogicalMappingTablePtr      logicalMapping;
    PhysicalMappingTablePtr     physicalMapping;
    union {
        AddressRange            range;
        MultipleAddressRange    multipleRanges;
    } rangeInfo;
};

typedef struct IOPreparationTable IOPreparationTable;
```

Field descriptions

`options`	The optional characteristics of the I/O preparation table and the transfer. These values are defined by the I/O preparation options enumeration (page 4-17).
`state`	The state of the I/O preparation table. This value is filled in by the `PrepareMemoryForIO` function. A value of `kIOStateDone` indicates that the `PrepareMemoryForIO`

function successfully prepared up to the end of the specified range or ranges.

If insufficient resources are available to prepare the specified range of memory, the `PrepareMemoryForIO` function prepares as much as possible, indicates to the caller how much memory was prepared, and clears the `kIOStateDone` bit to indicate a partial preparation.

`preparationID` The identifier that represents the I/O transaction. This value is filled in by the `PrepareMemoryForIO` function. When the I/O operation has been completed or aborted, the `CheckpointIO` function uses this identifier to finalize the transaction.

`addressSpace` The address space containing the logical range or ranges to be prepared.

`granularity` A number of bytes. The `PrepareMemoryForIO` function uses this value in the event of a partial preparation. It is useful for transfers with devices that operate on fixed-length buffers. The length prepared is zero or an integral multiple of the value specified by the `granularity` field rounded up to the next greatest page alignment. This prevents preparing more memory than the caller is willing to use. A value of zero for `granularity` specifies no granularity. The system does not check whether the specified range length or lengths are multiples of `granularity`.

`firstPrepared` The byte offset into the range or ranges at which to begin preparation. Note that when a multiple address range is specified, this offset is into the aggregate range. You can use the `firstPrepared` and `lengthPrepared` fields to control partial preparations. The first time you call the `PrepareMemoryForIO` function, specify 0 for the `firstPrepared` field. If the `PrepareMemoryForIO` function does not return a `state` field value with `kIOStateDone`, a partial preparation was performed.

After this data is transferred and the final call to the `CheckPointIO` function is made against this preparation, you can make another call to the `PrepareMemoryForIO` function to prepare as much as possible of the range or ranges that remain. This time, specify a value for `firstPrepared` that is the sum of the current `firstPrepared` and `lengthPrepared`. You can repeat this sequence of prepare-transfer-final checkpoint until the `state` field has a value with `kIOStateDone`.

`lengthPrepared`
The number of bytes, starting at the offset specified by the `firstPrepared` field, that were prepared. This is filled in by the `PrepareMemoryForIO` function whether the preparation was partial or complete.

`mappingEntryCount`
The number of entries in the tables referenced by the `logicalMapping` or `physicalMapping` fields or both.

Normally the driver should allocate as many entries as there are pages in the buffer. You can calculate the number of pages in a memory range from the range's base address and length. If there are not enough entries, a partial preparation is performed within the limit of the tables, and the `kIOStateDone` state bit is returned as 0.

The logical mapping table is assumed to have two entries per range if you specified the `kIOMinimalLogicalMapping` option, regardless of the value in the `mappingEntryCount` field.

`logicalMapping`
The address of the logical mapping table. The table, which is filled in by the `PrepareMemoryForIO` function, must have as many entries as there are logical pages in the range or ranges. This table is optional. A `nil` value specifies that there is no table.

On return, the logical mapping table contains the **static logical addresses** corresponding to the ranges' physical addresses. The table is a concatenation, in order, of the mappings for each specified range. The first entry of each range's mappings is the exact static logical mapping of the first prepared address in that range, regardless of page alignment, while the remaining entries are page aligned. However, if you specified the `kIOMinimalLogicalMapping` option, on return this table contains just the first and last static logical mappings of each range.

The structure of static logical mappings is guaranteed on a per-page basis; a static logical mapping endures from the time you call the `PrepareMemoryForIO` function through the final call to `CheckpointIO` function.

Specifying a logical mapping table implies that the transfer is to be made through the main processor's MMU and data caches. Such transfers are performed with programmable I/O devices, and the `PrepareMemoryForIO` and `CheckpointIO` functions take this into account when determining which cache operations are needed before and after the transfer.

`physicalMapping` The address of the physical mapping table. The table, which is filled in by the `PrepareMemoryForIO` function, must have as many entries as there are logical pages in the range. This table is optional. A `nil` value specifies that there is no table.

On return, the `PhysicalMappingTable` contains the physical addresses that comprise the range or ranges. The table is a concatenation, in order, of the mappings for each specified range. The first entry of each range's mappings is the exact physical mapping of the first prepared address in that range, regardless of page alignment, while the remaining entries are page aligned.

Specifying a physical mapping table implies that the transfer bypasses the main processor's MMU and data caches. Such transfers are performed with DMA I/O devices, and the `PrepareMemoryForIO` and `CheckpointIO` functions take this into account when determining which cache operations are needed before and after the transfer.

`rangeInfo` The range or ranges to prepare.

If you specify a single range (the `kIOMultipleRanges` bit in the `options` field is clear), this field is an address range structure (page 4-26) specifying the base address and the length of the range.

If you specify multiple ranges (by setting the `kIOMultipleRanges` bit in the `options` field), the `rangeInfo` field is a multiple address range structure (page 4-26) specifying the number of ranges and the address of a table containing an entry for an address range structure identifying each range.

The `firstPrepared` field value determines the range table entry at which to begin preparation. Any entries positioned earlier in the table are not prepared and therefore do not need to have mapping table space allocated for them.

The Address Range Structure

When you use the `PrepareMemoryForIO` function (page 4-67)without specifying the `kIOMultipleRanges` option, the function uses the address range structure to identify the address range being prepared.

The address range structure is defined by the `AddressRange` data type.

```
struct AddressRange {
 void                       *base;
 ByteCount                  length;
};
typedef struct AddressRange AddressRange;
typedef struct AddressRange *AddressRangeTablePtr;
```

Field descriptions

`base`	The lowest address in the range. If you do not specify the `kIOLogicalRanges` option, this is treated as a physical address. If you specify the `kIOLogicalRanges` option, this is treated as a logical address within the address space specified in the I/O preparation table.
`length`	The length of the range, in bytes.

The Multiple Address Range Structure

When you use the `PrepareMemoryForIO` function (page 4-67) and specify the `kIOMultipleRanges` option, the function uses the multiple address range structure to identify the array of address ranges being prepared.

A driver specifying a user buffer that consists of multiple ranges (scatter-gather buffer) can use the multiple address range structure to identify the ranges.

The multiple address range structure is defined by the `MultipleAddressRange` data type.

```
struct MultipleAddressRange {
 ItemCount                  entryCount;
 AddressRangeTablePtr       rangeTable;
};
typedef struct MultipleAddressRange MultipleAddressRange;
```

Field descriptions

`entryCount`	The number of entries in the address range table referenced in the `rangeTable` field.
`rangeTable`	The address of an array of address range structures—that is, an address range table. The specified ranges can overlap either directly or indirectly by being located on the same pages.

Memory Management Functions

You use the functions described in this section to manage address spaces, areas, and pages, to share memory, to control caching operations and to prepare memory for I/0.

Managing Address Spaces

The microkernel provides functions that you can use to create and destroy address spaces. The contents of a newly created address space are based on a template maintained by the mcirokernel. This template causes frame buffers, ROM, and the microkernel to be mapped at the same location in each address space, with appropriate access protection.

This section describes the functions you use to create an address space, to obtain information about address spaces, and to destroy an address space.

CreateAddressSpace

Creates an address spaces and returns an identifier for it.

```
OSStatus CreateAddressSpace (AddressSpaceID *theAddressSpace);
```

`theAddressSpace`

A pointer to the address space ID. On output, specifies the address space identifier that you can use to obtain information about the address space or to destroy the address space.If the microkernel is unable to create an address space, it returns `kInvalidID` for this parameter.

function result To be provided.

DISCUSSION

The `CreateAddressSpace` function builds a new address space and returns an `ID` for it. A new address space automatically contains any existing global areas and memory reservations.

An application that needed additional logical space could also use this function to create another address space. It could then access that space by using the `CreateArea` or `CreateAreaForRange` function and the `InterspaceBlockCopy` function.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To create areas in an address space, you can use the `CreateArea` function (page 4-33) or the `CreateAreaForRange` function (page 4-57).

To create a memory reservation, you can use the `CreateAreaReservation` function (page 4-52).

To delete an address space, you can use the `DeleteAddressSpace` function (page 4-45).

To obtain information about address spaces you can use the `GetSpaceInformation` function (page 4-29), the `CurrentAddressSpaceID` function (page 4-30), or the `GetAddressSpacesInSystem` function (page 4-31).

GetSpaceInformation

Returns information about a specified address space.

```
OSStatus GetSpaceInformation (AddressSpaceID theAddressSpace,
                              PBVersion version,
                              SpaceInformation *spaceInfo);
```

`theAddressSpace`
The address space for which to get the information.

`version`
The version number of the space information structure to be returned. This provides backwards compatibility. The constant `kSpaceInformationVersion` supplies the version of the space information structure defined in the interface.

`spaceInfo` A pointer to the space information structure (page 4-5) in which to return the information.

function results To be provided.

DISCUSSION

This function returns the following information:

- The number of logical pages in this address space, excluding pages in global areas and pages in guard ranges. This is the same as summing the existing area sizes.
- The number of logical pages in this address space that are currently in physical memory, excluding pages in global areas
- The number of pages in this address space that are locked in physical memory.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To obtain additional information about address spaces you can use the `CurrentAddressSpaceID` function (page 4-30), or the `GetAddressSpacesInSystem` function (page 4-31).

CurrentAddressSpaceID

Returns the address space ID of the current address space.

```
AddressSpaceID CurrentAddressSpaceID (void);
```

function result The ID of the current address space.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To delete an address space, you can use the `DeleteAddressSpace` function (page 4-45).

To obtain additional information about address spaces you can use the `GetSpaceInformation` function (page 4-29) or the `GetAddressSpacesInSystem` function (page 4-31).

GetAddressSpacesInSystem

Returns the address space ID of each existing address space in the system.

```
OSStatus GetAddressSpacesInSystem (ItemCount requestedAddressSpaces,
                    ItemCount *totalAddressSpaces,
                    AddressSpaceID *theAddressSpaces);
```

`requestedAddressSpaces`
: The maximum number of address space IDs that are to be returned. The function uses this value to provide adequate space at the location referenced by the parameter `theAddressSpaces`.

`totalAddressSpaces`
: On output, a pointer to the total number of address spaces in the system. If this is less than or equal to the value referenced by the `requestedAddressSpaces` parameter, all address spaces

were returned; if this is greater than the value referenced by `requestedAddressSpaces` parameter, insufficient space was available to return all address space IDs.

`theAddressSpaces`
On output, a pointer to a buffer containing the IDs of all the address spaces in the system.

function result To be provided.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To obtain information about address spaces you can use the `GetSpaceInformation` function (page 4-29) or the `CurrentAddressSpaceID` function.

To delete an address space, you can use the `DeleteAddressSpace` function (page 4-32).

DeleteAddressSpace

Destroys an address space.

```
OSStatus DeleteAddressSpace (AddressSpaceID theAddressSpace);
```

`theAddressSpace`
The ID of the address space to destroy.

SPECIAL CONSIDERATIONS

The `DeleteAddressSpace` function also destroys all nonglobal areas mapped into the specified address space.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To create areas in an address space, you can use the `CreateArea` function (page 4-33).

To obtain information about address spaces you can use the `GetSpaceInformation` function (page 4-29), the `CurrentAddressSpaceID` function (page 4-30), or the `GetAddressSpacesInSystem` function (page 4-31).

Managing Areas

You use the functions described in this section to create areas within an address space, to obtain information about areas, and to delete areas.

CreateArea

Creates an area and returns an identifier for it.

```
OSStatus CreateArea (KernelProcessID owningKernelProcess,
                     BackingObjectID backingObject,
                     Const BackingAddress *backingBase,
                     ByteCount  backingLength,
                     MemoryAccessLevel userAccessLevel,
                     MemoryAccessLevel privilegedAccessLevel,
                     ByteCount  guardLength,
                     AreaOptions  options,
                     LogicalAddress *areaBase,
                     AreaID *theArea);
```

`owningKernelProcess`
The kernel process in whose address space to create the area. Areas are automatically deleted when their owning kernel process terminates.

`backingObject`
The ID of the backing store whose content is to be mapped. Specifying a value of `kNobackingObjectID` for this parameter specifies that a scratch backing store file is to be used. If you specify `kResidentArea` for the `options` parameter or if all access to the area is excluded, this parameter must have a value of `kNobackingObjectID`.

You obtain a backing object ID from the file system function you used to create the backing object.

`backingBase` A pointer to the offset within the backing store specified by the `backingObject` parameter, which corresponds to the area's base address. To specify an offset of 0, specify `nil` for this parameter.

The range of possible `BackingAddress` values is not constrained by the memory system. Backing objects themselves may place restrictions (for example, on a block-oriented device, the base might need to be a whole multiple of the block size).

If you specify `kResidentArea` in the `options` parameter, you must use a value of `nil` for the `backingBase` parameter.

`backingLength` The number of bytes to map from the backing store, starting at the offset referenced by the `backingBase` parameter. The microkernel rounds this number up to a multiple of the logical page size, which implies that more backing store than you specified may be mapped in. The `backingLength` parameter must have a value other than 0.

`userAccessLevel`

The kinds of memory references that user mode software is allowed to make in the area. Use the memory access level enumeration (page 4-6) to specify a value for this parameter. References made in violation of the access level result in exceptions at the time of the access. There are processor-dependent restrictions on the access level combinations you can specify. Call the `GetAreaInformation` function (page 4-41) to find out what is allowable.

To create an area that excludes any access, set both user and privileged access to `kMemoryExcluded` and specify both the `kSparseArea` and `kResidentArea` area options. Specifying these options ensures that physical memory and disk space are not assigned to the area. If you plan on changing the area's access level, you do not have to specify these options.

`privilegedAccessLevel`

The kinds of memory references that privileged mode software is allowed to make in the area. Use the memory access level enumeration (page 4-6) to specify a value for this parameter. References made in violation of the access level result in exceptions at the time of the access. There are processor-dependent restrictions on the access level combinations you can specify. Call the `GetAreaInformation` function (page 4-41) to find out what is allowable.

If this parameter has a more restrictive value than that for the `userAccessLevel` parameter, the microkernel sets the `privilegedAccessLevel` parameter to the same value as that in the `userAccessLevel` parameter.

To create an area that excludes any access, set both user and privileged access to `kMemoryExcluded` and specify both the `kSparseArea` and `kResidentArea` area options. Specifying these options ensures that physical memory and disk space are not assigned to the area. If you plan on changing the area's access level, you do not have to specify these options.

`guardLength` The size in bytes of the area's guard range—that is, the excluded logical address ranges to place adjacent to each end of the area. The guard ranges exclude both privileged and user mode software, and references to addresses in these ranges

	result in exceptions. The microkernel aligns guard ranges to page boundaries, which means that the excluded ranges may be larger than you specified.
`options`	The desired characteristics of the area being created. Use the area options enumeration (page 4-7) to specify values for this parameter.
`areaBase`	On output, a pointer to the beginning logical address of the mapped memory. If you specify the `kPlacedArea` option, the `AreaBase` parameter also has a meaning as an input parameter: it is a pointer to the location where you want the area to be positioned. Please note that, for alignment reasons, the actual location of the area might differ from the requested value. In accessing this location, you should use the value returned in the `areaBase` parameter, not simply use the saved requested value.
`theArea`	On output, the area identifier to be used for subsequent operations on the created area.
function result	To be provided.

DISCUSSION

The `CreateArea` function creates a mapping between a portion of the specified address space and the specified backing store. The function returns the ID of the newly created area and the logical address of that area's origin. The logical address has meaning only within the context of the area's own address space.

You use the options parameter to specify the attributes of the area to be created. You can specify that

- memory in the area be initialized to 0
- the data for this area cannot be paged out
- resources for the area be allocated on demand
- the area be positioned at a specific logical address
- data in the area be addressable from any address space

SPECIAL CONSIDERATIONS

Always use the address returned in the `areaBase` parameter as the area's starting address.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To change an area's access levels, use the `SetAreaAccess` function (page 4-37).

To create areas that share ranges of addresses, use the `CreateAreaForRange` function (page 4-37).

To delete an area use the `DeleteArea` function (page 4-45).

To obtain information about an area use the `GetAreaInformation` function (page 4-41).

SetAreaAccess

Changes the access rights for an area.

```
OSStatus SetAreaAccess (AreaID theArea,
                        MemoryAccessLevel userAccessLevel,
                        MemoryAccessLevel privilegedAccessLevel);
```

`theArea`	The ID of the area whose access you are changing.

`userAccessLevel`
: The kinds of memory references that user mode software is allowed to make in the area. Reference kinds are defined by the memory access level enumeration (page 4-6). References made in violation of the access level result in exceptions at the time of the access.

`privilegedAccessLevel`
: The kinds of memory references that privileged mode software is allowed to make in the area. Reference kinds are defined by the memory access level enumeration (page 4-6). References made in violation of the access level result in exceptions at the time of the access. If this parameter has a more restrictive value than that for user mode software, the `privilegedAccessLevel` parameter is set to the same value as that specified by the `userAccessLevel` parameter.

function results To be provided.

DISCUSSION

There are processor-dependent restrictions on the access level combinations you can specify. Call the `GetAreaInformation` function (page 4-41) to find out what is allowable.

To create an area that excludes any access, set both user and privileged access to `kMemoryExcluded` and specify both the `kSparseArea` and `kResidentArea` area options. Specifying these options ensures that physical memory and disk space are not assigned to the area. If you plan on changing the area's access level, you do not have to specify these options.

It is sometimes useful to change the kind of access that is allowed to an area. For example, a code loader might need to make an area read-write while initializing it, then change it to read-only when the area is ready to use.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

Use the `GetAreaInformation` function (page 4-41) to determine an area's current access levels.

CheckUserAccess

Checks the legality of different kinds of operations occurring within a specified range of memory.

```
OSStatus CheckUserAccess (ConstLogicalAddress address,
                          ByteCount length,
                          MemoryReferenceKind referenceKind);
```

`address`	The beginning address of the range you wish to check.
`length`	The size in bytes of the range you want checked.
`referenceKind`	The kind of operation you wish to verify. Possible values for this parameter are given in the chapter "Exception Handling."
function result	To be provided.

DISCUSSION

The `CheckUserAccess` function, given an address, a length, and a kind of reference, returns a status that specifies whether the proposed reference is valid in user mode.

This `CheckUserAccess` function is useful in a situation like the following. A user mode client calls a privileged server which, in turn, calls an accept function in the client's address space. You want the accept function to store data back in the client's buffer, but you do not want to override the client's access mode. To do this, you call the `CheckUserAccess` function to make sure that you will not accidentally overwrite data that the client itself could not have done.

Note that validating the proposed access does not guarantee a successful access; it might still fail because the access is not synchronized with the client.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To copy data safely, use the `InterspaceBlockCopy` function (page 4-61).

DeleteArea

Destroys an area.

```
OSStatus DeleteArea (AreaID theArea);
```

`theArea`	The area to destroy.

function result To be provided.

DISCUSSION

The `DeleteArea` function removes the specified area. Further references to the logical addresses previously mapped to the deleted area result in memory exceptions.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Obtaining Information About An Area

You use the functions described in this section to obtain the location, backing store, and attributes of an area, to obtain the IDs of all areas in an address space, or to obtain the ID of an area, given a logical address.

GetAreaInformation

Returns information about a specified area.

```
OSStatus GetAreaInformation (AreaID theArea,
                    PBVersion version,
                    AreaInformation *areaInfo);
```

`theArea`	The ID of the area for which you seek information.

`version`	The version number of the area information structure to be returned. This provides backwards compatibility. The constant `kAreaInformationVersion` supplies the version of the area information structure defined in the interface.
`areaInfo`	A pointer to the area information structure (page 4-9) in which the information is returned.
function result	To be provided.

DISCUSSION

The area information structure referenced by the `areaInfo` parameter contains the following information:

- the ID of the address space containing the area,
- the base address and size of the area
- the kinds of references allowed to user-mode and privileged tasks accessing data in the area,
- the area's usage
- the ID of the object providing backing store for the area and the area's base address within the backing object,
- the options specified when the area was created and
- the ID of the kernel process specified when the area was created and whose termination causes the area to be deleted.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

Use the function `GetAreasInAddressSpace` (page 4-43) to obtain the IDs of all areas in an address space.

GetAreasInAddressSpace

Obtains the IDs of all areas in a specified address space

```
OSStatus GetAreasInAddressSpace (AddressSpaceID addressSpace,
                                 ItemCount requestedAreas,
                                 ItemCount *totalAreas,
                                 AreaID *theAreas);
```

`addressSpace`	The ID of the address space of interest.
`requestedAreas`	The maximum number of area IDs to be returned. This indicates the number of entries available at the location referenced by the parameter `theAreas`.
`totalAreas`	On output, a pointer to the total number of areas within the address space. If this is less than or equal to the value in the `requestedAreas` parameter, all areas were returned; if this is greater than the `requestedAreas` parameter, insufficient space was available to return all area IDs; but as many as possible were returned.
`theAreas`	On output, the IDs of the areas within the address space.
function result	The function returns the kernedIDErr status if the specified address space does not exist.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To obtain information about a single area, use the `GetAreaInformation` function (page 4-41).

GetAreaFromAddress

Obtains the area ID of the area associated with the specified logical address.

```
OSStatus GetAreaFromAddress (AddressSpaceID addressSpace,
                             ConstLogicalAddress address,
                             AreaID *theArea);
```

`addressSpace`	The address space containing the logical address specified by the `address` parameter.
`address`	The logical address to look up.
`theArea`	On output, a pointer to the ID of the area in question.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To get information about an area call the GetAreaInformation function (page 4-41).

Working With Backing Storage

This section describes two functions that you use to manage virtual memory operations relating to backing storage. You use the `SetAreaBackingBase` function to determine what portion of a backing store object is mapped to logical memory in those cases where the backing store object is larger than the address space. You use the `ReleaseData` function to let the microkernel know when it is not necessary to copy data to the backing store.

SetAreaBackingBase

Sets the specified backing address as the base for the specified area.

```
OSStatus SetAreaBackingBase (AreaID theArea,
                      const BackingAddress *backingBase);
```

`theArea`	The ID of the area in which to change the backing store base.
`backingBase`	A pointer to the offset within the backing store that is to correspond to the lowest address in the area. Specifying `nil` for `backingBase`, means a backing address of 0.

The range of possible `BackingAddress` values is not constrained by the memory system. Backing objects themselves may place restrictions (for example, on a block-oriented device, the base might need to be a whole multiple of the block size).

DISCUSSION

Some backing store objects are too large to view in their entirety in the space available to a single address space. A common way to deal with this limitation is to create a limited-size mapping (area) and then adjust where in the backing store that mapping corresponds.

The `SetAreaBackingBase` function sets the specified address as the base for the specified area. An area's base backing address and length determine which portion of the backing object is mapped to the area.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

ReleaseData

```
OSStatus ReleaseData (AddressSpaceID addressSpace,
                      ConstLogicalAddress base,
                      ByteCount length,
                      ReleaseDataOptions options);
```

`addressSpace`	The ID of the address space containing the range to release.
`base`	The base address of the range to release.

`length`	The number of bytes in the range to release. The system adjusts the beginning and end of the range as necessary so that the range released begins and ends on logical page boundaries. This means that *less* memory than was specified may be released.
`options`	The options available for this function. In this case, there is only the `kReleaseBackingStore` option to deallocate the backing store. Specifying this option causes the backing store associated with the range to be deallocated if appropriate (the backing store won't be deallocated if the backing object was opened with read only access, for instance). Specifying this option frees backing store space but increases the runtime cost of the operation and possibly incurs future costs when the page is touched again and backing store must be reallocated.

DISCUSSION

The `ReleaseData` function informs the memory system that the data values in the specified range are no longer needed. It is an optimizing hint to prevent writing the data to the backing store. The backing store, if any, remains allocated to the range. This is useful, for example, when deallocating modified memory that is no longer reflected in the backing store.

If the released range is subsequently accessed, the values in memory will be unpredictable.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Managing Pages

This sections describes the `GetPageInformation` function, which returns information about each logical page in a range and the `ControlPagingForRange` function, which you use to lock one or more logical pages in memory.

GetPageInformation

Obtains information for each logical page within a range of logical addresses.

```
OSStatus GetPageInformation (AddressSpaceID addressSpace,
                     ConstLogicalAddress base,
                     ItemCount requestedPages,
                     PBVersion version,
                     PageInformation *thePageInfo);
```

`addressSpace`	The ID of the address space containing the range of interest.
`base`	The base logical address of interest.
`requestedPages`	The number of pages for which information is to be returned.
`version`	The version number of the page information structure to be returned. This provides backwards compatibility. The constant `kPageInformationVersion` supplies the version of the page information structure defined in the interface.
`thePageInfo`	A pointer to a buffer containing a page information structure (page 4-14) describing the information returned. The buffer must be large enough to contain page state information (page 4-11) for each page specified in the `requestedPages` parameter.
function results	To be provided.

DISCUSSION

This function returns a pointer to a buffer that contains the following entries:

- The first field specifies the ID of the area associated with the range.

- The second field specifies the number of page state information entries returned.
- The third field is an array of page state information structures, one structure for each logical page in the specified range, which specify whether
  - the page protection for user mode and privileged software
  - the page has been modified or referenced
  - the page is locked and resident
  - the page is present in physical memory
  - the page is being shared
  - modifications to the page are written through the processor cache to main memory or whether modifications to the page may be cached by the processor and not immediately reflected in main memory.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

ControlPagingForRange

Allows programmatic control of the paging operations on a specified range.

```
OSStatus ControlPagingForRange (AddressSpaceID addressSpace,
                    ConstLogicalAddress base,
                    ByteCount length,
                    PageControlOperation operation);
```

`addressSpace`	The ID of the address space containing the range to change.
`base`	The base address of the range to change.

`length`	The number of bytes in the range. The beginning and end of the range are adjusted, if necessary, so that the range begins and ends on logical page boundaries. This means that more memory than you specified may be affected.
`operation`	The paging operation. Values for this parameter are defined by the page control operation enumeration (page 4-12).

DISCUSSION

You can call the `ControlPagingForRange` function to perform the following operations on the specified range:

- lock and unlock the range
- allocate backing store in a sparse pageable area or allocate physical memory for a page in a sparse resident area
- bring the range into physical memory asynchronously
- free physical memory associated with this range after writing modified pages to backing store. You can use this to optimize performance by giving the system information about pages that will not be needed in the near future.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

Call the `GetPageInformation` function to obtain information about pages in a range of memory.

Sharing Memory

Device drivers, debuggers, and client-server applications must often share data across address spaces. The underlying method for sharing memory is to create areas that map the same backing store data into the various clients' address spaces. The microkernel provides various services to share memory using this method:

- To create a **global area**—that is, an area that appears in every address space at the same location and with the same attributes, use the `kGlobalArea` option when creating the area with the `CreateArea` function (page 4-33).
- To create an area that can be shared between clients and servers, each having different access rights to the shared data, use the `CreateAreaForRange` function.
- To reserve space that can then be accessed from different address spaces use the `CreateAreaReservation` function.

This section describes the functions you use to reserve memory, to free a reserved memory range, to obtain information about a memory reservation, and to obtain the IDs of all memory reservations made in an address space.

- To allow your application ongoing access to data in other address spaces use the `CreateAreaForRange` function described in this section.
- To enable your application to simply read data from or write data to other address spaces, use the `InterspaceBlockCopy` function described in this section.

CreateAreaReservation

Reserves a logical address range.

```
OSStatus CreateAreaReservation (KernelProcessID owningKernelProcess,
                     LogicalAddress *reservationBase,
                     ByteCount length,
                     ReservationOptions options,
                     AreaReservationID *theReservation);
```

`owningKernelProcess`	The ID of the kernel process in whose address space to reserve the range.
`base`	On output, a pointer to the base logical address of the reservation. If you specify the `kPlacedReservation` option, this parameter is also an input specifying where to position the reservation. If the reservation cannot be placed at the requested location, the `CreateAreaReservation` function fails and an error is returned. If the reservation is made, the output and input values of the reservation's base address might not be the same because of page alignment. You should always use the value returned in the `base` parameter rather than saving your requested value and using it.
`length`	The number of bytes to reserve. Due to page alignment the reservation may be larger than you specified.
`options`	The optional characteristics of the reservation being created. You specify values for this parameter using the reservation options enumeration (page 4-15).
`theReservation`	On output, a pointer to the reservation identifier to be used for subsequent operations on the reservation.
function results	To be provided.

DISCUSSION

The `CreateAreaReservation` function cordons off a logical address range such that no areas can be created within that range unless they are specified to be there by using the `kPlacedArea` option to the `CreateArea` or `CreateAreaForRange` functions.

You use the `options` parameter to specify whether

- the reservation is to occur at a particular location
- the reservation is to apply across all existing and future address spaces—the reservation is global
- any areas created in the reserved range are also to be global

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

Use the `DeleteAreaReservation` function (page 4-53) to make a reserved area pageable.

Use the `GetReservationInformation` function (page 4-54) to obtain information about a reservation.

Use the `GetReservationsInAddressSpace` function (page 4-56) to obtain the IDs of all the reserved areas in an address space.

DeleteAreaReservation

Destroys a specified memory reservation.

```
OSStatus DeleteAreaReservation (AreaReservationID theReservation);
```

`theReservation`
The reservation to delete.

function results To be provided.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

Use the `GetReservationInformation` function (page 4-54) to obtain information about a reservation.

Use the `GetReservationsInAddressSpace` function (page 4-56) to obtain the IDs of all the reserved areas in an address space.

GetReservationInformation

Obtains information about a specified memory reservation.

```
OSStatus GetReservationInformation (AreaReservationID theReservation,
                     PBVersion version,
                     ReservationInformation *reservationInfo);
```

`theReservation`
The ID of the memory reservation for which to get the information.

`version`	The version number of the memory reservation information structure to be returned. This provides backwards compatibility. The constant `kReservationInformationVersion` defines the version of the memory reservation information structure defined in the current interface.
`reservationInfo`	A pointer to the memory reservation information structure in which to return the information.
function results	To be provided.

DISCUSSION

The `GetReservationInformation` function returns the following information about a reservation:

- The ID of the address space in which the reservation is made.
- the base logical address of the reservation
- the size of the reservation
- the options specified when the reservation was made: whether the reservation was placed, whether it is global, and whether the areas it includes are global.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

Use the `GetReservationsInAddressSpace` function (page 4-56) to obtain the IDs of all the reserved areas in an address space.

GetReservationsInAddressSpace

Obtains the IDs of all memory reservations in a specified address space

```
OSStatus GetReservationsInAddressSpace (AddressSpaceID addressSpace,
                     ItemCount requestedReservations,
                     ItemCount *totalReservations,
                     AreaReservationID *theReservations);
```

`addressSpace`	The ID of the address space of interest.
`requestedReservations`	The maximum number of reservation IDs to be returned. This indicates the number of entries available at the location pointed to by the parameter `theReservations`.
`totalReservations`	On output, a pointer to the total number of reservations within the address space. If this is less than or equal to the value in the `requestedReservations` parameter, all reservations were returned; if this is greater than the value specified with the `requestedReservations` parameter, insufficient space was available to return all reservation IDs, but as many as possible were returned.
`theReservations`	On output, a pointer to a buffer containing the IDs of the reservations within the address space.
function results	If the specified address space does not exist, the function returns the `kernelIDErr` status.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

Use the `DeleteAreaReservation` function (page 4-53) to make a reserved area pageable.

Use the `GetReservationInformation` function (page 4-54) to obtain information about a reservation.

CreateAreaForRange

Maps a logical address range from one address space into another.

```
OSStatus CreateAreaForRange (KernelProcessID owningKernelProcess,
                     AddressSpaceID otherSpace,
                     ConstLogicalAddress otherBase,
                     ByteCount length,
                     MemoryAccessLevel userAccessLevel,
                     MemoryAccessLevel privilegedAccessLevel,
                     ByteCount guardLength,
                     AreaOptions options,
                     LogicalAddress *areaBase,
                     AreaID *theArea);
```

`owningKernelProcess`

The kernel process in whose address space to create the area. When the kernel process terminates, the area is automatically deleted.

`otherSpace` The address space containing the range to map.

`otherBase` The start of the range of memory that you want to map. The system aligns the value you specify for the `otherBase` parameter to a page boundary, which means that a larger range than you specified may be mapped.

`length` The number of bytes in the range of memory you want to map. The system aligns `length` values to a page boundary, which means that a larger range than you specified may be mapped.

`userAccessLevel`

The kinds of memory references that user mode software is allowed to make in the area. Use the memory access level enumeration (page 4-6) to specify a value for this parameter. References made in violation of the access level result in exceptions at the time of the access. There are processor-dependent restrictions on the access level combinations you can specify. Call the `GetAreaInformation` function (page 4-41) to find out what is allowable.

To create an area that excludes any access, set both user and privileged access to `kMemoryExcluded` and specify both the `kSparseArea` and `kResidentArea` area options. Specifying these options ensures that physical memory and disk space are not assigned to the area. If you plan on changing the area's access level, you do not have to specify these options.

You can specify that the user mode access rights in the area to be created area inherit either the user or the privileged mode access rights of the existing area. This is useful if the code calling the `CreateAreaForRange` function might be running at a different access level than the code that was accessing the original area. For example, if a privileged server is servicing requests from a user-mode client and the server must create an area in response to a request, it would not want to augment the client's access rights to the area.

`privilegedAccessLevel`

The kinds of memory references that privileged mode software is allowed to make in the area. Use the memory access level enumeration (page 4-6) to specify a value for this parameter. References made in violation of the access level result in exceptions at the time of the access. There are processor-dependent restrictions on the access level combinations you can specify. Call the `GetAreaInformation` function (page 4-41) to find out what is allowable.

If this parameter has a more restrictive value than that for the `userAccessLevel` parameter, the microkernel sets the `privilegedAccessLevel` parameter to the same value as that in the `userAccessLevel` parameter.

You can specify that the privileged mode access rights in the area to be created inherit either the user or the privileged mode access rights of the existing area. This is useful if the code calling the `CreateAreaForRange` function is running at a different access level than the code that was accessing the original area.

`guardLength`	The size in bytes of the area's guard ranges—that is, the excluded logical address range to place adjacent to each end of the area. The guard ranges exclude both privileged and user mode software, and references to addresses in these ranges result in exceptions. The microkernel aligns guard ranges to page boundaries, which means that the excluded ranges might be larger than you specified.
`options`	The desired characteristics of the area being created. Use the area options enumeration (page 4-7) to specify values for this parameter.

The area to be created inherits the following option values from the existing area: `kZeroFill`, `kResidentArea`, and `kSparseArea`. If you specify different values for these options, the `CreateAreaForRange` function ignores them.

Specify the `kPlacedArea` option to position the area at the address referenced by the `areaBase` parameter. The `CreateAreaForRange` function fails if the area cannot be positioned as specified.

The address corresponding to the beginning of the range is returned in the `areaBase` parameter. Note that this is exactly as specified only if areaBase and otherBase have the same byte offset into their respective logical pages.

`areaBase`	On output, a pointer to the beginning logical address of the mapped memory.

	If you specify the `kPlacedArea` option, the `areaBase` parameter also has a meaning as an input parameter: it is a pointer to the location where you want the area to be positioned. Please note, that the actual location of the area might differ from the requested value. In accessing this location, you should use the value returned in the `areaBase` parameter, not simply use the saved requested value.
`theArea`	On output, a pointer to the area identifier that you can use for subsequent operations on the created area.
function results	To be provided.

DISCUSSION

The `CreateAreaForRange` function creates a mapping between two address spaces or between two spaces in one address space and it returns the ID of the newly created area and its starting logical address.

It is sometimes useful to have on-going access to data in other address spaces. Although you can do this by calling the `GetAreaInformation` function to obtain information about an area and then using the `CreateArea` function to map the same backing store to another area, this takes several calls and does not work if an area is resident. Using the `CreateAreaForRange` function combines those several steps into one and allows you to map a resident area to another address space.

If you need to read data from or write data to another address space, you might find it more efficient to use the `InterspaceBlockCopy` function (page 4-61) rather than creating a mapping using the `CreateAreaForRange` function.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

Use the functions described in the section "Obtaining Information About An Area," beginning on page 4-41 to get information about one or more areas.

Use the `InterspaceBlockCopy` function (page 4-61) to make a static copy of the contents of a memory range.

InterspaceBlockCopy

Copies bytes from the specified source address space and range to the specified destination address space and range.

```
OSStatus InterspaceBlockCopy (AddressSpaceID sourceAddressSpace,
                   AddressSpaceID targetAddressSpace,
                   LogicalAddress sourceBase,
                   LogicalAddress targetBase,
                   ByteCount length
                   InterspaceCopyOptions options);
```

`sourceAddressSpace`	The ID of the address space containing the source range.
`targetAddressSpace`	The ID of the address space containing the destination range.
`sourceBase`	The start of the source range.
`targetBase`	The start of the destination range.
`length`	The number of bytes to copy from the source to the destination.
`options`	The access checks to apply to the copy operation. Values for this parameter are defined by the interspace copy options enumeration (page 4-15).
function result	To be provided.

DISCUSSION

It is sometimes useful to read data from or write data to another address space. For example, a server might need to read or write client data, or a debugger might need to display or set data in the debugged address space. The

InterspaceBlockCopy function copies bytes from a specified source address space and range to a specified destination address space and range without the overhead of setting up a mapping and without the risk of encountering a memory access exception. Note that neither address space needs to be the current address space.

Calling the InterSpaceBlockCopy function is only economical when moving data across address spaces. If you need to move a block of memory safely within the same address space, it is recommended that you do not call this function but that you install an exception handler to guard against changes in access levels between the time you check for access rights and the time you move memory.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

Use the CreateAreaForRange function to create a mapping between two address spaces.

Working With Processor Caches

The functions described in this section might be useful to compilers, debuggers, and drivers.

If you are interested in maintaining cache coherency before or after an I/O operation, you need only to use the functions PrepareMemoryforIO and CheckpointI/O which ensure cache coherency when writing to or reading from external devices.

DataToCode

Enables execution of generated or copied instructions.

```
OSStatus DataToCode (AddressSpaceID addressSpace,
                     ConstLogicalAddress base,
                     ByteCount length);
```

`addressSpace`	The ID of the address space containing the range to be treated as code. This must be the current address space.
`base`	The start of the range to be treated as code.
`length`	The number of bytes in the range to be treated as code.
function results	To be provided.

DISCUSSION

The `DataToCode` functions performs the operations necessary for the specified memory range to be treated as processor instructions instead of simple data. This is required, for example, when reading instructions into scratch memory, or when generating instructions on the fly.

Placing executable data in memory requires synchronization with the processor's data and instruction caches. The details are specific to the processor and the internal operation of the memory system. Consequently, the virtual memory system provides this service, which encapsulates the necessary operations.

SPECIAL CONSIDERATIONS

The system adjusts the beginning and end of the range as necessary so that the range begins and ends on logical page boundaries, which means that more memory than was specified may be affected.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

FlushRange

Writes the specified data from the processor cache into main memory.

```
OSStatus FlushRange (AddressSpaceID addressSpace,
                     ConstLogicalAddress base,
                     ByteCount length);
```

`addressSpace`	The ID of address space containing the address range to be flushed. This must have the value of the current address space.
`base`	The start of the range of addresses to be written to memory.
`length`	The number of bytes in the address range to memory.
function results	To be provided.

SPECIAL CONSIDERATIONS

The system adjusts the beginning and end of the range as necessary so that the range begins and ends on logical page boundaries, which means that more memory than was specified may be affected.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SetProcessorCacheMode

Sets the memory hardware cache mode for the specified range.

```
OSStatus SetProcessorCacheMode (AddressSpaceID addressSpace,
                     ConstLogicalAddress base,
                     ByteCount length,
                     ProcessorCacheMode processorCacheMode);
```

`addressSpace`	The address space containing the range to change.
`base`	The start of the range.
`length`	The number of bytes in the range.
`processorCacheMode`	The cache mode. Values for this parameter are defined by the processor cache mode enumeration (page 4-17).
function results	To be provided.

DISCUSSION

You use this function to specify whether

- Data and/or code caching are available
- Read and write operations use the cache

SPECIAL CONSIDERATIONS

The system adjusts the beginning and end of the range as necessary so that the range begins and ends on logical page boundaries, which means that more memory than was specified may be affected.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Preparing For I/O

Memory usage in a demand paged, multi-tasking system is both highly dynamic and highly complex. When an I/O operation is performed between an external device and a buffer in system memory, the main processor, cache memory and the memory hardware must be coordinated.The microkernel provides two functions that you use to achieve this coordination. The `PrepareMemoryForIO` function assigns physical memory to the buffer, generates an appropriate buffer specification, and performs all necessary cache manipulation prior to each I/O operation. The `CheckpointIO` function cleans up following the I/O operation: it assures cache coherency and either prepares for further transfers or deallocates the resources associated with the buffer preparation if no further transfers will be made.

IMPORTANT

Failure to use these I/O-related microkernel services properly can result in data corruption or fatal system errors, or both. Correct system behavior is the responsibility of the microkernel and all I/O components including drivers, managers, and hardware. ▲

PrepareMemoryForIO

Prepares memory for use in I/O operations.

```
OSStatus PrepareMemoryForIO (IOPreparationTable *theIOPreparationTable);
```

`theIOPreparationTable`
: On input, a pointer to an I/O preparation table (page 4-21) indicating the memory buffer to be prepared. On output, a pointer to the same table, now also containing the mapping information.

function result To be provided.

DISCUSSION

The `PrepareMemoryForIO` function ensures that device I/O on one or more ranges is coordinated with the microkernel, the main processor caches, and other data transfers. Preparation includes ensuring that physical memory is assigned, and remains assigned, to the range at least until the final call to the `CheckpointIO` function relinquishes it, and, for logical I/O operations, that memory accesses do not page fault. Depending upon the I/O mode (programmed I/O or DMA), the I/O direction, and the data path coherence that you specify, the kernel manipulates the contents of the processor's caches, if any, and may make the underlying memory noncachable.

You must do I/O preparation before trying to transfer data. For operations with block oriented devices, the preparation might best be done just before moving the data, typically by the driver. For operations upon buffers such as memory shared between the main processor and a coprocessor, frame buffers, or buffers internal to a driver, the preparation might best be performed when the buffer is allocated.

You can specify values for the `options` field of the I/O preparation table that allow you to

- enable scatter gather memory operations
- specify whether address range structures contain logical addresses
- reduce memory usage when doing DMA transfers that require some logical I/O
- share mapping tables with the system, thus reducing memory requirements

- eliminating unnecessary cache manipulations
- automatically check access rights if I/O is done on behalf of a nonprivileged client

In the event that insufficient resources are available to prepare all of the specified memory, the `PrepareMemoryForIO` function does a partial preparation—that is, it prepares as much as possible and then returns a value indicating which memory has been prepared. You can examine the `kIOStateDone` bit in the `state` field of the I/O preparation table to check whether the preparation was complete or partial, and, in either case, you can examine the `firstPrepared` and `lengthPrepared` parameters to determine which part of the overall range or ranges has been prepared.

You must prepare and finalize memory for the benefit of the system and other users of the memory and backing store even if you don't need any of the information provided by `PrepareMemoryForIO`.

SPECIAL CONSIDERATIONS

The `PrepareMemoryForIO` function guarantees that the underlying physical memory remains assigned to the range or ranges at least until the `CheckpointIO` function relinquishes it. However, it does not guarantee that the original logical address range or ranges remain mapped. In particular, the controlling area or areas may be deleted before the `CheckpointIO` function completes. If the caller cannot somehow guarantee that the area or areas will continue to exist, logical address references to the underlying physical memory must be made through the static logical addresses provided in the mapping table or tables.

You need to match calls to the `PrepareMemoryForIO` function with calls to the `CheckpointIO` function, even if the I/O transfer is aborted.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function can be called only at task level from a driver's `DoDriverIO` routine or from a subroutine called by `DoDriverIO`.

SEE ALSO

To finalize preparing memory for I/O transfers, use the `CheckpointIO` function (page 4-69).

For details of the data structure that the `PrepareMemoryForIO` function uses, see the description of the I/O preparation table (page 4-21).

CheckpointIO

Finalizes the preparation of memory for use in I/O operations

```
OSStatus CheckpointIO (IOPreparationID thePreparationID,
                    IOCheckpointOptions theOptions);
```

`thePreparationID`	The I/O transaction ID returned by the `PrepareMemoryForIO` function. This ID is invalid after calling the `CheckpointIO` function without setting the `kMoreIOTransfers` option.
`theOptions`	For multiple I/O operations, this parameter defines the optional characteristics of subsequent transfers. Values for this parameter are defined by the I/O checkpoint options enumeration (page 4-20).
function results	To be provided.

DISCUSSION

The `CheckpointIO` function performs the necessary follow-up operations for the specified device I/O transfer, and optionally prepares for a new transfer or deallocates kernel resources used in preparing the range.

Multiple concurrent preparations of memory ranges or portions of memory ranges are supported.

SPECIAL CONSIDERATIONS

You must call the `CheckpointIO` function even if the I/O is aborted because the kernel resources need to be reclaimed.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To begin preparing memory for I/O transfers, use the `PrepareMemoryForIO` function (page 4-67).

Glossary

address space The domain of addresses that can be directly referenced by the processor at any given moment.

logical address A location within an address space. Logical addresses are unsigned; the lower bound of a logical address is zero; the upper bound is the size of the address space minus one.

physical address A location in physical memory: RAM, ROM, or cache.

nonpageable area An area which cannot be paged out.

sparse area An area in which memory is allocated as needed. If a resident area is sparse, this means that the physical memory is allocated by page faulting.

static logical address The logical address that can be used to access a physical address that you supplied while preparation for I/O is active. The structure of static logical mappings is guaranteed on a per-page basis; a static logical mapping endures from the time you call the `PrepareMemoryForIO` function through the final call to `CheckpointIO` function.

Server Manager

Contents

Server Manager Constants and Data Types

Server Main Entry Point

The Server Manager is responsible for launching, shutting down, and restarting servers. Also, it is responsible for facilitating communication between servers and their clients. The Server Manager identifies the available servers on the system by searching for them within a special subfolder of the System Folder.

The Server Manager can launch a server when the system starts up or delay launching it until the first time a client of the server wants to begin communicating with the server. When the Server Manager launches a server, it creates the main task of the server within a process and then the task begins executing at its main entry point.

In general, when a server begins executing at its main entry point, it performs some initialization and then receives communications from its clients. The entry point of a server must conform to the declaration of a server main entry point, which is a function that takes a server ID parameter (page 5-4) and returns nothing. The server ID, which is supplied by the Server Manager when it launches the server, is passed into the main entry point of the server when it begins executing.

When a client wants to begin communicating with a server, it calls either the `LookupServer` function (page 5-5) or the `LookupServerAsync` function (page 5-7) to obtain a way to communicate with the server. If the server is not ready to begin communicating with its clients, a caller of the `LookupServer` function may block until the server is ready (whereas a caller of the `LookupServerAsync` function, which operates asynchronously, receives notification when the server is ready). Therefore, as soon as a server has initialized itself and is ready to begin communicating with its clients, it should call the `ServerCreated` function (page 5-10) to inform the Server Manager of its readiness and to enable the Server Manager to unblock any callers of the `LookupServer` function or notify any callers of the `LookupServerAsync` function as quickly as possible. When it calls the `ServerCreated` function, the server must pass the same server ID that the Server Manager previously passed into the server's main entry point.

The `ServerMainEntry` data type defines a pointer to a server main entry point.

```
typedef void (*ServerMainEntry)(ServerID server);   /* entry point */
```

Server ID

The Server Manager uses a server ID to refer to a server. Although a client of a server refers to the server by name rather than by its server ID, a server itself must use this type when its calls the `ServerCreated` function (page 5-10) and passes its own server ID (which it obtains from the parameter passed to its server main entry point (page 5-3)) to inform the Server Manager that it is ready to communicate with its clients.

The `ServerID` data type defines a server ID.

```
typedef struct OpaqueServerID* ServerID;    /* server ID */
```

Server Manager Functions

Communicating With Servers

A client calls the `LookupServer` function (page 5-5) or the `LookupServerAsync` function (page 5-7) when it needs a way to communicate with a particular server. The communication information a client obtains by calling these functions supports some previously established communication protocol that is observed by the server and its clients.

LookupServer

Supplies a way to communicate with the specified server.

```
OSStatus LookupServer(
                ConstStr63Param serverName_t,
                Duration timeout_i,
                ServerID *server_o,
                void **refcon_o);
```

`serverName_t`	The name of the server (which is specified using the `ConstStr63Param` data type). You should supply the well known name of the server. The server specifies this name in its `'srvr'` resource (page 5-12). The `LookupServer` function returns result code `kernelIDErr` and supplies no value in the `refcon_o` parameter if the specified name does not correspond to a server that is available for communication with clients.
`timeout_i`	The maximum length of time to block if the server is not ready to communicate with its clients. You must be prepared for the possibility of blocking if the server is not ready to communicate with its clients (call the `LookupServerAsync` function (page 5-7) instead of `LookupServer` if you are not willing to block). If the server is ready to communicate with its clients when you call `LookupServer`, this parameter is ignored and `LookupServer` returns immediately, without blocking. The `Duration` data type is described in the chapter 'Timing Services Reference.'
`server_o`	A pointer to a server ID (page 5-4). On output, `LookupServer` supplies an ID for the server. Rather than using this ID, you usually refer to a server using a well known name that matches the name specified by the `'srvr'` resource of the server.
`refcon_o`	A pointer to a 32-bit word of unspecified data. On output, if the specified server is ready to communicate with its clients and the duration specified by the `timeout_i` parameter did not expire, then `LookupServer` supplies in this parameter information that indicates the way your program should communicate with the server. For example, the information might be the ID of a message object to which your program can send a microkernel message (described in the chapter 'Messaging Service Reference') or it might be the dispatcher ID to which your

program can send an AppleEvent (for information about AppleEvents, see the accompanying document called *Apple Events in Mac OS 8*). If the specified server is unavailable for communication with clients or the duration specified by the `timeout_i` parameter expires before the server becomes ready to communicate with its clients, nothing is returned on output in this parameter.

function result A result code. The result code `noErr` indicates that `LookupServer` successfully returned a value in the `refcon_o` parameter. The result code `kernelTimeoutErr` indicates that the duration specified by the `timeout_i` parameter expired. The result code `kernelIDErr` indicates that the specified server does not exist or it was unavailable for communication with clients. See "Server Manager Result Codes" (page 5-15) for a description of result codes that `LookupServer` may return.

DISCUSSION

A client should call `LookupServer` when it wants to begin communicating with a particular server. If the Server Manager has not launched the server already, calling `LookupServer` may result in the Server Manager blocking the caller and launching the server. If a client wants to begin communicating with a server, but it wants to avoid blocking if the server is not currently ready to communicate, it should call the `LookupServerAsync` function (page 5-7) instead of `LookupServer`.

The Server Manager maintains a list of servers that want to be available for communication with their clients. Typically, a server does want to be available for communication with its clients and indicates this by specifying it in its server resource (page 5-12). Any server that is not on the list cannot be launched, even if a client indicates that it wants to communicate with it. Consequently, the Server Manager supplies no value in the `refcon_o` parameter and `LookupServer` returns result code `kernelIDErr` if the client calls `LookupServer` for a server that is unavailable for communication with clients (the same behavior applies if the client specifies a nonexistent server name).

Soon after the Server Manager launches a server, the server informs the Server Manager that it is ready to begin communicating with its clients by calling the `ServerCreated` function (page 5-10) and passing some information that indicates how clients of the server should communicate with the server. This information supports some previously established communication protocol that is observed by this server and its clients. Once the Server Manager has obtained the server's communication information, the Server Manager can supply this same communication information to any blocked callers of `LookupServer` for this server and then unblock them. Clients subsequently calling `LookupServer` for the same server are not forced to block.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To obtain communication information for a server asynchronously, call the `LookupServerAsync` function (page 5-7). For information on how servers supply communication information, see the `ServerCreated` function (page 5-10).

LookupServerAsync

Asynchronously supplies a way to communicate with the specified server.

```
OSStatus LookupServerAsync(
                        ConstStr63Param serverName_t,
                        const KernelNotification *asyncNotification_i,
                        ServerID *server_o,
                        void **refcon_o);
```

`serverName_t` The name of the server (which is specified using the `ConstStr63Param` data type). You should supply the well known name of the server. The server specifies this name in its `'srvr'` resource (page 5-12). This asynchronous service completes without supplying a value in the `refcon_o` parameter if the specified name does not correspond to a server that is available for communication with clients.

`asyncNotification_i`
A pointer to a microkernel notification structure (described in the chapter 'Microkernel Notification Reference' to be provided at a later date) that specifies the mechanism by which the caller wants to be notified when this asynchronous service completes. The caller can use communication information supplied in the location it specified for the `refcon_o` parameter only if the notification it receives indicates that the result of this asynchronous service was `noErr`.

`server_o` A pointer to a server ID (page 5-4). On output, `LookupServerAsync` supplies an ID for the server. Rather than using this ID, you usually refer to a server using a well known name that matches the name specified by the `'srvr'` resource of the server.

`refcon_o` A pointer to a 32-bit word of unspecified data. When this asynchronous service completes (which is not when `LookupServerAsync` returns), if the specified server is ready to communicate with its clients, then information that indicates the way your program should communicate with the server is supplied in this location. For example, the information might be the ID of a message object to which your program can send a microkernel message (described in the chapter 'Messaging Service Reference') or it might be the dispatcher ID to which your program can send an AppleEvent (for information about AppleEvents, see the accompanying document called *Apple Events in Mac OS 8*). The caller should use the communication information supplied in this location only if the notification it receives indicates that this asynchronous service completed with the result code `noErr`. If the specified server is unavailable for communication with clients, no value is supplied in this parameter.

function result A result code. See "Server Manager Result Codes" (page 5-15) for a description of result codes that `LookupServerAsync` may return.

DISCUSSION

A client should call `LookupServerAsync` rather than the `LookupServer` function (page 5-5) when it wants to avoid blocking if the server is not currently ready to communicate. Callers of `LookupServerAsync` specify the mechanism by which they want to be notified when this asynchronous service completes. If the Server Manager has not launched the server already, calling `LookupServerAsync` may result in the Server Manager launching the server.

Whereas a client calling the `LookupServer` function determines whether the server is ready to begin communicating with it by inspecting the result returned by the function, a client calling `LookupServerAsync` determines whether the server is ready to begin communicating with it by inspecting the completion result in the notification it receives.

The Server Manager maintains a list of servers that want to be available for communication with their clients. Typically, a server does want to be available for communication with its clients and indicates this by specifying it in its server resource (page 5-12). Any server that is not on the list cannot be launched, even if a client indicates that it wants to communicate with it. Consequently, the Server Manager supplies no communication information to the client calling `LookupServerAsync` for a server that is unavailable for communication with clients (the same behavior applies if the client specifies a nonexistent server name).

Soon after the Server Manager launches a server, the server informs the Server Manager that it is ready to begin communicating with its clients by calling the `ServerCreated` function (page 5-10) and passing some information that indicates how clients of the server should communicate with the server. This information supports some previously established communication protocol that is observed by this server and its clients. Once the Server Manager has obtained the server's communication information, the Server Manager can supply this same communication information and deliver notification to any current or subsequent callers of `LookupServerAsync` for this server.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To obtain communication information for a server synchronously, call the `LookupServer` function (page 5-5). For information on how servers supply communication information, see the `ServerCreated` function(page 5-10).

Communicating With Clients

A server calls the `ServerCreated` function (page 5-10) once it is ready to begin communicating with its clients.

ServerCreated

Informs the Server Manager that a server is ready to communicate with its clients.

```
OSStatus ServerCreated(
                    ServerID server_t,
                    void *refcon_i);
```

`server_t`	A server ID (page 5-4). Specify the server ID passed into the server main entry point (page 5-3) when the server began executing. Specifying a nonexistent server ID causes `ServerCreated` to return the result code kernelIDErr.

`refcon_i`	A 32-bit word of unspecified data. When the server calls `ServerCreated`, it supplies in this parameter information that indicates the way that its clients should communicate with it. For example, it might supply the ID of a message object to which its clients can send a microkernel message (described in the chapter 'Messaging Service Reference') or it might supply the dispatcher ID to which its clients can send an AppleEvent (for information about AppleEvents, see the accompanying document called *Apple Events in Mac OS 8*). The Server Manager supplies this communication information to previous and subsequent callers of the `LookupServer` function (page 5-5) or the `LookupServerAsync` function (page 5-7) for this server. Supplying this communication information allows any clients who blocked calling the `LookupServer` function for this server to unblock and allows delivery of a completion notification to any clients who called the `LookupServerAsync` function for this server. Once the server calls `ServerCreated` and continues executing, clients calling the `LookupServer` function subsequently will obtain this communication information from the Server Manager immediately, without blocking.
function result	A result code. The result code `noErr` indicates that `ServerCreated` successfully passed its communication information to the Server Manager. The result code `kernelIDErr` indicates that the server specified the wrong server ID. See "Server Manager Result Codes" (page 5-15) for a description of the result codes that `ServerCreated` may return.

DISCUSSION

Once a server begins executing at its main entry point (page 5-3), it should call `ServerCreated` as soon as it is ready to begin communicating with its clients. Calling `ServerCreated` informs the Server Manager of the server's readiness and provides the Server Manager with information that indicates the way that clients of the server should communicate with the server (this information supports some previously established communication protocol that is observed by this server and its clients). Once the Server Manager obtains this communication information, it can unblock any callers of the `LookupServer` function and notify any callers of the `LookupServerAsync` function who are waiting to be supplied with this communication information. Furthermore, the Server Manager can supply this communication information immediately in response to any subsequent calls to the `LookupServer` function or the `LookupServerAsync` function by clients who want to communicate with this server.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For information on obtaining a way to communicate with a server, see the `LookupServer` function (page 5-5) and the `LookupServerAsync` function (page 5-7).

Server Manager Resources

The Server Resource

When you define your own server, you store it as a resource with the resource type `'srvr'`. When the Server Manager is looking for servers among the files within the server subfolder of the System Folder, it only recognizes a file as a server if the file has a server resource of type `'srvr'`.

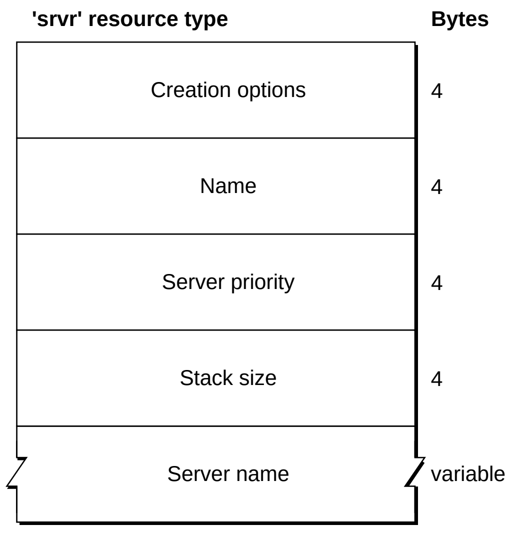
A resource of resource type `'srvr'` must have a resource ID of 0. The server resource type is defined by the `kCreateServerResType` constant and the server resource ID is defined by the `kCreateServerResID`.

```
enum {
    kCreateServerResType    = 'srvr',   /* server resource type */
    kCreateServerResID      = 0         /* server resource ID */
};
```

When the Server Manager identifies a server within the server subfolder of the System Folder, it opens the resource fork of the file and examines the information in the server resource. The information you specify in the server resource affects the Server Manager's treatment of the server. For example, you can specify the name by which clients of this server should refer to the server and you can specify whether the Server Manager should launch the server immediately or delay launching the server until the first time a client wants to communicate with it.

The `'srvr'` resource type defines a server resource. Figure 5-1 shows the format of a server resource.

Figure 5-1 Format of a server resource



Listing 5-1 shows the Rez template for a server resource.

Listing 5-1 The server resource

```
type 'srvr' {
unsigned longint    kServerIsPrivileged = 2,
                    kCreateServerForLookup = 4,
                    kCreateServerAtStartup = 8;         /* options */
unsigned longint    name;                               /* task name */
unsigned longint    kTaskLowServerPriority = 0x2006,
                    kTaskServerPriority = 0x2007,
                    kTaskHighServerPriority = 0x2008;   /* priority */
unsigned longint    stackSize;                          /* stack */
pstring;                                                /* server name */
};
```

You define a server resource by specifying these elements in a resource with the `'srvr'` resource type:

- The server creation options. You can specify any combination of the available constants.
    - Specifying `kServerIsPrivileged` causes the Server Manager to create the server within a process containing privileged tasks. In general, you should avoid specifying this option, which allows the Server Manager to create the server within a process containing nonprivileged tasks.
    - Specifying `kCreateServerForLookup` causes the Server Manager to add the server to a list of servers that want to be available for communication with their clients. Typically, you do specify this option so that the Server Manager can launch this server when necessary.
    - Specifying `kCreateServerAtStartup` causes the Server Manager to launch the server immediately when it identifies it in a special subfolder of the System Folder. Typically, you do not specify this option, although you do specify `kCreateServerForLookup` so that the Server Manager can delay launching the server until the first time a client wants to communicate with the server.
- The name of the main task of the server. This name is used only for debugging purposes. The name you specify is supplied when the main task of the server is created. See the chapter 'Tasks Reference' for more information on creating tasks.

- The priority of the main task of the server. You specify one of the available constants. See the chapter 'Tasks Reference' for more information on task priorities.
- The size of the stack you want the microkernel to create for the server. If you do not specify a size, the microkernel uses a default size.
- The name of the server. This should be the well known name by which clients should refer to this server.

Server Manager Result Codes

`noErr`	0	No error
`memFullErr`	–108	Not enough memory
`kernelIDErr`	–2419	Server does not exist
`kernelTimeoutErr`	–2415	Duration expired

Microkernel Queues Reference

Contents

Microkernel Queues Constants and Data Types

Microkernel Queue Options

You use microkernel queue options to specify how to create a microkernel queue. The `CreateKernelQueue` function (page 6-4) requires you to specify microkernel queue options. The `kNilOptions` enumerator is the only option available at this time.

The `KernelQueueOptions` data type defines the microkernel queue options.

```
typedef OptionBits KernelQueueOptions;  /* creation options */
```

Microkernel Queue ID

You use a microkernel queue ID to refer to a microkernel queue. You must use an ID to refer to a microkernel queue because you cannot refer directly to the underlying data structure of a microkernel queue.

When you create a microkernel queue using the `CreateKernelQueue` function (page 6-4), it generates a microkernel queue ID for the newly created microkernel queue. You must supply this ID when you wish to operate upon this microkernel queue using the other functions described in this chapter.

The `KernelQueueID` data type defines a microkernel queue ID.

```
typedef struct OpaqueKernelQueueID* KernelQueueID;  /* identifier */
```

Microkernel Queues Functions

Creating and Deleting Microkernel Queues

You can use the `CreateKernelQueue` function (page 6-4) to create a microkernel queue and the `DeleteKernelQueue` function (page 6-6) to delete one. Once you have created a microkernel queue, tasks can use it to communicate small amounts of data or to synchronize their operations.

CreateKernelQueue

Creates a microkernel queue.

```
OSStatus CreateKernelQueue (
                KernelQueueOptions options,
                KernelQueueID *theQueue);
```

`options`	The microkernel queue options that specify how to create the microkernel queue. You must specify `kNilOptions` (page 6-3) for this parameter.
`theQueue`	A pointer to a microkernel queue ID (page 6-3). On output, `CreateKernelQueue` supplies an ID for the newly created microkernel queue. You use this ID to refer to this microkernel queue in other microkernel queue functions.
function result	A result code. The result code `noErr` indicates that `CreateKernelQueue` successfully created a microkernel queue. See “Microkernel Queues Result Codes” (page 6-15) for a description of other result codes that `CreateKernelQueue` may return.

DISCUSSION

A microkernel queue is contained by the process in which it is created. Creating a microkernel queue generates an ID by which you can identify it. Because you cannot access the underlying data of a microkernel queue, you must refer to it by its ID in all functions that operate on existing microkernel queues.

Once a microkernel queue has been created, it can be notified and waited upon. When a task notifies a microkernel queue, it causes an entry containing three words of data to be placed in the microkernel queue. A task waiting for entries to arrive in the microkernel queue retrieves the data from the entry and removes the entry from the microkernel queue.

A microkernel queue exists until the process in which it was created terminates or until it is explicitly deleted by calling the `DeleteKernelQueue` function (page 6-6).

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For information on placing entries in a microkernel queue, see the `NotifyKernelQueue` function (page 6-11). For information on retrieving entries from a microkernel queue, see the `WaitOnKernelQueue` function (page 6-7). For information on notifying microkernel queues from secondary interrupt level, see the `AdjustKernelQueueSIHLimit` function (page 6-13). For information on freeing the resources associated with a microkernel queue that you are finished using, see the `DeleteKernelQueue` function (page 6-6).

DeleteKernelQueue

Deletes the specified microkernel queue.

```
OSStatus DeleteKernelQueue (KernelQueueID theQueue);
```

`theQueue`	A microkernel queue ID (page 6-3). You must specify the microkernel queue ID previously generated by the `CreateKernelQueue` function (page 6-4). If you specify an invalid microkernel queue ID, `DeleteKernelQueue` returns the result code `kernelIDErr`.
function result	A result code. The result code `noErr` indicates that `DeleteKernelQueue` successfully deleted the microkernel queue. See "Microkernel Queues Result Codes" (page 6-15) for a description of other result codes that `DeleteKernelQueue` may return.

DISCUSSION

When a microkernel queue is deleted, any tasks waiting on that microkernel queue become unblocked and return from the `WaitOnKernelQueue` function (page 6-7) with the result code `kernelIncompleteErr`.

SPECIAL CONSIDERATIONS

When a process (described in the chapter 'Process Manager Reference' to be provided at a later date) terminates, all microkernel queues created within that process are deleted automatically.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Working With Microkernel Queues

A task can use a microkernel queue to communicate small amounts of data or to synchronize its operations with another task. This is accomplished by having one task (or secondary interrupt handler) place an entry containing some data in a microkernel queue and having another task retrieve the data from the entry when the entry arrives in the microkernel queue. The tasks establish in advance which microkernel queue they will be using, what data each entry will contain, and how to interpret the data.

The task (or secondary interrupt handler) that places data in the microkernel queue uses the `NotifyKernelQueue` function (page 6-11). The task that retrieves data from the microkernel queue uses the `WaitOnKernelQueue` function (page 6-7).

WaitOnKernelQueue

Retrieves data from an entry in a microkernel queue or, if the microkernel queue is empty, optionally blocks to wait for the arrival of an entry in the microkernel queue.

```
OSStatus WaitOnKernelQueue (
                    KernelQueueID theQueue,
                    void **p1,
                    void **p2,
                    void **p3,
                    Duration timeOut);
```

`theQueue`	A microkernel queue ID (page 6-3). You must specify the microkernel queue ID previously generated by the `CreateKernelQueue` function (page 6-4). If you specify an invalid microkernel queue ID, `WaitOnKernelQueue` returns the result code `kernelIDErr`.
`p1`	A pointer to a 32-bit word of unspecified data or `null`. On output, if `WaitOnKernelQueue` successfully retrieved an entry from the microkernel queue, then `WaitOnKernelQueue` returns in this parameter the first of three 32-bit words of data from the entry. If `WaitOnKernelQueue` returns without retrieving an entry, nothing is returned on output in this parameter. Specify `null` if tasks using this microkernel queue have established a protocol that does not make use of the first word of each entry placed in the microkernel queue.
`p2`	A pointer to a 32-bit word of unspecified data or `null`. On output, if `WaitOnKernelQueue` successfully retrieved an entry from the microkernel queue, then `WaitOnKernelQueue` returns in this parameter the second of three 32-bit words of data from the entry. If `WaitOnKernelQueue` returns without retrieving an entry, nothing is returned on output in this parameter. Specify `null` if tasks using this microkernel queue have established a protocol that does not make use of the second word of each entry placed in the microkernel queue.
`p3`	A pointer to a 32-bit word of unspecified data or `null`. On output, if `WaitOnKernelQueue` successfully retrieved an entry from the microkernel queue, then `WaitOnKernelQueue` returns in this parameter the third of three 32-bit words of data from the entry. If `WaitOnKernelQueue` returns without retrieving an entry, nothing is returned on output in this parameter. Specify `null` if tasks using this microkernel queue have established a protocol that does not make use of the third word of each entry placed in the microkernel queue.
`timeOut`	The maximum length of time to block while waiting to retrieve data from the microkernel queue. Specifying `kDurationNoWait` ensures your task will not block if there are no entries in the microkernel queue. The `Duration` data type is described in the chapter 'Timing Services Reference.'

If you want your task to retrieve data from the microkernel queue only if it is possible to do so without blocking, specify `kDurationNoWait` for this parameter. If you specify `kDurationNoWait` for this parameter and the microkernel queue is empty, `WaitOnKernelQueue` will return immediately with result code `kernelTimeoutErr`, in which case you can either give up or get ready to allow your task to block before calling `WaitOnKernelQueue` again and specifying a value for this parameter that is not `kDurationNoWait`.

When you wish to repeatedly retrieve data as it arrives in the microkernel queue, call `WaitOnKernelQueue` in a loop and specify a value for this parameter that is not `kDurationNoWait` (this approach is preferable to polling).

function result

A result code. The result code `noErr` indicates that `WaitOnKernelQueue` successfully retrieved data from an entry in the microkernel queue. The result code `kernelTimeoutErr` indicates that the time specified by the `timeOut` parameter expired before `WaitOnKernelQueue` retrieved data from the microkernel queue. The result code `kernelIncompleteErr` indicates that the microkernel queue was deleted before `WaitOnKernelQueue` retrieved data from the microkernel queue. See "Microkernel Queues Result Codes" (page 6-15) for a description of other result codes that `WaitOnKernelQueue` may return.

DISCUSSION

When a task calls `WaitOnKernelQueue` for a microkernel queue that contains an entry, the task retrieves the data from the entry and returns immediately, without blocking. A task calling `WaitOnKernelQueue` also returns immediately, without blocking, if it specifies a `timeOut` value of `kDurationNoWait`, although in this case `WaitOnKernelQueue` returns the result code `kernelTimeoutErr` if it failed to retrieve data.

If the microkernel queue is empty and the task calling `WaitOnKernelQueue` did not specify a `timeOut` parameter of `kDurationNoWait`, the task will block and remain blocked until it retrieves an entry (or until its `timeOut` parameter expires). If multiple tasks are waiting on a microkernel queue when an entry arrives, the task which has been waiting the longest retrieves the data and returns from `WaitOnKernelQueue`, while any other tasks waiting on the microkernel queue continue to wait for the arrival of another entry (unless their `timeOut` parameters expire).

Each entry in a microkernel queue contains three 32-bit words of data. When `WaitOnKernelQueue` retrieves data from a microkernel queue containing multiple entries, it retrieves the three words of data from the entry which has been in the microkernel queue the longest. After it retrieves the data from the entry, `WaitOnKernelQueue` removes the entry from the microkernel queue and deallocates the entry's resources. Removing the entry ensures that another task cannot retrieve that data again. The other entries remain in the microkernel queue until other callers of `WaitOnKernelQueue` remove each one or the microkernel queue is deleted.

The `NotifyKernelQueue` function (page 6-11) places an entry in a specified microkernel queue. When the tasks using a microkernel queue have established a protocol that does not make use of all three words in an entry, a task calling the `NotifyKernelQueue` function specifies `null` on input for any of its `p1`, `p2`, and `p3` parameters that correspond to unused words in the entry. In this case, `WaitOnKernelQueue` returns `null` on output in its corresponding `p1`, `p2`, and `p3` parameters.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For information on placing an entry in a microkernel queue, see the `NotifyKernelQueue` function (page 6-11).

NotifyKernelQueue

Places an entry in a microkernel queue.

```
OSStatus NotifyKernelQueue (
                    KernelQueueID theQueue,
                    void *p1,
                    void *p2,
                    void *p3);
```

`theQueue`	A microkernel queue ID (page 6-3). You must specify the microkernel queue ID previously generated by the `CreateKernelQueue` function (page 6-4). If you specify an invalid microkernel queue ID, `NotifyKernelQueue` returns the result code `kernelIDErr`.
`p1`	A 32-bit word of unspecified data or `null`. When `NotifyKernelQueue` places an entry in the microkernel queue, it copies this data into the first of three 32-bit words in the entry. Specify `null` if tasks using this microkernel queue have established a protocol that does not make use of the first word of each entry placed in the microkernel queue.
`p2`	A 32-bit word of unspecified data or `null`. When `NotifyKernelQueue` places an entry in the microkernel queue, it copies this data into the second of three 32-bit words in the entry. Specify `null` if tasks using this microkernel queue have established a protocol that does not make use of the second word of each entry placed in the microkernel queue.
`p3`	A 32-bit word of unspecified data or `null`. When `NotifyKernelQueue` places an entry in the microkernel queue, it copies this data into the third of three 32-bit words in the entry. Specify `null` if tasks using this microkernel queue have established a protocol that does not make use of the third word of each entry placed in the microkernel queue.

function result

A result code. The result code `noErr` indicates that `NotifyKernelQueue` successfully placed the entry in the microkernel queue. A secondary interrupt handler may receive the result code `memFullErr` if you neglected to preallocate sufficient resources using the `AdjustKernelQueueSIHLimit` function (page 6-13). See “Microkernel Queues Result Codes” (page 6-15) for a description of other result codes that `NotifyKernelQueue` may return.

DISCUSSION

`NotifyKernelQueue` returns immediately after placing an entry in the microkernel queue. If no tasks are waiting on the microkernel queue, the entry will remain in the microkernel queue until some task calls the `WaitOnKernelQueue` function (page 6-7) to retrieve it or the microkernel queue is deleted. If one or more tasks are waiting on the microkernel queue, `NotifyKernelQueue` causes the task which has been waiting the longest to retrieve the data from the entry, removes the entry from the microkernel queue, deallocates the entry’s resources, and unblocks the task.

You can use `NotifyKernelQueue` to pass any three words of data to another task that will call `WaitOnKernelQueue` on the same microkernel queue.

The asynchronous services provided by Mac OS 8 allow you to request notification of their completion. You request notification and specify the mechanism by which you want to be notified by supplying a microkernel notification structure (described in the chapter ‘Microkernel Notification Reference’ to be provided at a later date) when you call the service. Notification of a particular microkernel queue is a mechanism you can request by supplying the microkernel queue ID in a parameter of the microkernel notification structure. Then, upon completion of the asynchronous operation, the microkernel will call `NotifyKernelQueue` for the microkernel queue you specified and pass the completion status returned by the asynchronous operation in the `p3` parameter `(`the `p1` and `p2` parameters of the microkernel notification structure are passed through to the corresponding `p1` and `p2` parameters of `NotifyKernelQueue).`

▲ **WARNING**

Placing an entry in a microkernel queue consumes resources. Therefore, if you intend to call `NotifyKernelQueue` at secondary interrupt level, you must call `AdjustKernelQueueSIHLimit` (page 6-13) at task level beforehand to preallocate resources. ▲

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers.

SEE ALSO

For information on retrieving the data from an entry placed in a microkernel queue, see the `WaitOnKernelQueue` function (page 6-7).

Microkernel Queues And Secondary Interrupt Handlers

A secondary interrupt handler may wish to notify a microkernel queue. Before this can occur, a task must preallocate the microkernel queue resources that the `NotifyKernelQueue` function (page 6-11) will consume when the secondary interrupt handler calls it. The `AdjustKernelQueueSIHLimit` function (page 6-13) is provided for this purpose.

AdjustKernelQueueSIHLimit

Preallocates (or deallocates) microkernel queue resources to allow secondary interrupt handlers to notify microkernel queues.

```
OSStatus AdjustKernelQueueSIHLimit (
                    KernelQueueID theQueue,
                    SInt32 amount,
                    ItemCount *newLimit);
```

`theQueue`	A microkernel queue ID (page 6-3). You must specify the microkernel queue ID previously generated by the `CreateKernelQueue` function (page 6-4). If you specify an invalid microkernel queue ID, `AdjustKernelQueueSIHLimit` returns the result code `kernelIDErr`.
`amount`	The amount by which to either increment or decrement a pool of resources available to secondary interrupt handlers using the microkernel queue. To sufficiently increment the limit, pass the maximum number of entries that can be in the microkernel queue simultaneously as a result of the secondary interrupt handler calling the `NotifyKernelQueue` function(page 6-11). To decrement the limit after the resources are no longer needed, pass the negative of the amount by which you previously incremented the limit.
`newLimit`	A pointer to an item count or `null`. On output, `AdjustKernelQueueSIHLimit` supplies the new limit for the microkernel queue. Specify `null` if you have no interest in this information.
function result	A result code. The result code `noErr` indicates that `AdjustKernelQueueSIHLimit` successfully allocated or deallocated the resources. See "Microkernel Queues Result Codes" (page 6-15) for a description of other result codes that `AdjustKernelQueueSIHLimit` may return.

DISCUSSION

The microkernel cannot allocate microkernel queue resources at secondary interrupt level. Therefore, if a secondary interrupt handler is going to call the `NotifyKernelQueue` function, you must call `AdjustKernelQueueSIHLimit` at task level to preallocate a sufficient quantity of microkernel queue resources. If you neglect to preallocate a sufficient quantity of resources, you may cause your secondary interrupt handler or another secondary interrupt handler to return from the `NotifyKernelQueue` function with the result code `memFullErr`.

Only drivers, which are the only type of software that executes at interrupt level, need to call `AdjustKernelQueueSIHLimit` if they notify microkernel queues at secondary interrupt level. Each driver that needs to call `AdjustKernelQueueSIHLimit` should call it once during initialization and call it again when the driver is removed (calling it prior to each notification of the microkernel queue would increase a driver′s execution time unnecessarily). The first call should increment the quantity of resources and the last call should decrement the quantity by the same amount.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Microkernel Queues Result Codes

`noErr`	0	No error
`memFullErr`	–108	Not enough memory
`kernelIDErr`	–2419	Microkernel queue ID does not exist
`kernelTimeoutErr`	–2415	Wait duration expired
`kernelIncompleteErr`	–2401	Microkernel queue deleted
`kernelOptionsErr`	–2403	Invalid microkernel queue options

Interrupt Services Reference

Contents

Interrupt Services Reference

This chapter describes the functions you use to control software interrupts and secondary interrupts, and discusses the function prototypes for software interrupt handlers, secondary interrupt handlers, and hardware interrupt handlers.

Functions

This section describes the functions you use to create and send software interrupts, to determine whether a software interrupt is executing, to call a secondary interrupt handler, and to ensure that the system has sufficient resources to run your secondary interrupt handler.

Controlling Software Interrupts

You can enable and disable software interrupts for a given task, using the `EnableSoftwareInterrupts` function and the `DisableSoftwareInterrupts` function.

DisableSoftwareInterrupts

Disables software interrupts

```
void DisableSoftwareInterrupts (void)
```

DISCUSSION

By default, software interrupts are enabled. The `DisableSoftwareInterrupts` function disables all pending software interrupts for the calling task. You cannot use this function to disable an interrupt that is currently executing.

Calls to the `EnableSoftwareInterrupts` and the `DisableSoftwareInterrupts` functions nest automatically, so that you must match every call to the `DisableSoftwareInterrupts` function with a call to the `EnableSoftwareInterrupts` function.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

Use the `EnableSoftwareInterrupts` function (page 7-4) to enable software interrupts.

The software interrupt handler is described on page 7-11.

EnableSoftwareInterrupts

Enables software interrupts.

```
void EnableSoftwareInterrupts (void);
```

DISCUSSION

The `EnableSoftwareInterrupts` function enables all pending software interrupts for the calling task. Because software interrupts are enabled by default, you need to call this function only if you have disabled software interrupts using the `DisableSoftwareInterrupts` function.

Calls to the `EnableSoftwareInterrupts` and the `DisableSoftwareInterrupts` functions nest automatically, so that you must match every call to the `DisableSoftwareInterrupts` function with a call to the `EnableSoftwareInterrupts` function.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

Use the `DisableSoftwareInterrupts` function (page 7-3) to disable software interrupts.

The software interrupt handler is described on page 7-11.

Creating a Software Interrupt

You use the `CreateSoftwareInterrupt` function to create a software interrupt. Creating a software interrupt assigns it a software interrupt ID, specifies whether the ID is temporary or persistent, and defines additional information that is associated with the interrupt request.

CreateSoftwareInterrupt

Returns a software interrupt ID that specifies an instance of a software interrupt request.

```
OSStatus CreateSoftwareInterrupt (SoftwareInterruptHandler handler,
                     TaskID task,
                     void * p1,
                     Boolean persistent,
                     SoftwareInterruptID * theSoftwareInterrrupt);
```

`handler` The address of the routine that is to execute when the software interrupt specified by the parameter `theSoftwareInterrupt` is activated. This address must be in the same address space as the task to which you are sending it.

`task` The ID of the task to which you are going to send this software interrupt. The task ID must identify a task that is in the same process as the task sending the interrupt. Specifying `NULL` causes the software interrupt to be sent to the task that created the interrupt.

`p1` A user-defined 32-bit value. The microkernel passes this value to the software interrupt handler routine specified by the `handler` parameter.

`persistent` A boolean specifying whether the software interrupt is persistent or temporary.

A value of `true` means that the ID of the software interrupt is valid until you delete the interrupt with the `DeleteSoftwareInterrupt` function. You can send a persistent interrupt more than once, but it must run before it can be scheduled again.

A value of `false` means that the ID of the software interrupt is valid until the microkernel starts to execute the interrupt. This is some time after you send it.

`theSoftwareInterrupt`
On return, a pointer to a software interrupt ID.

function result

If the ID specified for the `task` parameter is invalid, the function returns the result code `kernelIDErr`.

If the value specified for the parameter `theSoftwareInterrupt` is not `nil` and is invalid, the function returns the result. `paramErr`.

If the microkernel does not have enough memory to create the software interrupt, it returns the result `memfullErr`.

DISCUSSION

You use the `CreateSoftwareInterrupt` function to have the microkernel create a software interrupt ID and to associate that ID with the following information: the address of a software interrupt handler that executes when the microkernel activates this software interrupt, the ID of the task to which you send the interrupt, the duration of the software interrupt ID, and a user-defined value that the microkernel passes to the interrupt handler when it calls it.

The sender and the recipient of a software interrupt must be in the same process.

After you have created a software interrupt, you can cause its handler to execute by calling the `SendSoftwareInterrupt` function. The handler may or may not execute immediately, depending on the task's priority. Note that sending a software interrupt to a task does not change that task's priority.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To send a software interrupt, you use the `SendSoftwareInterrupt` function (page 7-8).

To delete a software interrupt, you use the `DeleteSoftwareInterrupt` function (page 7-11).

The software interrupt handler is described on page 7-11.

Sending Software Interrupts

After you create a software interrupt, you can send it to a specific task by calling the `SendSoftwareInterrupt` function.

SendSoftwareInterrupt

Sends a software interrupt to a task.

```
OSStatus SendSoftwareInterrupt (SoftwareInterruptID theSoftwareInterrupt,
                      void * p2);
```

`theSoftwareInterrupt`
The ID of the software interrupt you want to send. This value is returned by the `CreateSoftwareInterrupt` function in the parameter `theSoftwareInterrupt`.

`p2` A user-defined 32-bit value. The microkernel passes this value to the software interrupt handler routine when it activates it.

function result
The function returns the result code `noErr` unless one of the following occurs:

If the ID specified for the parameter `theSoftwareInterrupt` is invalid, the function returns the result code `kernelIDErr`.

If a software interrupt is already in the process of being sent, the function returns the result `KernelInUseErr`.

If the target task is in the process of terminating, the function returns the result `KernelTerminatedErr`.

DISCUSSION

You use the `SendSoftwareInterrupt` function to send a software interrupt created with the `CreateSoftwareInterrupt` function. The interrupt specified by the parameter `theSoftwareInterrupt` is sent to the task specified by the `task` parameter. The sender of the software interrupt can be in a different process than the creator or the target of the interrupt.

The microkernel queues software interrupts sent to a task and activates a software interrupt when all of the following conditions are met: the task to which it is sent becomes eligible for execution, software interrupts are enabled, and all software interrupts previously sent to the task have been processed. Note that a task is eligible for execution even if it is currently blocked. Thus a software interrupt sent to a blocked task can execute even while the task is blocked.

The system deletes the ID of a temporary software interrupt after it has been activated. If you send a temporary interrupt more than once, the `SendSoftwareInterrupt` function returns an error. You can send a persistent interrupt more than once; however, it can be queued for execution only once. If you send it more than once before it is activated, it only executes once. If you send it again after it has run, it is queued again.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

SEE ALSO

Use the `CreateSoftwareInterrupt` function (page 7-6) to obtain a software interrupt ID, to specify the software interrupt handler that is to run, and to specify the task to which the software interrupt is sent.

Use the `InSoftwareInterruptHandler` function (page 7-10) to determine whether a task is executing at software interrupt level.

The software interrupt handler is described on page 7-11.

Querying the Level of Execution

You call the `InSoftwareInterruptHandler` function to determine whether the current task is executing at software interrupt level.

InSoftwareInterruptHandler

Determines whether the current task is running at software interrupt level.

```
Boolean InSoftwareInterruptHandler (void);
```

function result
The function returns `true` if the current task is running at software interrupt level; otherwise, it returns `false`.

DISCUSSION

The `InSoftwareInterruptHandler` function is useful if you have code that can run both at task level and at software interrupt level. You can use this function with such code to trigger different behavior depending on whether the code was running at task level or not.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To delete a software interrupt, use the `DeleteSoftwareInterrupt` function (page 7-11).

To disable software interrupts, use the `DisableSoftwareInterrupts` function (page 7-3).

The software interrupt handler is described on page 7-11.

Deleting a Software Interrupt

You can delete a software interrupt by calling the `DeleteSoftwareInterrupt` function.

DeleteSoftwareInterrupt

Deletes the specified software interrupt.

```
OSStatus DeleteSoftwareInterrupt (SoftwareInterruptID
                    theSoftwareInterrupt);
```

`theSoftwareInterrupt`
The ID of a software interrupt that has been created using the `CreateSoftwareInterrupt` function.

function result
The function returns the result code `noErr` unless the ID specified for the parameter `theSoftwareInterrupt` is invalid, in which case it returns the result code `kernelIDErr`.

DISCUSSION

You can use the `DeleteSoftwareInterrupt` function to delete a software interrupt that has been created or that has been sent. If the software interrupt is queued and you delete it, it will not be activated.

You cannot use this function to abort a software interrupt handler that is currently executing.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

Use the `DisableSoftwareInterrupts` function to disable software interrupts (page 7-3).

The software interrupt handler is described on page 7-11.

Calling a Secondary Interrupt Handler

You can invoke a secondary interrupt handler in one of two ways. To call a secondary interrupt handler asynchronously, use the `QueueSecondaryInterruptHandler` function. Normally you would make this type of call from hardware interrupt level. To call a secondary interrupt handler synchronously, use the `CallSecondaryInterruptHandler2` function. You can make this type of call from task level or secondary interrupt level.

QueueSecondaryInterruptHandler

Queues a secondary interrupt handler for execution.

```
OSStatus QueueSecondaryInterruptHandler
                    (SecondaryInterruptHandler2 theHandler,
                    ExceptionHandler theExceptionHandler,
                    void * p1,
                    void * p2);
```

`theHandler` A pointer to a secondary interrupt handling routine that is to be queued for execution.

`theExceptionHandler`
A pointer to an exception handler to which the system transfers control should an exception arise. You can specify `NIL` if you do not plan to provide an exception handler. If you do not provide a handler, an exception will result in a system crash.

`p1` A 32-bit user-defined value that the microkernel passes to the handler when it executes.

`p2` A 32-bit user-defined value that the microkernel passes to the handler when it executes.

function result
If there is not enough memory available to queue the handler, the function returns the result `memfullErr`. If the address referenced by the parameter `theHandler` or the parameter `theExceptionHandler` is invalid, the function returns the result code `paramErr`.

DISCUSSION

You can queue secondary interrupt handlers during the processing of a hardware interrupt or from secondary interrupt level. In the first case, the secondary interrupt handler executes just before execution is transferred to task level. In the second case, the secondary interrupt handler executes after all other secondary handlers that have been queued before it have executed.

If you queue a secondary interrupt handler from hardware interrupt level, you must first use the `AdjustSecondaryInterruptHandlerLimit` function to inform the microkernel that it will need to allocate additional resources to handle the queued handler. I/O plug-ins would normally call this function during their initialization sequence.

If you do not provide a pointer to an exception handler and an exception occurs, a system-fatal error is raised and the user is alerted.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

This function can only be called by privileged software.

SEE ALSO

You use the `AdjustSecondaryInterruptHandlerLimit` function (page 7-16) to increase or decrease the secondary interrupt limit that is used by the microkernel to allocate resources.

If you are not calling a secondary interrupt handler from hardware interrupt level, you can also use the function `CallSecondaryInterruptHandler2` (page 7-14) to execute the handler.

CallSecondaryInterruptHandler2

Calls a secondary interrupt handler from task level or secondary interrupt level.

```
OSStatus CallSecondaryInterruptHandler2
                    (SecondaryInterruptHandler2 theHandler,
                    ExceptionHandler theExceptionHandler,
                    void * p1,
                    void * p2);
```

`theHandler` A pointer to a secondary interrupt handling routine that is to be queued for execution.

`theExceptionHandler`
A pointer to an exception handler to which the system transfers control should an exception arise. You can specify `NIL` if you do not plan to provide an exception handler.

`p1`	A 32-bit user-defined value that the microkernel passes to the handler when it executes.
`p2`	A 32-bit user-defined value that the microkernel passes to the handler when it executes.
function result	The function returns the result code `noErr` unless the address referenced by the parameter `theHandler` or the parameter `theExceptionHandler` is invalid, in which case it returns the result code `paramErr`.

DISCUSSION

The secondary interrupt handler that you invoke using the function `CallSecondaryInterruptHandler2` is executed immediately; it is never queued.

If you do not provide a pointer to an exception handler and an exception occurs, the system will crash.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	No

CALLING RESTRICTIONS

This function can only be called by privileged software. It cannot be called at hardware interrupt level.

SEE ALSO

If you need to call a secondary interrupt handler from hardware interrupt level, you must use the `QueueSecondaryInterruptHandler2` function (page 7-12).

Adjusting the Limit of Secondary Interrupt Handlers

You use the `AdjustSecondaryInterruptHandlerLimit` function to inform the microkernel that you plan to queue a secondary interrupt handler. This function sets aside microkernel memory for secondary interrupt handlers. I/O plug-ins should call this function during initialization to reserve memory for the maximum number of concurrently queued secondary interrupt handlers.

AdjustSecondaryInterruptHandlerLimit

Increases or decreases the limit known to the microkernel for queued secondary interrupt handlers and returns the new value.

```
OSStatus AdjustSecondaryInterruptHandlerLimit (long amount, unsigned
                        long * newLimit);
```

`amount`	The number of secondary interrupt handlers that you are adding to or subtracting from the queue of secondary interrupt handlers. Specify a positive number to increase the limit; specify a negative number to decrease the limit.
`newLimit`	A pointer. On return, it points to the number of secondary interrupt handlers currently available for execution.

DISCUSSION]

Secondary interrupt handlers that are queued from hardware interrupt handlers consume microkernel resources from the time they are queued until the time they begin to execute. Because the microkernel can’t allocate the needed resources dynamically at interrupt level, you must call the `AdjustSecondaryInterruptHandlerLimit` function at task level to ensure that there are sufficient resources. To allocate the necessary resources, figure out the number of interrupt handlers that you plan to have queued at the same time and call the `AdjustSecondaryInterruptHandlerLimit` function during your software’s initialization to increase the limit by that amount.

You must call the `AdjustSecondaryInterruptHandlerLimit` function if you are using or have used the function `QueueSecondaryInterruptHandler2`. For each simultaneously queued secondary interrupt handler you call, you should increment the queued limit by one. For example, if you queue a secondary interrupt handler from your hardware interrupt handler and don’t queue any additional secondary interrupt handlers until the first has finished executing, you only need to increment the queued limit by one. Before the calling software terminates or is removed, it should decrement the queued secondary interrupt handler limit by as many elements as it has added.

If you fail to increment the queued software interrupt handler limit, your software might run without any problems. However another process' attempt to queue a secondary interrupt handler might fail with a `memFullErr` result. Because the limit manipulated by the `AdjustSecondaryInterruptHandlerLimit` function is shared by all callers of `QueueSecondaryInterrruptHandler`, the function that receives a bad result from the call might not necessarily be the software that failed to adjust the queued limit.

SEE ALSO

You use the `QueueSecondaryInterruptHandler2` function (page 7-12) to queue a secondary interrupt handler for execution.

User-Defined Functions

This section describes the prototypes for and use of software interrupt handlers, secondary interrupt handlers, and hardware interrupt handlers.

Software Interrupt Handlers

A software interrupt handler defines a function that the microkernel can run at software interrupt time, within the context of a given task but asynchronously to that task. A software interrupt can be sent to a task by another task, by a secondary interrupt handler, or by the task itself.

When you create a software interrupt, you specify the address of a software interrupt handler that the microkernel executes whenever the software interrupt is activated. You write the software interrupt handling routine; the declaration of the routine must conform to the following prototype

```
typedef void (*SoftwareInterruptHandler) ( void * p1, void *p2);
```

`p1`	A 32-bit user-defined parameter that is passed to the interrupt handler when the microkernel calls it. The value of `p1` is defined by the creator of the software interrupt using the `p1` parameter to the `CreateSoftwareInterrupt` function.

`p2`	A 32-bit user-defined parameter that is passed to the interrupt handler when the system calls it. The value of `p2` is defined by the sender of the software interrupt using the `p2` parameter to the `SendSoftwareInterrupt` function.

DISCUSSION

A software interrupt handler is a function that is invoked by the microkernel when a software interrupt is sent to a task. The microkernel saves the interrupted task's state, executes the specified handler, and restores the task's state when the handler has finished executing. The software interrupt shares the same context with the task to which it is sent; the software interrupt handler is executed on that task's stack and has the same addressing context as that of the task. The software interrupt is asynchronous to the task to which it is sent—that is, it interrupts the execution of that task as if an invisible call to the handler were inserted into the task's execution. Consequently, the software interrupt handler can execute even when that task is blocked. When a software interrupt handler completes, the task to which the interrupt is sent, resumes execution at the point where it was interrupted. If the task was blocked when it was interrupted, it remains blocked after the interrupt processing is complete, waiting for the condition that has caused it to block to clear.

Software interrupt handlers can operate on a task's private data area because the task does not execute while the interrupt is executing.

Software interrupts are executed sequentially. If a task executing a software interrupt function is sent another software interrupt, it finishes processing the first interrupt before processing the second interrupts. This is true even if the first software interrupt handler performs some blocking operation.

The presence of a pending software interrupt or the invocation of a software interrupt handler does not affect the way in which the targeted task is scheduled except that it might run the handler even in a task which is blocked.

SEE ALSO

You use the `CreateSoftwareInterrupt` function (page 7-6) to obtain a software interrupt ID for a software interrupt. Once you have obtained the ID, you can send the software interrupt to a specific task by calling the `SendSoftwareInterrupt` function (page 7-8).

For information about the use of software interrupts for synchronization, see the chapter “Software Interrupts.”

Secondary Interrupt Handler

A secondary interrupt handler is a user-defined function that the microkernel executes at secondary interrupt time. You can queue a secondary interrupt handler from hardware interrupt level to complete processing that you cannot do at that level because of constraints on available time or on the functions you can call. Or, you can call a secondary interrupt handler from task level or secondary interrupt level when you need to synchronize with other secondary interrupt level code. Depending on the function you use to invoke the secondary interrupt handler, the handler either executes immediately or is queued for execution.

No matter which function you use to execute the handler, the caller of a secondary interrupt handler, specifies the address of the handler, the address of an exception handling routine, and two additional user-defined parameters that are passed to the routine when it executes.

The declaration of a secondary interrupt handler must conform to the following prototype definition:

```
typedef OSStatus (*SecondaryInterruptHandler2) ( void * p1, void *p2);
```

`p1`	A 32-bit user-defined parameter that is passed to the secondary interrupt handler when the handler executes. The value of `p1` can be passed either by the `QueueSecondaryInterruptHandler` function or by the `CallSecondaryInterruptHandler` function.
p2	A 32-bit user-defined parameter that is passed to the secondary interrupt handler when the handler executes. The value of `p2` can be passed either by the `QueueSecondaryInterruptHandler` function or by the `CallSecondaryInterruptHandler` function.

DISCUSSION

Secondary interrupt handlers must conform to the interrupt execution environment rules. They may not cause page faults and they are restricted to using a limited set of system services.

Secondary interrupt handlers execute on a special interrupt stack and should not make any assumptions about the task context in which they execute.

Secondary interrupt handlers can be called by hardware interrupts, by other secondary interrupt handlers, or by privileged tasks. If a secondary interrupt handler is queued from a hardware interrupt handler, it executes after the latter has finished executing and right before the system goes back to task-level execution. If a secondary interrupt handler is called from a secondary interrupt or privileged task the timing of execution is determine by the function used to invoke it. If you use the `QueueSecondaryInterruptHandler` function, it executes after all other handlers queued before it have finished executing. It you use the `CallSecondaryInterruptHandler2` function, it executes immediately.

Because secondary interrupt handlers are guaranteed to be executed serially, they are the primary synchronization mechanism used within the microkernel and its extensions. You can also use them for this purpose. For example, if you have a data structure that requires complex update operations and each of the operations uses secondary interrupt handlers to access or update the data structure, all access to the data structure is synchronized even though hardware interrupts are enabled during the access.

You must take care not to rely on the use of secondary interrupt handling as a means of achieving synchronization if this results in degraded system performance. Although hardware interrupts are still processed during the execution of secondary interrupts, task-level execution is entirely halted, which can result in severely degraded system response. For this reason, you should use secondary interrupt handlers only when absolutely necessary or when the operations performed at secondary interrupt level are very brief. It is always preferable to execute entirely at task level and use appropriate mechanisms to obtain synchronization.

SPECIAL CONSIDERATIONS

When a secondary interrupt handler executes on a multiprocessor system, task level execution continues on other processors. While the microkernel guarantees that all secondary interrupt handlers are serialized, it does not guarantee that no tasks run while a secondary interrupt handler runs. As a result, any code that depends upon this behavior might not work correctly on multiprocessor system.

SEE ALSO

You can use the `QueueSecondaryInterruptHandler` function or the `CallSecondaryInterruptHandler` function to call a secondary interrupt handler.

For more information on secondary interrupts, see the chapter "Secondary Interrupts" and additional related material in *Modular I/O*.

Hardware Interrupt Handlers

A hardware interrupt handler or primary interrupt handler is a user-defined function that is invoked by the microkernel in response to a signal from an external device.

The declaration of a hardware interrupt handler must conform to the following prototype definition:

```
typedef void (*InterruptVectorHandler) ( InterruptVector theVector,
                     void *parameter);
```

`theVector`	A 32-bit value specifying a vector number that identifies the source of the interrupt. This is the same value that is specified when the hardware interrupt handler is installed. It allows a single interrupt handler installed for multiple sources to determine the source of the current invocation.
`parameter`	A pointer to a value that is passed to the function used to install the interrupt handler in the parameter `theParameter`.

DISCUSSION

A hardware interrupt causes all software execution (including the execution of software and secondary interrupts) on the interrupted processor to stop until the handler executes. Consequently, for best system performance, you should write a hardware interrupt handler so that it performs only those actions that must be synchronized with the external device that caused the interrupt. If you need to do additional work, you should queue a secondary interrupt handler to perform the remaining work. Of course, any hardware interrupt handler must act to remove the cause of the interrupt before it returns.

Interrupt handlers execute on a special stack dedicated to interrupt processing. Hardware interrupt handlers are invoked with the addressing context that was current when they were installed. However, all data and code references generated during the processing of a hardware interrupt must be to pages that are physically resident. Attempting to access non-resident pages causes access error exceptions.

Microkernel services pertaining to hardware interrupts are available only to privileged clients. The microkernel services that can be called from hardware interrupt level are limited to the following: `SetEvents`, `ClearEvents`, `SendSoftwareInterrupt`, `QueueSecondaryInterruptHandler`, and `SetInterruptTimer`. The primary client for these services is the I/O system, which layers a hierarchical interrupt dispatcher upon the microkernel's services. I/O plug-ins should use this I/O system interrupt management layer to install a hardware interrupt handler.

SPECIAL CONSIDERATIONS

In a multiprocessor system, only one hardware interrupt handler can execute at one time.

SEE ALSO

Secondary interrupt handlers are described on (page 7-19). You use the `QueueSecondaryInterruptHandler2` function (page 7-12) to queue a secondary interrupt handler for execution from a hardware interrupt handler.

You can find more information about the I/O system interrupt management layer in *Modular I/O*.

Exception Handling

Contents

Exception Handling Reference

This chapter describes the prototype for an exception handler and the function you use to install the handler. The microkernel invokes this exception handler whenever an exception occurs in the context of the code from which you installed the handler. Exception handlers are used in Mac OS 8 to handle exceptions that occur in every type of code: hardware interrupts, secondary interrupts, debuggers, tasks, and software interrupts.

When the microkernel invokes your exception handler, it passes information to it about the exception that has occurred and about the state of the cpu when the exception was raised. This chapter also describes the data structures used to specify this information.

Constants and Data Types

When an exception occurs, the microkernel returns information that describes the state of the machine and the cause of the exception to the installed exception handler or debugger.

This section describes the constants and data types used to specify exception information.

Exception Kind Enumeration

When an exception occurs, the microkernel passes an exception information structure (page 8-6) to an exception handler or debugger. The field `theKind` of this structure contains a constant, defined by the exception kind enumeration, that specifies the kind of exception that has occurred.

```
enum {
    kUnknownException               = 0,
    kIllegalInstructionException    = 1,
    kTrapException                  = 2,
    kAccessException                = 3,
    kUnmappedMemoryException        = 4,
    kExcludedMemoryException        = 5,
    kReadOnlyMemoryException        = 6,
    kUnresolvablePageFaultException = 7,
    kPrivilegeViolationException    = 8,
    kTraceException                 = 9,
    kInstructionBreakpointException = 10,
    kDataBreakpointException        = 11,
    kIntegerException               = 12,
    kFloatingPointException         = 13,
    kStackOverflowException         = 14,
    kTaskTerminationException       = 15,
    kTaskCreationException          = 16
};
```

`kUnknownException`	Unknown kind of exception. This exception code is defined for completeness only; it is never actually passed to an exception handler.
`kIllegalInstructionException`	Illegal instruction exception. The cpu attempted to decode an instruction that is either illegal or unimplemented.
`kTrapException`	Unknown trap type exception. The cpu decoded a trap type instruction that is not used by the system software but which might be used by a debugger.
`kAccessException`	Memory access exception. A memory reference failed because the physical address is not accessible either because the hardware is bad or non-existent.
`kUnmappedMemoryException`	Unmapped memory exception. A memory reference was made to an address that is unmapped. This might happen because no area corresponds to the address, because the address is in an area's guard range, or because the address is in an unmapped page of a sparse area. Sparse area pages must be explicitly loaded by the `ControlPagingForRange` function.
`kExcludedMemoryException`	Excluded memory exception. A memory reference was made to an address in an area whose memory access level is `kMemoryExcluded` for the mode (user or privileged) in which an access was made.

`kReadOnlyMemoryException`

Read-only memory exception. A memory reference was made to an address in an area whose memory access level is `kMemoryReadOnly` for the mode (user or privileged) in which a write access was made.

`kUnresolvablePageFaultException`

Unresolvable page fault exception. A memory reference resulted in a page fault that could not be resolved. The field `theError` of the memory exception structure contains a status value indicating the reason for this unresolved page fault. In general, either the error occurred at secondary interrupt level or at hardware interrupt level (`theError` == `kernelExecutionLevelErr`) or the fault could not be processed because there were insufficient resources.

Disabling hardware interrupts and then attempting to reference data that's not in physical memory can also raise this type of exception because all virtual memory operations occur at hardware interrupt time.

`kPrivilegeViolationException`

Privilege violation exception. The cpu decoded a privileged instruction but was not executing in privileged mode.

`kTraceException`

Trace exception. This exception is used by debuggers to support single-step operations.

`kInstructionBreakpointException`

Instruction breakpoint exception. Not used.

`kDataBreakpointException`

Data breakpoint exception. This exception is used by debuggers to support data breakpoint operations.

`kIntegerException`

Integer exception. This exception is not used by PowerPC processors.

`kFloatingPointException`

Floating-point arithmetic exception. The floating-point processor has exceptions enabled and an exception has occurred.

`kStackOverflowException`

Stack overflow exception. The stack limits have been exceeded and the stack cannot be expanded.

`kTaskTerminationException`	Termination exception. Not used.
`kTaskCreationException`	Creation exception. This exception is used to notify debuggers of the creation of a new task. When the debugger is notified, the PC points to the task's entry point. For additional information see the description of the `DSWaitForException` function in the chapter "Debugger Services Reference."

If the exception kind indicates a problem with memory, you can examine the memory exception structure (page 8-11) to determine the logical address reference that caused the exception, the area containing that address, and the type of operation (read, write, or fetch) being performed.

Memory Reference Enumeration

For memory related exceptions, the microkernel returns a memory exception structure (page 8-11). The field `theReference` of this structure specifies the kind of memory access operation that caused the exception.

```
enum {
    kWriteReference     = 0,
    kReadReference      = 1,
    kFetchReference     = 2,
};
typedef unsigned long MemoryReferenceKind;
```

Field descriptions

`kWriteReference`	The exception occurred during a write operation.
`kReadReference`	The exception occurred during a read operation.
`kFetchReference`	The exception occurred during a fetch operation. (Not all processors are able to distinguish read operations from fetch operations. As a result, fetch operation failures might be reported as failed read operations.)

Exception Information Structure

You examine the exception information structure to determine the state of the cpu at the time an exception occurs. If you are writing an exception handler, it will return the exception information structure to you in its parameter `theException`. If you are writing a debugger, you can obtain the exception information structure by calling the `DSWaitForException` function. The exception information structure is a field of the exception structure returned by this function. (The `DSWaitForException` function is described in "Debugging Services Reference.")

The `ExceptionInformationPowerPC` data type defines an exception information structure.

```
struct ExceptionInformationPowerPC {
    ExceptionKind               theKind;
    MachineInformationPowerPC   *machineState;
    RegisterInformationPowerPC  *registerImage;
    FPUInformationPowerPC       *FPUImage;
    ExceptionInfo               info;
};
typedef struct ExceptionInformationPowerPC ExceptionInformationPowerPC;
```

Field descriptions

`theKind`	One of the exception kind enumeration values (page 8-3), specifying the kind of exception that has occurred.
`machineState`	A pointer to a machine information structure (page 8-8). This structure specifies the values stored in the machine's special purpose registers the time the exception occurred.
`registerImage`	A pointer to a general purpose register information structure (page 8-9) specifying the values stored in the machine's general purpose registers at the time the exception occurred.
`FPUImage`	A pointer to an FPU register information structure (page 8-10) specifying the values stored in the machine's floating point registers at the time the exception occurred.
`info`	Additional information about the memory access violation if the value specified for the field `theKind` is a memory-related exception. This information is specified by a memory exception information structure (page 8-11).

Machine Information Structure

You examine the machine information structure to determine the contents of the cpu's special registers at the time an exception occurs. The machine information structure is a field of the exception information structure (page 8-6).

The `MacineInformationPowerPC` data type defines a machine information structure.

```
structMachineInformationPowerPC {
    UnsignedWide    CTR;
    UnsignedWide    LR;
    UnsignedWide    PC;
    unsigned long   CR;
    unsigned long   XER;
    unsigned long   MSR;
    unsigned long   ExceptKind;
    unsigned long   DSISR;
    UnsignedWide    DAR;
    UnsignedWide    Reserved;
};
```

Field descriptions

`CTR`	The contents of the Count Register.
`LR`	The contents of the Link Register
`PC`	The contents of the Program Counter Register
`CR`	The contents of the Condition Register
`XER`	The contents of the Fixed-Point Exception Register.
`NSR`	The contents of the Machine State Register.
`MQ`	The contents of the MQ register. This register is part of the Power architecture (used by 601 processors only). It is used by Power instructions that perform certain kinds of arithmetic operations.
`ExceptKind`	A constant specifying the type of exception that occurred. Possible values are given by the exception kind enumeration (page 8-3).
`DSISR`	The contents of the Data Access Exception Source Instruction Service Register. This register holds additional information about memory-related exceptions.

`DAR`	The contents of the Data Access Register. Normally, this is the address that caused a data access exception.
`Reserved`	Reserved for future use.

General Purpose Register Information Structure

You examine the `registerImage` field of the exception information structure (page 8-6) to determine the contents of the general purpose registers at the time that an exception occurs. The `registerImage` field points to a general purpose register information structure.

The `RegisterInformationPowerPC` data type defines a general purpose register information structure.

```
struct RegisterInformationPowerPC {
    UnsignedWide    R0;
    UnsignedWide    R1;
    UnsignedWide    R2;
    UnsignedWide    R3;
    UnsignedWide    R4;
    UnsignedWide    R5;
    UnsignedWide    R6;
    UnsignedWide    R7;
    UnsignedWide    R8;
    UnsignedWide    R9;
    UnsignedWide    R10;
    UnsignedWide    R11;
    UnsignedWide    R12;
    UnsignedWide    R13;
    UnsignedWide    R14;
    UnsignedWide    R15;
    UnsignedWide    R16;
    UnsignedWide    R17;
    UnsignedWide    R18;
    UnsignedWide    R19;
    UnsignedWide    R20;
    UnsignedWide    R21;
    UnsignedWide    R22;
    UnsignedWide    R23;
    UnsignedWide    R24;
    UnsignedWide    R25;
    UnsignedWide    R26;
    UnsignedWide    R27;
    UnsignedWide    R28;
    UnsignedWide    R29;
    UnsignedWide    R30;
    UnsignedWide    R31;
};
typedef struct RegisterInformationPowerPC RegisterInformationPowerPC;
```

Field descriptions

`R0 - R31`	The contents of general-purpose registers GPR0 through GPR31.

Floating Point Information Structure

You examine the `FPUImage` field of the exception information structure (page 8-6) to determine the contents of the floating point registers and the contents of the Floating-Point Status and Control Register at the time that an exception occurs. The `FPUImage` field points to a floating point information structure.

The `FPUInformationPowerPC` data type defines a floating point information structure.

```
struct FPUInformationPowerPC {
    UnsignedWide    Registers [32];
    unsigned long   FPSCR;
    unsigned long   Reserved;
};
typedef struct FPUInformationPowerPC FPUInformationPowerPC;
```

Field descriptions

`Registers`	A thirty-two element array. The contents of the *n*th floating point (FPU) register is stored in the corresponding element of the array.
`FPSCR`	The contents of the Floating-Point Status and Control Register (FPSCR).
`Reserved`	Reserved for future use.

Memory Exception Information Structure

You examine the memory exception information structure to obtain more detailed information about an exception caused by a memory violation

The `info` field of the exception information structure (page 8-6) points to a memory exception information structure. The microkernel fills in the structure referenced by the `info` field if the field `theKind` of the exception information structure specifies a memory-related exception.

The `MemoryExceptionInformation` data type defines a memory exception information structure.

```
struct MemoryExceptionInformation {
    AreaID                          theArea;
    LogicalAddress                  theAddress;
    OSStatus                        theError;
    MemoryReferenceKind             theReference;
};
typedef struct MemoryExceptionInformation MemoryExceptionInformation;
```

Field descriptions

`theArea`	The ID of the area that contains the address that caused the memory-related exception. When the memory reference that caused the exception is to an unmapped range of the logical address space, this field contains the value `kNoAreaID`.
`theAddress`	The logical address that caused the memory-related exception
`theError`	A status value. When the exception kind is `unresolvablePageFaultException`, this field contains a value that indicates the reason the page fault could not be resolved.
`theReference`	One of the values specified by the memory reference enumeration (page 8-6). This value indicates the type of operation (read, write, or fetch) that was being performed when the exception occurred.

Functions

This section describes the prototype of a user-defined exception handler and describes the function you use to install the handler for a task or accept function.

InstallExceptionHandler

Installs a user-defined exception handler

```
ExceptionHandler InstallExceptionHandler (ExceptionHandler theHandler,
                          void * refcon);
```

`theHandler`	The address of the exception handler to be installed. The exception handler is a user-defined, native PowerPC function whose prototype is described next.
`refcon`	A 32-bit value that you want to pass to the exception handler when it is invoked. If you are implementing a long jump type of handler, you can use this parameter to store the address of the jump buffer.
function result	The address of any existing exception handler. If there is not an exception handler installed for the current execution context, the `InstallExceptionHandler` function returns `nil`.

DESCRIPTION

You use the `InstallExceptionHandler` function to install the exception handler specified by the parameter `theHandler`. That handler replaces any existing exception handler associated with the current execution context. The newly installed handler remains active until you install some other handler or until you remove the current handler by calling `InstallExceptionHandler` with `theHandler` set to `nil`.

You use the `InstallExceptionHandler` function to install exception handlers for tasks and accept functions. For hardware interrupts, secondary interrupts, software interrupts, and debuggers, you specify the exception handler as a parameter to the function you use to create the interrupt or install the debugger. However, you can still use the `InstallExceptionHandler` function to install an exception handler for this type of code if you are interested in passing a `refcon` value to the exception handler.

Exception handlers installed by software interrupt handlers are in effect only for the duration of that particular invocation of the software interrupt.

SPECIAL CONSIDERATIONS

The `InstallExceptionHandler` function is available to any code executing in the PowerPC native environment. You do not need to call it if your application or other software exists as 680x0 code and hence executes under the 68LC040 Emulator on PowerPC processor-based Macintosh computers.

EXECUTION ENVIRONMENT

Reentrant?	**Call at secondary interrupt level?**	**Call at hardware interrupt level?**
Yes	Yes	Yes

CALLING RESTRICTIONS

This function is reentrant and can be run in privileged mode. You can call this function from any level.

User-Defined Routines

This section describes exception handlers, native PowerPC routines that you write to handle specific types of exceptions that might occur during the execution of your software.

MyExceptionHandler

An exception handler must consist of native PowerPC code and has the following prototype:

```
OSStatus (*ExceptionHandler) (ExceptionInformationPowerPC *theException,
                        void * refcon);
```

`theException`	A pointer to an exception information structure (page 8-6). On return the microkernel fills in the fields of this structure with information that describes the exception that caused the handler to execute and the state of the cpu at the time the exception occurred.
`refcon`	A 32-bit value. This is a user-defined value that you pass as a parameter to the `InstallExceptionHandler` function and that the microkernel passes to your exception handler when it calls it.
function result	If the handler returns `NoErr`, this means that it has resolved the exception and that the task giving rise to the exception can resume execution. If the handler returns any other result, the microkernel terminates the task that caused the handler to be invoked and, if this is a main task, the microkernel terminates the process to which the task belongs.

DESCRIPTION

You pass the address of the function `MyExceptionHandler` to the `InstallExceptionHandler` function. If the `InstallExceptionHandler` function returns successfully, the microkernel calls your exception handler for all exceptions that arise in your application's context**.**

Your exception handler can take whatever steps are necessary to handle the exception or to correct the error or special condition that caused the exception. This includes changing the machine state, by changing the fields of the exception information structure that is passed to it when the handler is invoked.

■ If your handler is successful, it should return the `noErr` result code. If you pass back `noErr`, the microkernel restores the machine state to the state contained in the exception information structure referenced by the parameter `theException` and resumes execution.

- If your handler is not able to handle the exception, it should return some other result code. However, if your handler returns a nonzero result code, the current task is likely to be terminated. If it is a main task, the microkernel process to which it belongs is also terminated.

The microkernel executes an exception handler on the same stack that was active when the exception occurred. To ensure that no stack data is destroyed, the microkernel advances the stack pointer prior to calling the exception handler.

If an exception is raised as a result of stack overflow (`kStackOverflowException`), the microkernel still needs to use the excepted stack to run the exception handler. In this case, it will cut the stack back before running the handler. Consequently handlers that are responding to stack overflows cannot make any assumptions about the state of the stack when they begin to execute or when they return control to the application.

SPECIAL CONSIDERATIONS

It is especially important that you provide exception handlers for secondary interrupts and hardware interrupts. Failing to do so can result in a system crash.

An exception handler must follow the same general guidelines as other kinds of asynchronous software.

An exception handler must be reentrant if it can itself generate exceptions.

Messaging Service Reference

Contents

Messaging Service Reference

Messaging is an interprocess communications service that allows a message to be sent from a task, the **sender**, to another piece of software (a task or an accept function) the **receiver**. The **message** is a contiguous set of bytes (which can be zero bytes) that is understood only by the sender and receiver; it is not interpreted by the messaging service. A message is always associated with a **reply**, which is a response (that might also be zero bytes long) from the receiver to the sender. Thus, the message and its reply form a transaction between the sender and receiver.

Messages are sent to message objects; they are received from ports. Thus, a message object must be associated with a port. Typically, many objects are associated with a port. Before you can send a message, the message object to which you send the message and its port must exist. The receiver typically creates the port and message objects to which messages are sent by senders.

A **message object** represents something to which a request can be made. For example, a message object might represent a window or dialog box that can be used to formulate a search of a database (and display the results of the search), in which case the message contains the request, or a message object might represent a file that can be read or written to, in which case the message contains the instruction to read or write blocks on disk, and so on. Messages sent from many different tasks can be sent to the same message object.

A **port** represents the place where a message is delivered after it is sent to a message object. The receiver can retrieve a message from a port, take some action, and then issue a reply. You can have multiple receivers waiting on messages from the same port.

Messages arrive at message objects in chronological order; however, messages are not guaranteed to arrive at a port in chronological order. This situation can occur when messages are sent to different message objects associated with the same port.

The following sections describe the constants, data types, and functions that you can use to send, receive, and reply to messages and to manipulate messages, message objects, and ports. For a conceptual overview of the messaging service and examples of how to use it, see (To Be Provided)

Constants and Data Types

The following sections describe the constants and data types that can be used with messaging service functions. These constants and data types define:

- identifiers
- message types
- message content
- message and port information
- message object and port options
- send, receive, and accept options

Messaging Service Identifiers

Identifiers used by messaging services have the same purpose and use as identifiers used elsewhere within the microkernel and operating system—they uniquely identify an object or entity within the system. The following identifiers are used by messaging service functions:

```
typedef struct OpaqueMessageID* MessageID;
typedef struct OpaqueObjectID* ObjectID;
typedef struct OpaquePortID* PortID;
typedef struct OpaqueReceiveID* ReceiveID;
```

Identifier

`MessageID`	Unique identifier for a message.
`ObjectID`	Unique identifier for a message object.
`PortID`	Unique identifier for a port.
`ReceiveID`	Unique identifier for a request to receive a message.

Message Types

All messages are identified by a message type, which can be used to specify the kind of message. For example, you could decide that an “open connection” message is one type and a “retrieve data request” is another type. The sender of the message can specify the message type. The receiver of the message (a task or accept function) can use the message type to determine how to respond to the message. You can also use multiple receivers, each set up to handle specific types of messages.

A message type is a 32-bit mask:

```
typedef UInt32 MessageType;
```

Message types are specified in functions that send messages and in functions that receive messages. You specify the message type of the message being sent when you send the message. A receiver specifies the message types that it wants to handle. The sender's message type is bitwise ANDed with the message type specified by the receiver. If the result is non-zero, the receiver is allowed to receive the message. If the result is zero, another receiver is allowed to respond to the message.

If the result is non-zero for several receivers, the following rules determine which receiver will actually receive the message:

1. If any receiver is an accept function (page 9-52), the accept function receives the message.
2. If none of the receivers are accept functions, the highest priority task receives the message. If several tasks with the same priority are waiting, the task waiting the longest receives the message.

If the result of the AND operation is zero for all receivers, the message is queued.

Note
You can have at most one accept function associated with a port. Only receivers waiting on the port are eligible to receive a message delivered to the port. ◆

The first four bits in the mask specifies whether the message was sent by the kernel. You should not set this bit when sending a message.

Mac OS 8 defines two message types for you. If you specify `kAllMessages` in your function call to receive a message, it is eligible to receive every message delivered to the port. If you specify `kAllNonKernelMessageTypes`, messages sent by the kernel cannot be received; the receiver is eligible to receive all other messages.

```
#define kAllNonKernelMessageTypes   (SInt32)0x0FFFFFFF
#define kAllMessages                (SInt32)0xFFFFFFFF
```

You should not use a message type of `(SInt32)0x0000000`, because a bitwise AND of a zero always results in zero. A message with this message type can be received by any receiver whose message type is `kAllMessages`, however, because the `kAllMessages` message type specifies a match on all messages, regardless of the message type.

Message Definition

The message contents and other information about a message are returned in a message control block by functions that receive messages. The control block specifies the complete contents of a message, which include the contents and other information. If the receiver is a task, the control block is assembled in the buffer specified by the function that receives the message. If the receiver is an accept function, the control block is assembled in the microkernel's memory.

The message format is defined as a `MessageControlBlock` structure, which can be referenced by a `MessageControlBlockPtr` pointer. The definitions of the structure and pointer are as follows:

```
struct MessageControlBlock {
    MessageID           message;            /* message ID */
    AddressSpaceID      addressSpace;       /* sender's address space */
    KernelProcessID     sendingKernelProcess; /* sender's process */
    TaskID              sendingTask;        /* sending task */
    void *              refcon;             /* msg. object's ref. con. */
    SendOptions         options;            /* send options */
    MessageType         theType;            /* message type */
    LogicalAddress      messageContents;    /* message buffer */
    ByteCount           messageContentsSize;/* message buffer size */
    LogicalAddress      replyBuffer;        /* reply buffer */
    ByteCount           replyBufferSize;    /* reply buffer size */
    OSStatus            currentStatus;      /* message status */
    UInt32              reserved[4];        /* reserved */
};
typedef struct MessageControlBlock MessageControlBlock;

typedef MessageControlBlock *MessageControlBlockPtr;
```

Field descriptions

`message`	The message ID, which is created by the operating system to uniquely define the message. Use this ID to reply to the message, cancel a message sent asynchronously, or to obtain information about the message.
`addressSpace`	The address space of the task that sent the message. You do not need to know the address space if you are only interested in the contents of the message; however, you can use this address space for other purposes, such as accessing non-message data in the sending task's address space.
`sendingKernelProcess`	The process ID of the sending task's process. You can use this ID, for example, to verify that the sending process is authorized to send the message.
`sendingTask`	The task ID of the task that sent the message.
`refcon`	The reference constant associated with the message object to which the message was sent. For examples of using a reference constant, see(To Be Provided)
`theType`	The message type. For information about message types, see "Message Types" (page 9-4).
`messageContents`	The message data. You always reference the message data with this logical address. The logical address may be the address of the buffer that you specified in the function that receives the message or it may be the address of the data in the sender's address space, depending on how the data was transferred to the receiver. For information on how data is transferred, see "Send Options" (page 9-14).
`messageContentsSize`	The size of the message data, in bytes.
`replyBuffer`	The logical address of the sender's reply buffer, or `nil`. A `nil` value indicates that the sender either doesn't have a reply buffer or that the microkernel chose not to map the buffer into the receiver's address space. The reply data will not be copied to the sender's address space when you can use the sender's reply buffer, that is, when this field is not `nil`. If the reply buffer exists (that is, if `replyBufferSize` is not zero), data can be sent back as part of the reply.

`replyBufferSize`	The size of the reply buffer or zero if the sender did not specify a reply buffer. The receiver must also specify the size of the reply buffer when calling `ReplyToMessage` (page 9-48) or `ReplyToMessageAndRecieve` (page 9-49).
`currentStatus`	Unused.
`reserved[4]`	Reserved for future use.

Message Information

This section describes message information version and structure.

Message Information Version

The message information version specifies which version of the message information structure is currently being used. The version information provides backward compatibility for the message information structure—you specify the version to match the information structure you use.

```
enum {
    kMessageInformationVersion  = 1
};
```

The message information version is specified as follows:

Enumeration

`kMessageInformationVersion`	The current version of the message information structure.

Message Information Structure

The message information structure is primarily useful for debugging, because all information needed to handle a message is available when the message is received. (See “Message Definition” (page 9-6).) You use the message information structure when calling the `GetMessageInformation` function (page 9-20) to return information about a message. Message information is defined as a `MessageInformation` structure, which can be referenced by a `MessageInformationPtr`:

```
struct MessageInformation {
    ObjectID                    object;             /* message object */
    TaskID                      sendingTask;        /* sender’s task */
    KernelProcessID             sendingKernelProcess;/* task’s process */
};

typedef struct MessageInformation MessageInformation;

typedef MessageInformation *MessageInformationPtr;
```

Field descriptions

`object`	The message object to which the message was sent.
`sendingTask`	The task ID of the task that sent the message.
`sendingKernelProcess`	The process ID of the process with which the sending task is associated.

Message Object Options

You send message to a message object. A message object is associated with a port. This section describes options you can specify when you:

- create a message object
- lock a message object
- set a message object's characteristics

Object Creation Options

Object creation options allow additional control over how objects are created. They are set by the `CreateObject` function (page 9-21) that you call to create a message object. The object options are type `ObjectOptions`, which are specified as an `OptionBits` data type:

```
typedef OptionBits ObjectOptions; /* message object's create options */
```

No object options are currently defined; you should specify `kNilOptions` for object creation options.

Object Locking Options

The message object lock options are used by the `LockObject` function (page 9-23). The message object lock options are type `ObjectLockOptions`, which are specified as an `OptionBits` data type:

```
typedef OptionBits ObjectLockOptions; /* message object's lock options */
```

The only message object lock option currently defined is specified as follows:

```
enum {
    kLockObjectWithOneMessage   = 0x00000001    /* allow locking while
                                                   one message is outstanding */
};
```

Enumeration

`kLockObjectWithOneMessage`	Allow the message object to become locked even if a message has been received from the message object's port but has not yet been replied to. If this option bit is not set, the message object becomes locked only when all messages that have been received have also been replied to. Thus, at most one message can be outstanding when the lock is acquired (when `kLockObjectWithOneMessage` is specified); otherwise, no messages can be outstanding.

Object Setting Options

Object setting options specify which of the object's properties are to be changed when you call the `SetObjectInformation` function (page 9-27). These options may be ORed together to specify changing more than one property in the same function call. The object options are type `SetObjectOptions`, which are specified as an `OptionBits` data type:

```
typedef OptionBits SetObjectOptions;
```

The object setting options are specified as follows:

```
enum {
    kSetObjectPort      = 0x00000002,   /* set the port */
    kSetObjectRefcon    = 0x00000004    /* set the ref. constant */
};
```

Enumeration

`kSetObjectPort`	Set the port with which the message object is associated.
`kSetObjectRefcon`	Set the value of the message object's reference constant.

Port Information and Options

This section describes port-related information and options:

- options you specify when you create a port
- version information
- the port information structure

Port Creation Options

Port options allow additional control over how ports are created. You specify them when you call the `CreatePort` function (page 9-30) to create a port. The port creation options are type `PortOptions`, which are specified as an `OptionBits` data type:

```
typedef OptionBits PortOptions;     /* create port options */
```

No port options are currently defined; you should specify `kNilOptions` for port options.

Port Information Version

The port information version specifies which version of the port information structure is being used. The version information provides backward compatibility for the port information structure—you can specify the version of the structure you are using.

```
enum {
    kPortInformationVersion     = 1
};
```

The port information version is specified as follows:

Enumeration

`kPortInformationVersion`
The current version of the port information structure.

Port Information Structure

You use the port information structure when calling the `GetPortInformation` function (page 9-32) to return information about the current state of a port, its owning process, and transaction statistics. Port information is defined as a `PortInformation` structure, which can be referenced by a `PortInformationPtr` pointer.

```
struct PortInformation {
    KernelProcessID         owningKernelProcess;/* creator process */
    MessageAcceptProc       acceptProc;         /* accept function */
    ExceptionHandler        acceptHandler;      /* accept function’s
                                                 exception handler */
    AcceptOptions           theAcceptOptions;   /* accept optios */
    void *                  acceptRefcon;       /* accept function’s
                                                 reference constant */
    ItemCount               objectCount;        /* count of message
                                                 objects associtated
                                                 with the port */
    ItemCount               pendingReceives;    /* count of outstanding
                                                 receive requests */
    ItemCount               pendingSends;       /* count of outstanding
                                                 messages (queue size) */
    ItemCount               pendingReplies;     /* count of received but
                                                 unreplied messages */
    ItemCount               transactionCount;   /* count of replied to
                                                 messages */
    ItemCount               blockedAsyncSenders;/* count of async senders
                                                 waiting on resources */
    ItemCount               blockedAsyncReceivers;  /* count of async
                                        receivers waiting on resources */
};

typedef struct PortInformation PortInformation;

typedef PortInformation *PortInformationPtr;
```

Field descriptions

`owningKernelProcess`	The process ID of the process that created this port.
`acceptProc`	The accept function that is currently registered for this port or `nil` if an accept function is not currently registered.
`acceptHandler`	The exception handler for the currently registered accept function or `nil` if an exception handler is not currently registered.
`theAcceptOptions`	The options associated with the accept function.
`acceptRefcon`	The reference constant associated with the accept function.
`objectCount`	The number of message objects that are associated with the port.
`pendingReceives`	The number of receive requests that have not been matched with a message.
`pendingSends`	The number of messages that have been sent to the port's message objects but have not been matched with a receiver (either a function call to receive a message or an accept function).
`pendingReplies`	The number of messages that have been received but not yet replied to.
`transactionCount`	The number of send-receive-reply sequences delivered through the port since it was created.
`blockedAsyncSenders`	The number of asynchronous send requests that are waiting for message system memory resources.
`blockedAsyncReceivers`	The number of asynchronous receive requests that are waiting for message system memory resources.

Send Options

Send options specify how to send the message. They are set when calling a function that sends a message and are available in the message control block that is returned to the function that receives the message. Mac OS 8 may also set options in the message control block being returned to specify additional information about how a message was sent.

The send options are type `SendOptions`, which are specified in an `OptionBits` data structure:

```
typedef OptionBits SendOptions;
```

The option bits for send options are as follows:

```
enum {
    kSendTransferKindMask       = 0x00000003,       /* Set by sender*/
    kSendByChoice               = 0x00000000,       /* kernel choice */
    kSendByReference            = 0x00000001,       /* by reference */
    kSendByValue                = 0x00000002,       /* by value */
    kSendIsBuffered             = 0x00000003,       /* buffer by kernel*/
    kSendIsPrivileged           = 0x00000008,       /* Set by kernel*/
    kSendIsAtomic               = 0x00000010,       /* Set by sender*/
    kSendPtrsAddressable        = 0x00000020,       /* Set by kernel*/
    kSendPtrsNeedAccessCheck    = 0x00000040        /* Set by kernel*/
};
```

Enumeration

`kSendTransferKindMask`	The mask in the `SendOptions` structure that specifies how to transfer the message. The mask may specify `kSendByChoice`, `kSendByReference`, `kSendByValue`, or `kSendIsBuffered` as described below.
`kSendByChoice`	Allow Mac OS 8 to choose the fastest way to transfer a message, which is either `kSendByReference` or `kSendByValue`. This choice partially depends on the kind of receiver (task or accept function), the message size, the mode (user or supervisor), the address space of the sender and receiver, as well as whether or not enough space is allocated for the message in the receiver. You should use the `kSendByChoice` option whenever possible, because it is not always possible to determine best choice in advance unless you know the environment in which the message is sent and received.
`kSendByReference`	Send the message contents as a reference to its address in the sender's address space. When `kSendByReference` is specified, the message contents holds the address of the sender's buffer that contains the data. If the message's receiver is in a different address space than that of the

	sender, the microkernel maps the buffer that contains the message into the address space of the receiver. The `kSendByReference` option is especially useful when the message contains a large data structure and the receiver needs to read some values in the structure and update other values in the structure.
	If the microkernel maps the message buffer into the receiver's address space, data that resides on the same page of physical memory as the buffer, either before the start of the buffer or after the end of the buffer, is also mapped into the receiver's address space, thus, making this data accessible to the receiver along with the buffer itself.
`kSendByValue`	Send the message by copying the message contents from the sender's buffer to the receiver's buffer. The receiver's buffer must be large enough to receive the entire message; otherwise, the sending function receives an error.
`kSendIsBuffered`	Use a buffer in the microkernel to hold the message contents until the message is received. This option is allowed only when the message is sent asynchronously—the sender can reuse the buffer specified in the function that sends the message as soon as the function returns, instead of requiring the sender to keep the buffer and its contents until a reply is received. An error is reported if the microkernel cannot buffer the message (for example, if the message is too large or there are too many messages being buffered in this way). An error is also reported if the buffer size is larger than `maxBufferedMessageSize`. (Call `GetSystemInformation` (page(To Be Provided)) to determine the maximum buffered message size.)
	You should only use the `kSendIsBuffered` option if you cannot keep the buffer while waiting for a reply; for example, when the buffer is created on the sending task's stack. The use of this option results in the data being copied twice, once from the sender's buffer to one provided by the microkernel and once from the microkernel's buffer to the receiver's buffer.
`kSendIsPrivileged`	The microkernel sets this option bit if the sending task is executing in supervisor mode when the message is sent. It provides information to the receiver about the sender's

	mode. If the sending task is in user mode and this bit is specified as an option, it is cleared.
`kSendIsAtomic`	Make the send-receive-reply sequence atomic with respect to the message object to which the message is sent. When the message is sent to a message object, the object becomes locked until the message is replied to. Thus, other messages sent to the object while it is locked become blocked until a reply is issued for the first message; if another message is sent synchronously, its sending task blocks.
`kSendPtrsAddressable`	The microkernel sets this option bit if the sending task's address space is directly addressable by the receiver. Thus, this option bit is set when the sender and receiver are both executing in the same address space (for example, two user mode tasks in the same process), if they are both supervisor-mode tasks, or if the receiver is an accept function.
`kSendPtrsNeedAccessCheck`	The microkernel sets this option bit if the sending task's address space is directly addressable by the receiver and the sender has different access rights to that address space than does the receiver. If this option bit is set, the receiver must explicitly check the access rights of the message buffer before accessing the data. This option bit is always set if the sender and receiver are in different address spaces. It is also set if the sender and receiver execute in different modes; for example, when a user-mode task sends a message that is received by an accept function. In this case, the accept function can directly access the sender's buffer; however, it should perform an address check because the buffer may contain a pointer to data that is read/write accessible in supervisor mode (the mode in which the accept function executes) yet is read-only in user mode.

Receive Options

Receive options allow additional control over how messages are received. They are set by the function that receives the message. The receive options are type `ReceiveOptions`, which are specified in an `OptionBits` data structure:

```
typedef OptionBits ReceiveOptions;
```

The option bits for receive options are defined as follows:

```
enum {
    kReceiveNoAddressTranslation    = 0x00000002 /* not addr. mapping */
};
```

Enumeration

`kReceiveNoAddressTranslation`

Do not allow the message content buffer to be mapped into the receiving task's address space. This option prevents Mac OS 8 from mapping the addresses that contain the data in the sender's address space to the receiving task's address space, which could happen if the sending options are either `kSendByReference` or `kSendByChoice`. If you specify this option, you cannot use the buffer if it is mapped; however, you can pass the buffer to another task which does allow the mapping.

Accept Options

Accept options allow additional control over how accept functions are executed. They are set when you call the `AcceptMessage` function (page 9-45) to register an accept function with a port. The accept options are type `AcceptOptions`, which are specified in an `OptionBits` data structure:

```
typedef OptionBits AcceptOptions;
```

The option bits for accept options are defined as follows:

```
enum {
    kAcceptFunctionIsResident=  0x00010000
};
```

Enumeration

`kAcceptFunctionIsResident`
: Specify this option if you want to ensure that the accept function is resident in memory.

Accept Function

You can use the `MessageAcceptProc` function to define an accept function. The accept function is invoked when a message with the specified message type is received by the port to which the accept function is associated. This function is defined as follows:

```
typedef OSStatus    (*MessageAcceptProc)
                     const MessageControlBlock *message,
                     void *acceptRefcon);
```

For information about creating your own accept function, see the description of the `MyMessageAcceptProc` function (page 9-52).

Messaging Service Functions

Messaging Service functions allow you to implement messaging in your application. These functions are grouped into the following categories:

- obtaining information about messages
- manipulating message objects
- manipulating message ports
- sending messages
- receiving messages
- replying to messages
- canceling asynchronous messages

Obtaining Information About Messages

The `GetMessageInformation` function returns information about a message. To retrieve the contents of a message, you must call a function that receives or accepts messages. These functions are described in “Receiving a Message” (page 9-40). For information about the structure of a message, see “Message Definition” (page 9-6).

GetMessageInformation

Obtains information about a message.

```
OSStatus GetMessageInformation (
                        MessageID theMessage,
                        PBVersion version,
                        MessageInformation *messageInfo);
```

`theMessage`	The message ID (page 9-6) of the message about which you want to obtain information.
`version`	The version number of the message information structure.
`messageInfo`	A pointer to a message information structure (page 9-8). On output, the structure contains information about the message.
function result	(To Be Provided)

DISCUSSION

You specify the version number of the message information structure that you want to use. The structure may change between versions of the operating system. The `GetMessageInformation` function fills in the specified `MessageInformation` structure with information about the message, including the object to which the message was sent, the task that sent the message, and the process that owns the sending task.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For information about message information versions and the message information structure, see “Message Information” (page 9-8).

Creating and Deleting Message Objects

CreateObject

Creates a message object.

```
OSStatus CreateObject (
                    PortID port,
                    void *refcon,
                    ObjectOptions options,
                    ObjectID *theObject);
```

`port`	The port with which the message object is to be associated.
`refcon`	A pointer to a reference constant by which you can provide information to the receiver.
`options`	Message object options (page 9-10). Because no creation options are currently defined, use `kNilOptions` for this parameter.

`theObject`	A pointer to a message object ID. On output, the parameter points to the message object ID that you use to specify the destination for a message.
function result	To Be Provided

DISCUSSION

When you create a message object, you specify the port to associate with the object and a reference constant. Messages sent to the message object are received from the port and the reference constant is available for use by the receiver. You can have more than one message object associated with a port.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

You can change the port or reference constant (or both) with the `SetObjectInformation` function (page 9-27).

DeleteObject

Deletes a message object.

```
OSStatus DeleteObject (ObjectID theObject);
```

`theObject`	The message object ID of the message object you want to delete.
function result	To Be Provided

DISCUSSION

An attempt to send a message to a message object that has been deleted results in an error return from functions used to send the message. If the message has already been sent but not yet replied to, the message is canceled. If the message object is locked when it is deleted, any task waiting to send a message to it is unblocked and the function that sent the message returns an error.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For descriptions of functions that can send messages, see “Sending a Message” (page 9-34).

For information about canceling a message, see “CancelAsyncReceive” (page 9-45).

Locking and Unlocking Message Objects

LockObject

Locks a message object.

```
OSStatus LockObject (ObjectID theObject,
                     ObjectLockOptions options,
                     Duration timeout);
```

`theObject`	The ID of the message object that you want to lock.
`options`	The message lock options. If there are messages that have been received by the port but not replied to, the message object's lock options (page 9-10) control when the lock is granted. If `kNilOptions` is specified, all replies must have been sent before the lock is granted. If `kLockObjectWithOneMessage` is specified, a single outstanding message is allowed. The task that calls the `LockObject` function blocks until the lock is granted, meaning that the messages have been replied to or until the lock is no longer held by other tasks.
`timeout`	A value that specifies the maximum amount of time to wait when attempting to lock the message object.
function result	To Be Provided

DISCUSSION

Locking a message object prevents messages sent to the message object from being received by the message object's port. Messages sent to the message object while it is locked wait for the message object to become unlocked. This is useful in a client-server model when the server wants to change the message object's characteristics (for example, its reference constant) upon receiving a message. The server can receive the message and lock the message object, which prevents messages from being received at the port. (The `kLockObjectWithOneMessage` option must be specified.) It can then change the characteristics and unlock the message object.

IMPORTANT

In the client-server scenario described above, you must use the `kLockObjectWithOneMessage` option; otherwise, the task that receives the message will deadlock waiting for the lock because it has not replied to the message. You should consider using the `kSendIsAtomic` option when sending the message to achieve the same result. ▲

Multiple tasks may attempt to lock a message object; however, only one task can acquire the lock at a time. Other tasks that attempt to lock the message object are blocked in priority order while waiting for the message object to become unlocked.

Any messages previously sent to the message object but which have not yet been received from the port are removed from the port's queue and wait to be queued on the port again when the message object becomes unlocked. Any messages that already have been received from the port are not affected by the attempt to lock a message object.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To unlock a message object, call the `UnlockObject` function (page 9-25).

For information about the `kSendIsAtomic` option, see "Send Options" (page 9-14).

UnlockObject

Unlocks a message object.

```
OSStatus UnlockObject (ObjectID theObject);
```

`theObject`	The ID of the message object that you want to unlock.
function result	To Be Provided

DISCUSSION

When the `UnlockObject` function is called, waiting messages are sent to the message object's port, where they are queued for receipt. You should consider using the `kSendIsAtomic` option when sending the message to achieve the same result as locking and unlocking the message object.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For information about the `kSendIsAtomic` option, see "Send Options" (page 9-14).

Getting and Setting Message Object Information

GetObjectInformation

Obtains information about the message object.

```
OSStatus GetObjectInformation(
                     ObjectID theObject,
                     PortID *port,
                     void **refcon);
```

`theObject`	The ID of the message object whose characteristics you wish to determine.
`port`	On output, the port ID associated with the message object.

`refcon`	On output, the contents of the reference constant for this message object.
function result	To Be Provided

SEE ALSO

For information about changing the port ID and reference constant for a message object, see the `SetObjectInformation` function (page 9-27).

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SetObjectInformation

Changes the characteristics of a message object.

```
OSStatus SetObjectInformation(
                     ObjectID theObject,
                     SetObjectOptions options,
                     PortID port,
                     void *refcon);
```

`theObject`	The ID of the message object whose characteristics you wish to change.
`options`	Message object options (page 9-10) that specify which characteristics of the message object you wish to change.
`port`	The port with which the message object is to be associated.

`refcon`	A pointer to a reference constant by which you can provide information to the receiver.
function result	To Be Provided

DISCUSSION

The characteristics for a message object are set initially by the `CreateObject` function (page 9-21). The options parameter specifies the characteristics to change. You can specify `kSetObjectPort` to change the port with which the object is associated, `kSetObjectRefcon` to change the value of the object's reference constant, or `kSetObjectPort | kSetObjectRefcon` to change both of them.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To determine the current characteristics for a message object, call the `GetObjectInformation` function (page 9-26).

GetObjectsInPort

Identifies message objects associated with a port.

```
OSStatus GetObjectsInPort(
                    PortID port,
                    ItemCount requestedObjects,
                    ItemCount *totalObjects,
                    ObjectID *theObjects);
```

`port`	The port whose message objects you wish to identify.
`requestedObjects`	The maximum number of message objects you wish to identify.
`totalObjects`	A pointer to an item count. On output, the item count contains the total number of message objects associated with the port.
`theObjects`	A pointer to an array of message object IDs. On output, the array contains the message object IDs that are associated with the port. Allocate enough space in the data structure pointed to by the `theObjects` parameter to return the ID of each message object that you request.
function result	To Be Provided

DISCUSSION

The `GetObjectsInPort` function retrieves the message objects associated with the specified port, up to the maximum specified in the `requestedObjects` parameter. The total number of objects associated with the port is returned in the `totalObjects` parameter. If the total number of objects is less than or equal to the number of objects requested, all message object IDs are returned; otherwise, only the requested number of message object IDs are returned.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Creating and Deleting Message Ports

CreatePort

Creates a port.

```
OSStatus CreatePort (PortOptions options,
                     PortID *thePort);
```

`options`	Port options (page 9-11). Because no creation options are currently defined, use `kNilOptions` for this parameter.
`thePort`	A pointer to a port ID. On output, the `CreatePort` function returns the port ID of the newly created port.
function result	To Be Provided

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

DeletePort

Deletes a port.

```
OSStatus DeletePort (PortID thePort);
```

`thePort`	The port ID of the port you want to delete.
function result	To Be Provided

DISCUSSION

The `DeletePort` function deletes the specified port and all message objects associated with the port. Messages sent to any of these message objects are canceled as a result of the message object being deleted. Receive requests on the port are also canceled.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For information about canceling a message, see “CancelAsyncReceive” (page 9-45).

Obtaining Information about Ports

GetPortInformation

Obtains information about a port.

```
OSStatus GetPortInformation(
                    PortID thePort,
                    PBVersion version,
                    PortInformation *portInfo);
```

`thePort`	The ID of the port whose characteristics you wish to determine.
`version`	The version of the port information structure that you are using.
`portInfo`	A pointer to a port information structure (page 9-12). On output, the structure contains the information about the port, including statistics about the current state of the port and information about its accept function.
function result	To Be Provided

DISCUSSION

The port information structure may change between versions of the operating system.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For information about port information versions and the port information structure, see “Port Information and Options” (page 9-11).

GetPortsInSystem

Identifies all the ports that currently exist.

```
OSStatus GetPortsInSystem(
                    ItemCount requestedPorts,
                    ItemCount *totalPorts,
                    PortID *thePorts);
```

`requestedPorts`	The maximum number of ports you wish to identify.
`totalPorts`	A pointer to an item count. On output, the item count contains the total number of ports that currently exist.

`thePorts`	A pointer to an array of port IDs. On output, the array contains the port IDs in the system. Allocate enough space in the data structure pointed to by the `thePorts` parameter to return the ID of each port that you request.
function result	To Be Provided

DISCUSSION

The `GetPortsInSystem` function retrieves the port IDs of all ports systemwide, up to the maximum specified in the `requestedPorts` parameter. The total number of ports systemwide is returned in the `totalPorts` parameter. If the total number of ports is less than or equal to the number of ports requested, all port IDs are returned; otherwise, only the requested number of port IDs are returned.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Sending a Message

Messages can be sent either synchronously, which causes the task that sends the message to block until a reply is received, or asynchronously, which allows task execution to continue and a notification to be delivered to the task when a reply has been received.

Asynchronous messages can be canceled on a “per message” basis. You cannot cancel synchronous messages. The operating system, however, may cancel both synchronous and asynchronous sends. It cancels messages under the following conditions:

- when a timeout for a synchronous send expires before a message is received, the operating system cancels the message.
- when the message object is deleted, the operating system cancels messages sent to the message object that have not yet been received.
- when the message object's port is deleted, the operating system cancels all messages sent to the port (via a message object) that have not been received.

The following sections describe functions you can use to send messages and cancel asynchronous messages.

SendMessageSync

Sends a message synchronously.

```
OSStatus SendMessageSync(
                    ObjectID object,
                    MessageType theType,
                    ConstLogicalAddress messageContents,
                    ByteCount messageContentsSize,
                    LogicalAddress replyBuffer,
                    ByteCount *replyBufferSize,
                    SendOptions options,
                    Duration timeout);
```

`object`	The message object to which the message is sent.
`theType`	The message type. For information about message types, see "Message Types" (page 9-4).
`messageContents`	The logical address of the start of the data to send or `nil` if there is no message data being sent.
`messageContentsSize`	The size of the message data, in bytes.
`replyBuffer`	The logical address of the reply buffer or `NULL` if you do not wish to receive reply data.

`replyBufferSize`
: A pointer to a byte count. On input, specify the size of the reply buffer, in bytes. On output, the byte count specifies the actual number of bytes of data in the buffer.

`options`
: The send options. These options specify how a message is to be sent, such as whether the contents are copied (passed by value) or passed as an address (by reference), whether the message is atomic, and so on. For information about send options, see "Send Options" (page 9-14).

`timeout`
: A timeout value that specifies the maximum amount of time, specified as a type `Duration` (page(To Be Provided)), to wait for a the message to be received. You can specify a value of `kDurationForever` to prevent the message from being canceled. You can specify a value of `kDurationImmediate` when you want to cancel the message immediately if no receiver is waiting for the message.

function result
: To Be Provided

DISCUSSION

The `SendMessageSync` function sends a message to the specified message object and waits for a reply. The task that calls `SendMessageSync` is blocked until either a reply is received or a timeout occurs. If a timeout occurs, the message is canceled and the task becomes unblocked.

Note
The operating system may map the reply buffer into the receiver's address space temporarily; however, it will be unmapped when the message is replied to. ◆

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SendMessageAsync

Sends a message asynchronously.

```
OSStatus SendMessageAsync(
                    ObjectID object,
                    MessageType theType,
                    ConstLogicalAddress messageContents,
                    ByteCount messageContentsSize,
                    LogicalAddress replyBuffer,
                    ByteCount replyBufferSize,
                    SendOptions options,
                    const KernelNotification *notification,
                    ByteCount *replySize,
                    MessageID *theMessage);
```

`object`	The message object that receives the message.
`theType`	The message type. For information about message types, see "Message Types" (page 9-4).
`messageContents`	The logical address of the start of the data to send or `NULL` if there is no message data being sent.
`messageContentsSize`	The size of the message data, in bytes.

`replyBuffer`	The logical address of the reply buffer or `NULL` if you do not wish to receive reply data.
`replyBufferSize`	The size of the reply buffer, in bytes.
`options`	The send options. The send options specify how a message is to be sent, such as whether the contents are copied (passed by value) or passed as an address (by reference), whether the message is atomic, and so on. For information about send options, see "Send Options" (page 9-14).
`notification`	A pointer to a `KernelNotification` structure, which specifies how to receive the notification.
`replySize`	A pointer to a byte count. On output, the byte count contains the actual number of bytes of data returned in the reply buffer.
`theMessage`	A pointer to a message ID. On output, it contains the message ID for this message. You can use the message ID for the message to cancel the message. For more information about canceling messages, see `CancelAsyncReceive` (page 9-45).
function result	To Be Provided

DISCUSSION

The `SendMessageAsync` function sends a message to the specified message object and returns immediately. The task that calls `SendMessageAsync` is not blocked. The sender is notified that the message has been replied to (and thus, the send-receive-reply sequence is complete) by a notification mechanism (either by setting flags in an event group, issuing a software interrupt, or placing an entry in a kernel queue). When the notification occurs, the reply data is available in the reply buffer and the size of the reply data is in the variable pointed to by the `replySize` parameter.

IMPORTANT

Do not reuse or delete the message buffer until being notified that the send operation has completed, unless you specify the `kSendIsBuffered` option. ▲

Note
The operating system may map the reply buffer into the receiver's address space temporarily; however, it will be unmapped when the message is replied to. ◆

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For information about the `KernelNotification` structure, see(To Be Provided)

For information about event groups and flags, see(To Be Provided)

For information about kernel queues, see(To Be Provided)

For information about software interrupts, see(To Be Provided)

CancelAsyncSend

Cancels a message sent asynchronously.

```
OSStatus CancelAsyncSend (MessageID theMessage);
```

`theMessage`	The message you want to cancel.
function result	To Be Provided

DISCUSSION

If the message has not yet been received from the port, it is removed from the port and cannot be received. If the message has been received but not yet replied to, the message is not canceled and the function returns an error.

IMPORTANT

Canceling a message can result in a race condition if the request to cancel is sent just as the message is replied to. It is also possible for the receiver to ignore a request to cancel a message. Thus, a request to cancel may result in the message being replied to anyway; you should anticipate this condition occurring. ▲

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Receiving a Message

Messages can be received by tasks, either synchronously, which causes the task that requests a message to block until a message of the appropriate type is received, or asynchronously, which allows task execution to continue and a notification to be delivered to the task when a message of the appropriate type has been received.

Messages can also be received by an accept functions, which is a function associated with a port that allows efficient transfer of a message from a sending task's address space to the kernel's address space.

You can cancel asynchronous receives. The operating system can cancel both synchronous and asynchronous receives. It cancels all receives that are waiting when the port they are waiting on is deleted. It cancels synchronous receives when the receive request's timeout value expires.

The following sections describe functions you can use to receive messages.

ReceiveMessageSync

Receives a message synchronously.

```
OSStatus ReceiveMessageSync(
                    PortID port,
                    MessageType theTypes,
                    MessageControlBlock *theControlBlock,
                    LogicalAddress buffer,
                    ByteCount bufferSize,
                    ReceiveOptions options,
                    Duration timeout);
```

`port`	The port from which you wish to receive a message.
`theTypes`	The message types of the messages you are willing to receive. For information about message types, see “Message Types” (page 9-4).
`theControlBlock`	A pointer to a message control block. For information about message control blocks, see “Message Definition” (page 9-6).
`buffer`	The logical address of a buffer. On output, the buffer contains the message contents if the message was sent by value. If the message was sent by reference, the buffer is not modified.
`bufferSize`	The size of the buffer you allow for the message content data.
`options`	The receive options. For information about receive options, see “Receive Options” (page 9-18).
`timeout`	A timeout value that specifies the maximum amout of time to wait, specified as a type `Duration` (page(To Be Provided)), to receive a message of the appropriate type. You can specify a value of `kDurationForever` to prevent the receive request from being canceled. You can specify a value of `durationImmediate` if you want to cancel the receive request immediately when no message of the appropriate type is waiting.

function result To Be Provided

DISCUSSION

The `ReceiveMessageSync` function attempts to receive a message that matches the specified message type from the specified port. The task that calls `ReceiveMessageSync` is blocked until either the appropriate message is received or a timeout occurs. If a timeout occurs, the receive request is canceled and the task becomes unblocked.

Messages are not guaranteed to be queued on a port in chronological order. This situation can occur when messages are sent to different message objects associated with the same port. Additionally, the message type controls which message is received, not the message's position within the queue.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

You can also use the `ReplyToMessageAndReceive` function (page 9-49) to reply to a message and receive the next one.

ReceiveMessageAsync

Receives a message asynchronously.

```
OSStatus ReceiveMessageAsync(
                    PortID port,
                    MessageType theTypes,
                    MessageControlBlock *theControlBlock,
                    LogicalAddress buffer,
                    ByteCount bufferSize,
                    ReceiveOptions options,
                    const KernelNotification *notification,
                    ReceiveID *theReceive);
```

`port`	The port from which you wish to receive a message.
`theTypes`	The message types of the messages you are willing to receive. For information about message types, see “Message Types” (page 9-4).
`theControlBlock`	A pointer to a message control block. For information about message control blocks, see “Message Definition” (page 9-6).
`buffer`	The logical address of a buffer. On output, the buffer contains the message contents if the message was sent by value. If the message was sent by reference, the buffer is not modified.
`bufferSize`	The size of the buffer you allow for the message content data.
`options`	The receive options. For information about receive options, see “Receive Options” (page 9-18).
`notification`	A pointer to a `KernelNotification` structure, which specifies how to receive the notification.
`theReceive`	A pointer to a receive ID for this receive. You can use the receive ID to cancel the receive request. (See `CancelAsyncReceive` (page 9-45) for more information.)
function result	To Be Provided

DISCUSSION

The `ReceiveMessageAsync` function requests a message of the specified message type from the specified port. A task that calls `ReceiveMessageAsync` is not blocked. The availability of an appropriate message is delivered by a notification mechanism (either by setting flags in an event group, issuing a software interrupt, or placing an entry in a kernel queue). When the notification occurs, the message control block contains information about the message and the buffer contains the message contents.

Messages are not guaranteed to be queued on a port in chronological order. This situation can occur when messages are sent to different message objects associated with the same port. Additionally, the message type controls which message is received, not the message's position within the queue.

Note
The operating system may map the reply buffer into the receiver's address space temporarily; however, it will be unmapped when the message is replied to. ◆

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For information about the `KernelNotification` structure, see(To Be Provided)

For information about event groups and flags, see(To Be Provided)

For information about kernel queues, see(To Be Provided)

For information about software interrupts, see(To Be Provided)

CancelAsyncReceive

Cancels an asynchronous receive request.

```
OSStatus CancelAsyncReceive (ReceiveID theReceive);
```

`theReceive`	The receive request you want to cancel.
function result	To Be Provided

DISCUSSION

The receive request is canceled. Messages are not affected by cancelling a receive request.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

AcceptMessage

Registers an accept function for the port.

```
OSStatus AcceptMessage(
                     PortID port,
                     MessageType theTypes,
                     MessageAcceptProc acceptProc,
                     ExceptionHandler theExceptionHandler,
                     AcceptOptions options,
                     void *acceptRefcon);
```

`port`	The port associated with the accept function.
`theTypes`	The message types of the messages you are willing to receive. For information about message types, see "Message Types" (page 9-4).
`acceptProc`	An accept function. For the definition of an accept function, see "MyMessageAcceptProc" (page 9-52).
`ExceptionHandler`	The accept function's exception handler. For information about exception handlers, see(To Be Provided)
`options`	Options associated with the accept function. Specify `kAcceptFunctionIsResident` if you want the accept function to remain resident or `kNilOptions` if you do not require it to be resident. For more information about accept options, see "Accept Options" (page 9-18).
`acceptRefcon`	A pointer to the reference constant associated with the accept function.
function result	To Be Provided

DISCUSSION

After registering the accept function specified by the `acceptProc` parameter, all messages sent to the specified port with the appropriate message types will be received by the accept function, regardless of whether there are any other receive requests on the port for the same message type.

Messages sent to the port before the accept function is registered are not handled by the accept function. For this reason, you should call the `AcceptMessage` function before messages are allowed to be sent to the port; for example, after you create the port but before you create the port's message objects.

Only one accept function can be associated with a port at a time. If an accept function is already registered for the port and you call the `AcceptMessage` function again, the current one is deregistered and is replaced by the one specified in the function call. You can set the `acceptProc` parameter to `nil` and call the `AcceptMessage` function to deregister the current accept function without installing another one.

If an exception arises while executing the accept function, the specified exception handler handles the exception, not the exception handler associated with the sending task. If an exception arises and no exception handler is installed, the sender of the message is notified of the error.

Messages are not guaranteed to be queued on a port in chronological order. This situation can occur when messages are sent to different message objects associated with the same port. Additionally, the message type controls which message is received, not the message's position within the queue.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Replying to a Message

All messages that are received from a port must be replied to. The reply consists of at least a status code of type `OSStatus`; it may contain other data as well. Accept functions may implicitly reply to a message—the operating system generates a reply based on the return value from the accept function. Other tasks that receive messages must explicitly reply to them. The following sections describe functions you can use to explicitly reply to messages.

ReplyToMessage

Replies to a message.

```
OSStatus ReplyToMessage(
                    MessageID theMessage,
                    OSStatus status,
                    ConstLogicalAddress replyBuffer,
                    ByteCount replyBufferSize);
```

`theMessage`	The message ID of the message you want to reply to.
`status`	The status you wish to indicate to the sender. If the message was sent by calling `SendMessageSync` (page 9-35), the status is delivered as its return value. If the message was sent by calling `SendMessageAsync` (page 9-37), the status is delivered by calling `DeliverKernelNotification` (…).
`replyBuffer`	The logical address of the reply buffer, or `nil`. A `nil` value indicates that there is no reply data.
`replyBufferSize`	The size of the reply buffer.
function result	To Be Provided

DISCUSSION

The `ReplyToMessage` function replies to the message specified by its message ID. Delivery of the status value completes the send-receive-reply transaction. You may supply additional information in a reply buffer and specify its address and size. If the sender specified a reply buffer, you can specify its address as the location of your reply buffer; otherwise you can specify the location of your reply buffer. If the reply buffer size is greater than the value specified by the sender, the reply data is truncated to fit in the smaller buffer.

If the sender specified a non-`nil` reply buffer, using that buffer avoids the cost of copying the reply data from the receiver to the sender. The address of sender's reply buffer is in the message control block. For information about the message control block, see "Message Definition" (page 9-6).

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers.

SEE ALSO

The `ReplyToMessageAndReceive` function (page 9-49) can also be used to reply to a message.

ReplyToMessageAndReceive

Replies to a message and receives another message synchronously.

```
OSStatus ReplyToMessageAndReceive(
                    MessageID theMessage,
                    OSStatus status,
                    ConstLogicalAddress replyBuffer,
                    ByteCount replyBufferSize,
                    PortID port,
                    MessageType theTypes,
                    MessageControlBlock *theControlBlock,
                    LogicalAddress receiveBuffer,
                    ByteCount receiveBufferSize,
                    ReceiveOptions options,
                    Duration timeout);
```

`theMessage`	The message ID of the message you want to reply to.

`status`	The status you wish to indicate to the sender. If the message was sent by calling `SendMessageSync` (page 9-35), the status is delivered as its return value. If the message was sent by calling `SendMessageAsync` (page 9-37), the status is delivered by calling `DeliverKernelNotification` (…).
`replyBuffer`	The logical address of the reply buffer, or nil. A nil value indicates that there is no reply data.
`replyBufferSize`	The size of the reply buffer.
`port`	The port from which you wish to receive a message.
`theTypes`	The message types of the messages you are willing to receive. For information about message types, see “Message Types” (page 9-4).
`theControlBlock`	A pointer to a message control block that receives enough information to retrieve the message.
`receiveBuffer`	The buffer to receive the message contents. If the message was sent by value, the buffer receives the message contents. If the message was sent by reference, the buffer is not modified.
`receiveBufferSize`	The size of the buffer you allow for the message content data.
`options`	The receive options. For information about receive options, see “Receive Options” (page 9-18).
`timeout`	A timeout value that specifies the maximum amout of time to wait, specified as a type `Duration` (page(To Be Provided)), to receive a message of the appropriate type.

DISCUSSION

The `ReplyToMessageAndReceive` function is equivalent to calling `ReplyToMessage` (page 9-48) followed immediately by calling `ReceiveMessageSync` (page 9-41).

The `ReplyToMessageAndReceive` function replies to the message specified by its message ID, which completes a send-receive-reply transaction, and initiates a new receive operation on the specified port. If you specify `kInvalidID` for the message ID, the reply is skipped and the receive begins immediately.

When replying to a message, you may supply additional information in a reply buffer and specify its address and size. If the sender specified a reply buffer, you can specify its address as the location of your reply buffer; otherwise, you can specify the location of your reply buffer. If the reply buffer size is greater than the value specified by the sender, the reply data is truncated to fit in the smaller buffer.

The `ReplyToMessageAndReceive` function then attempts to receive a message that matches the specified message type from the specified port. The task that calls `ReplyToMessageAndReceive` is blocked until either the appropriate message is received or a timeout occurs. If a timeout occurs, the receive request is canceled and the task becomes unblocked. You can specify a value of `kDurationForever` in the `timeout` parameter to prevent the receive request from being canceled. You can specify a value of `kDurationImmediate` if you want to cancel the receive request immediately if no message of the appropriate type is waiting.

If the sender specified a non-`nil` reply buffer, using that buffer avoids the cost of copying the reply data from the receiver to the sender. The address of sender's reply buffer is in the message control block. For information about the message control block, see "Message Definition" (page 9-6).

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

You can use the `ReplyToMessage` function (page 9-48) to reply to a message without receiving another one.

Application-Defined Function

MyMessageAcceptProc

Defines an accept function.

```
typedef OSStatus MyMessageAcceptProc(
                    const MessageControlBlock *message,
                    void *acceptRefcon);
```

`message`	A pointer to a message control block. For information about message control blocks, see “Message Definition” (page 9-6).
`acceptRefcon`	A pointer to the accept function’s reference constant.
function result	To Be Provided

DISCUSSION

An accept function defines the function to execute when a message with the appropriate message type arrives at the port to which the accept function is registered. Each time an accept function executes, the `message` parameter contains the pointer to the message control block for the received message and the `acceptRefcon` parameter contains a pointer to the accept function’s reference constant.

The accept function executes in supervisor mode, therefore, all code it executes and memory it accesses must be accessible from supervisor mode.

An accept function can explicitly call the `ReplyToMessage` function (page 9-48) to reply to the message received, or it may allow a reply to be generated by the operating system. These implicit replies are generated by returning an `OSStatus` value from the function with any status other than `kernelIncompleteErr`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For information about registering an accept function, see “AcceptMessage” (page 9-45).

For information about the definition of an accept function, see “Accept Function” (page 9-19).

C H A P T E R 1 0

Timing Services Reference

Contents

This chapter documents the constants, data types, and functions that you use to implement timers that are for your program's internal use in timing hardware or software events.

If your program manipulates times that are displayed to the user, you should use the data types and functions describe in the chapter To Be Provided.

You use the functions described in this chapter to

- obtain the current time
- determine how much processor time a task has consumed
- set and clear a synchronous timer
- set and clear an asynchronous timer
- set and clear an interrupt timer
- convert time units
- perform arithmetic operations with time units

Constants and Data Types

This section describes the constants and data types you use to represent time when using the microkernel's timing services.

Duration Enumeration

You can use the constant names provided by the duration enumeration to specify values of type duration. You use this type to specify the `delayDuration` parameter to the `DelayFor` function (page 10-12) or to specify a time-out value required by any System 7 function or data structure field of type duration.

You can combine these enumerated values into expressions to specify a duration that cannot be expressed by a single symbolic constant. For example, you can specify a duration of 2 hours and 12 minutes as

```
2*kDurationHour + 12*kDurationMinute
```

Possible duration units are given by the following enumeration:

```
typedef long Duration;

enum {
    kDurationMicrosecond    = -1L,         // Microseconds are negative
    kDurationMillisecond    = 1L,          // Milliseconds are positive
    kDurationSecond         = 1000L,       // 1000 * durationMillisecond
    kDurationMinute         = 60000L,      // 60 * durationSecond,
    kDurationHour           = 3600000L,    // 60 * durationMinute,
    kDurationDay            = 86400000L,   // 24 * durationHour,
    kDurationNoWait         = 0L,          // don't block
    kDurationForever        = 0x7FFFFFFF   // no time limit
};
```

Enumerator descriptions

`kDurationMicrosecond`	Wait one microsecond.
`kDurationMillisecond`	Wait one millisecond.
`kDurationSecond`	Wait one second.
`kDurationMinute`	Wait one minute.
`kDurationHour`	Wait one hour.
`kDurationDay`	Wait one day.
`kDurationNoWait`	Do not wait: schedule for execution immediately.
`kDurationForever`	Wait indefinitely.

The Absolute Time Data Type

Both the rate at which timer hardware is clocked and the representation of time in the time base register vary with each processor family. Absolute time is a means of representing time that makes these differences transparent to your application. In addition, the use of absolute time permits you to specify very small amounts of time and supports reliable drift-free timing services.

At system startup, the microkernel sets the absolute time to zero and continually increments the value throughout the life of the system. While absolute time is linear—one unit of absolute time is equal to another unit of absolute time for the boot of a machine, absolute time units are not guaranteed to be equal between two tasks running over a network. You can determine the rate at which absolute time is incremented by calling the `GetBaseTimeInfo` function (page 10-23).

Most of the timing functions described in this chapter take a parameter that you must specify in absolute time units. An absolute time value is defined by the absolute time data type.

```
typedef     UnsignedWide    AbsoluteTime;
```

The timing services shared library provides time conversion routines (page 10-25) that you can use to convert times expressed in conventional time units (nanoseconds, microseconds, milliseconds) into absolute time units. Because absolute time is stored as a 64-bit value and the C language does not include arithmetic functions that can manipulate integers of this size, the timing services library also provides arithmetic routines (page 10-33) that you can use to add and subtract absolute times. If you need to perform operations that are not supplied by this library, you should use the functions included in the `Math64` library.

Absolute time units are roughly of the same magnitude as nanoseconds.

The Nanoseconds Data Type

A time value in nanoseconds is defined as an unsigned 64-bit integer of type nanoseconds.

```
typedef     UnsignedWide    Nanoseconds;
```

Most timing services functions take parameters that you must specify using absolute time values (page 10-4). The timing services shared library provides time conversion routines (page 10-25) that you can use to convert nanoseconds into absolute time units.

Because a nanosecond value is stored as a 64-bit value and the C language does not include arithmetic functions that can manipulate integers of this size, the timing services library also provides arithmetic routines (page 10-33) that you can use to perform arithmetic operations on nanosecond values and other time unit values.

The Duration Data Type

A positive value of type duration designates the specified number of milliseconds; a negative value of type duration designates the specified number of microseconds. For example, a duration value of 100 is 100 milliseconds and a value of –100 is 100 microseconds.

If you are calling System 7 functions that require input in the form of microseconds or milliseconds, you must supply a 32-bit value of type duration. The `DelayFor` function (page 10-12) also requires an input parameter of type duration.

```
typedef long Duration;
```

You can use values of type duration to express time-out values that are as small as one microsecond or as large as 24 days. When setting and interpreting duration values, keep in mind that you can express these in two different ways. For example, 1000 and –1000000 both represent exactly one second.Two representations that have equal value are interchangeable; neither is preferred nor is inherently more accurate.

You can express duration as a numeric value or you can use symbolic constants defined by the duration enumeration (page 10-3). You use timeout or duration values for synchronous operations. Two duration values have special meaning: The value 0 (`kDurationNoWait`) means that you do not want to wait for the operation. If it cannot be executed immediately, it should not be done at all. The value `0x7FFFFFFF` (`kDurationForever`) specifies that a task can block indefinitely.

The timing services function `UpTime` returns the current time in absolute time. You can use the functions provided in the timing services shared library (page 10-25) to convert differences between absolute times into values of type duration. The timing services library also provides arithmetic routines that you can use to add values of type duration to absolute time values.

Timer ID

When you create a new timer using the `SetTimer` function (page 10-14) or the `SetInterruptTimer` (page 10-16) function, the microkernel returns a timer ID that identifies that timer.

```
typedef Ref TimerID;
```

You use this ID value to identify the timer when calling the `CancelTimer` function (page 10-22) or the `ResetTimer` function (page 10-18). The ID of a timer becomes invalid when the timer expires or when you cancel the timer by calling the `CancelTimer` function.

Timing Functions

You use the functions described in this chapter to obtain the current time, to determine how much time a task has consumed, to set and clear timers, and to manipulate time units.

Obtaining the Current Absolute Time

You use the two functions described in this section to obtain the current time. The time returned by the `UpTime` function returns the time (in absolute time units) since the machine was booted. The time returned by the `TaskCPUTime` function returns the time (in absolute time units) since the calling task began execution.

UpTime

Returns the current absolute time value.

```
AbsoluteTime UpTime (void);
```

function result The current absolute time value (page 10-4) maintained by the microkernel.

DISCUSSION

You can use the `UpTime` function to return the current absolute time value maintained by the microkernel. The timing services use this absolute time value as their reference to the current time.

You typically use the `UpTime` function as part of any timing operation that you want to execute periodically without incurring any drift in measuring elapsed time. The following example shows how you schedule an operation to execute every second:

```
OSStatus DriftFreeWorker (void * work)
{
    AbsoluteTime nextWorkTime;
    &nextWorkTime = UpTime ;
    do
        {DoTheWork (work)
        nextWorkTime = AddDurationToAbsolute(kDurationSecond,
                                    &NextWorkTime);
        DelayUntil (&nextWorkTime);
        } while (True);
}
```

You also need to use the current time when specifying the absolute expiration time of an asynchronous timer created by the `SetTimer` function or by the `SetInterruptTimer` function (page 10-16).

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

SEE ALSO

You can use the timing conversion functions beginning on page 10-25 to convert between `AbsoluteTime`, `Nanoseconds`, and `Duration` values.

You can use the absolute time arithmetic functions beginning on page 10-33 to add and subtract various combinations of absolute time, nanoseconds, and duration values and obtain the results in absolute time.

You can use the `AbsoluteDeltaToNanoseconds` function (page 10-25) to obtain the time that elapsed between two absolut time values and represent the result in terms of nanoseconds. You can also use the `AbsoluteDeltaToDuration` function (page 10-26) to obtain the time that elapsed between two absolute time values and represent the result as a duration value.

TaskCPUTime

Returns the absolute time since the calling task began to execute.

```
AbsoluteTime TaskCPUTime (void);
```

function result	The current absolute time (page 10-4) since the calling task began to execute.

DISCUSSION

The absolute time returned by the `TaskCPUTime` function does not include any time that was allotted to other concurrent tasks since the calling task began execution, but it does include time given to interrupts.

This function is useful for code profilers, which keep account of how time is shared between the system, other tasks, and the calling task.

EXECUTION ENVIRONMENT

Reentrant?	**Call at secondary interrupt level?**	**Call at hardware interrupt level?**
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

You can use the timing conversion functions described beginning on page 10-25 to convert between absolute time, nanoseconds, and duration values.

Synchronous Timers

Synchronous timers cause the calling task to stop executing for a specified period of time. The timing services provide synchronous timers you can use to specify time in both absolute and relative terms.

The `DelayUntil` function blocks the calling task until a specific absolute time is reached. This type of delay is useful to prevent drift when this function is invoked successively. The `DelayFor` function blocks the task for a period of time that is relative to when the `DelayFor` function is called.

DelayUntil

Blocks the calling task until an absolute time is reached.

```
OSStatus DelayUntil (const AbsoluteTime *expirationTime);
```

`expirationTime`
: On input, a reference to an absolute time (page 10-4) that specifies when the delayed task is eligible to resume execution. If the time you specify has already occurred, your task is eligible for execution immediately.

function result See "Result Codes" (page 10-42) for a list of the result codes that can be returned.

DISCUSSION

The `DelayUntil` function blocks the calling task until the absolute time specified by `expirationTime` is reached, after which the task is again made eligible for execution. The task might not be scheduled immediately. Use this function rather than the `DelayFor` function if you require drift-free timing service.

Each time you call the `DelayUntil` function, you specify the time when the task is to execute. What makes this future time *absolute* is that it is not relative to any previous invocation of the `DelayUntil` function. Any variations in the delay period are thus isolated to a single `DelayUntil` call and do not compound into long-term drift between the time when you expect the task to execute and the time it actually executes. The following example shows how you schedule a task to execute every second:

```
OSStatus DriftFreeWorker (void * work)
{
    AbsoluteTime nextWorkTime;
     &nextWorkTime = UpTime;
    do
        {DoTheWork (work)
        nextWorkTime = AddDurationToAbsolute(kDurationSecond,
                                    &nextWorkTime);
        DelayUntil (&nextWorkTime);
        } while (True);
}
```

In contrast, the `DelayFor` function (page 10-12) is used to delay the execution of a task for a period of time that is relative to when you call the `DelayFor` function.

SPECIAL CONSIDERATIONS

Your task can continue to receive software interrupts while it is delayed unless you disable software interrupts before calling the `DelayUntil` function or unless you call the `DelayUntil` function from within a software interrupt handler.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

When long-term drift is not an issue, you can use the `DelayFor` function (page 10-12) to delay your task for a relative period of time.

You can use the timing conversion functions described beginning on page 10-25 to convert between absolute time, nanoseconds, and duration values.

You can use the absolute time arithmetic functions described beginning on page 10-33 to add and subtract various combinations of absolute time, nanoseconds, and duration values and obtain the results in absolute time.

DelayFor

Blocks the calling task for a time period that is relative to the time when this function executes.

```
OSStatus DelayFor (Duration delayDuration);
```

`delayDuration`
: The duration of time the task is to be blocked. You can express this duration in units as small as microseconds and as large as days. See the description of the `Duration` type (page 10-6) for details. If you specify a delay duration time of 0, your task does not block but yields execution to any other tasks with the same priority level. If there are no other tasks at the same priority, your task continues to execute without delay.

function result See "Result Codes" (page 10-42) for a list of the result codes that can be returned.

DISCUSSION

The `DelayFor` function blocks the calling task for the time duration specified by the parameter `delayDuration`. This function allows the caller to delay for a time relative to when the service is called. You cannot achieve drift-free timing by using repeated calls to `DelayFor`, as you can using the `DelayUntil` function (page 10-10).

The problem with using a relative time value in situations that require the timing between tasks to occur at precise intervals, is that any timing latency incurred by interrupts, page faults, or the execution of higher priority tasks can slightly offset the time at which the blocked task resumes execution. For example, if you call the `DelayFor` function to delay your task for 1000 microseconds, the task might not be executed until 1002 microseconds from when the function was called. While such small variations may not be critical by themselves, when a task is repeatedly blocked and executed using the `DelayFor` function, slight differences in each individual delay period can compound into large drifts over the long term.

SPECIAL CONSIDERATIONS

You can continue to receive software interrupts while your task is delayed unless you disable software interrupts prior to calling the `DelayFor` function, or unless you call the `DelayFor` function from within the software interrupt handler.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

You can use the `DelayUntil` function (page 10-10) to delay your task until an absolute time has been reached. This function is useful to prevent long-term drift when calling the delaying function successively.

You can use the timing conversion functions described beginning on page 10-25 to convert between absolute time, nanoseconds, and duration values.

You can use the absolute time arithmetic functions described beginning on page 10-33 to add and subtract various combinations of absolute time, nanoseconds, and duration values and obtain the results in absolute time.

Asynchronous and Interrupt Timers

You can create a timer that executes asynchronously to the calling task and that uses a kernel notification structure to notify your task when it expires. This section describes the function you use to create an asynchronous timer that you can call from task level, as well as the functions you use to reset or cancel the timer.

This section also describes how you create an interrupt timer, which is used to execute a secondary interrupt handler. You can create an interrupt timer from a privileged task or from a secondary interrupt handler.

SetTimer

Creates an asynchronous timer to deliver a kernel notification upon expiration.

```
OSStatus SetTimer  (const AbsoluteTime *expirationTime,
                    const KernelNotification *notification,
                    TimerOptions options,
                    TimerID *theTimer);
```

`expirationTime`
: On input, a pointer to an absolute time (page 10-4) when the specified notification is to be delivered. If you specify a time that has already occurred, your timer may expire instantly or within a short period of time, so you should not depend on the exact behavior of the timer under these circumstances.

`notification`
: On input, a pointer to a kernel notification structure that specifies the manner in which your notification is to be delivered when the time specified by the `expirationTime` parameter is reached. Depending on how you set up the kernel notification structure, your notification can take the form of one or more event flags, a software interrupt, a kernel queue notification, or any combination of the three. A value of `nil` indicates that you do not want a notification to be delivered to the calling task.

`options`
: Timer options. Reserved for future use.

`theTimer`	On output, a pointer to the ID (page 10-6) of the timer created by the `SetTimer` function. You can use this ID to reset or cancel the timer prior to its expiration. The timer ID becomes invalid when you call a `CancelTimer` function (page 10-22) specifying this ID, or when the timer expires.
function result	The result code `memFullErr` indicates the system memory is exhausted and cannot create the timer. The result code `paramErr` means that the kernel notification structure you specified for the `notification` parameter is not found or is invalid. See "Result Codes" (page 10-42) for a list of other result codes that can be returned.

DISCUSSION

You can use the `SetTimer` function to create an asynchronous timer. After creating the timer, the calling task continues to execute. Upon expiration, the timer delivers a kernel notification, as described by the kernel notification structure referenced by the `notification` parameter.

If the kernel notification structure referenced by the `notification` parameter is set to deliver a notification using a software interrupt, the `SetTimer` function saves the current value of the program counter when the timer expires and passes it as the `p2` parameter to the software interrupt handler. If at this time the program counter is in an address space that is different from the address space of the calling task, the `SetTimer` function sets the `p2` parameter to 0, which means that the value has no meaning in the current address space.

If the kernel notification structure referenced by the `notification` parameter is set to use a kernel queue notification, the `SetTimer` function sets the `queueP3` field of the kernel notification structure to the current value of the program counter when the timer expires.

SPECIAL CONSIDERATIONS

When you call the `SetTimer` function, the microkernel makes a copy of the kernel notification structure referenced by the `notification` parameter. Thus you can dispose of that structure without affecting the operation of the timer or the delivery of the specified kernel notification.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

See the `AddAbsoluteToAbsolute` function (page 10-34) for information on how you would use the `UpTime` function (page 10-7) and `AddAbsoluteToAbsolute` to calculate a future expiration time that is relative to the present absolute time.

The `ResetTimer` function (page 10-18) allows you to change time and notification parameters for an existing timer.

The `CancelTimer` function (page 10-22) allows you to cancel an existing timer.

You can use the `DelayUntil` (page 10-10) and `DelayFor` (page 10-12) functions as synchronous timers to block a task for either absolute or relative time periods.

You can create an interrupt timer to run a secondary interrupt handler using the `SetInterruptTimer` function (page 10-16).

SetInterruptTimer

Creates an interrupt timer to run a secondary interrupt handler upon expiration.

```
OSStatus SetInterruptTimer (const AbsoluteTime *expirationTime,
                     SecondaryInterruptHandler2 handler,
                     void *p1,
                     TimerID *theTimer);
```

`expirationTime`
: On input, a reference to the absolute time (page 10-4) when the timer is to expire. If you specify a time that has already occurred, your timer might expire instantly or within a short period of time, so you should not depend on the exact behavior of the timer under these circumstances.

`handler`
: The address of a secondary interrupt handler routine that is to be run when the expiration time is reached. See "Interrupt Services Reference" for more information on secondary interrupt handlers.

`p1`
: On input, a pointer to the value that will be passed as the first parameter to the secondary interrupt handler when the timer expires. The second parameter, `p2`, passed to the secondary interrupt handler is set to the current value of the program counter when the timer expires.

`theTimer`
: On output, a pointer to the ID (page 10-6) of the timer created by the `SetInterruptTimer` function. You can use this ID to cancel the timer prior to its expiration. The timer ID becomes invalid when you call the `CancelTimer` function (page 10-22) or when the timer expires.

function result
: The result code `memFullErr` indicates that there is not enough memory allocated to accommodate your interrupt timer. The result code `paramErr` is returned if the address of the secondary interrupt handler routine you specified for the `handler` parameter is not found or is invalid. See "Result Codes" (page 10-42) for a list of other result codes that can be returned.

DISCUSSION

You use the `SetInterruptTimer` function to create an interrupt timer. When the timer expires, the secondary interrupt handler specified by the `handler` parameter is queued for execution. The secondary interrupt handler executes almost immediately after being queued. The value you specify for the `p1` parameter to the `SetInterruptTimer` function is passed to the secondary interrupt handler. The `p2` parameter passed to the handler contains the value stored in the program register (PC) when the timer expired.

Secondary interrupt handlers run with hardware interrupts enabled and task switching disabled, so you generally use the `SetInterruptTimer` function when coding device drivers and other system-level software that require timers with less latency than the task-level timers described in this section. Interrupt timers run asynchronously from the caller, which can be a secondary interrupt handler or a privileged task.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers.

SEE ALSO

Use the `AdjustInterrrupTimerSIHLimit` function (page 10-20) to adjust the number of concurrently running interrupt timers that are known to the system.

The `CancelTimer` function (page 10-22) allows you to cancel an existing timer.

You can use the `SetTimer` to create asynchronous timers for use by task-level software.

For more information on secondary interrupt handlers, see "Interrupt Services Reference."

ResetTimer

Resets an existing asynchronous timer with a new expiration time and notification.

```
OSStatus ResetTimer (TimerID theTimer,
                     const AbsoluteTime *expirationTime,
                     const KernelNotification *notification,
                     TimerOptions options);
```

`theTimer`	The ID (page 10-6) of the timer to be reset. The timer ID is returned by the `SetTimer` function (page 10-14).
`expirationTime`	On input, a pointer to an absolute time value (page 10-4) that specifies the new time when the specified `notification` is to be delivered. If you specify a value of `nil`, the previously set expiration time does not change. If you specify an absolute time that has already occurred, your timer might expire instantly or within a short period of time.
`notification`	On input, a pointer to a kernel notification structure that specifies how your notification is to be delivered when the absolute time specified by the `expirationTime` parameter is reached. Depending on how you set up the kernel notification structure, your notification can take the form of one or more event flags, a software interrupt, a kernel queue notification operation, or any combination of the three. A value of `nil` indicates that the notification is to be delivered as you specified the last time you set the timer.
`options`	Timer options. Reserved for future use.
function result	The result code `kernelIDerr` indicates that the timer has already expired and the timer reset did not take place. The result code `paramErr` is returned if the kernel notification structure you specified for the `notification` parameter is not found or is invalid. See "Result Codes" (page 10-42) for a list of other result codes that can be returned.

DISCUSSION

You can use the `ResetTimer` function to modify the characteristics of an asynchronous timer created by the `SetTimer` function.You can call this function any number of times to reset a timer. All other usage information is the same as described for the `SetTimer` function.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

The `SetTimer` function (page 10-14) allows you to create a new asynchronous timer.

The `CancelTimer` function (page 10-22) allows you to cancel an existing timer.

You can use the `DelayUntil` (page 10-10) and `DelayFor` (page 10-12) functions as synchronous timers to block a task for either absolute or relative time periods.

AdjustInterruptTimerSIHLimit

Adjusts the number of interrupt timers for which the system must allocate resources.

```
extern OSStatus AdjustInterruptTimerSIHLimit(SInt32 amount,
      ItemCount *newLimit);
```

`amount`	The number of interrupt timers that you are adding or subtracting from the current total.
`newLimit`	A pointer. On return, it references the number of interrupt timers for which resources are currently allocated.
function result	The result code `memFullErr` indicates that there is not enough memory available to allocate for another interrupt timer. See “Result Codes” (page 10-42) for a list of other result codes that can be returned

DISCUSSION

Interrupt timers consume microkernel resources from the time they are set until the time they complete execution. It is your responsibility to inform the microkernel of the amount of resources you will need by calling the `AdjustInterruptTimerSIHLimit` function. You need to provide this information to the microkernel whether you set an interrupt timer from task level or from secondary interrupt level.

You need to call the `AdjustInterruptSIHLimit` function from task level during your software's initialization. To increase the limit, specify the number of simultaneously active timers you are adding. If you set an interrupt timer and don't set any additional timers until the first timer has executed, you need only increment the limit by one. You need to call the function again to decrement the limit when your software is terminating.

Adjusting the limit of interrupt timers does not guarantee your exclusive use of the new allocated memory. If other concurrently running software creates interrupt timers without adjusting the limit, it might enjoy use of the memory you allocated for your own use.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by secondary interrupt handlers or by hardware interrupt handlers.

SEE ALSO

The `CancelTimer` function (page 10-22) allows you to cancel an existing timer.

You can use the `SetTimer` function to create asynchronous timers for use by task-level software.

For more information on secondary interrupt handlers, see "Interrupt Services Reference."

CancelTimer

Cancels a previously created timer.

```
OSStatus CancelTimer (TimerID theTimer,
                      AbsoluteTime *timeRemaining);
```

`theTimer`	The ID (page 10-6) of the asynchronous or interrupt timer to be canceled. This value is returned by either the function you used to create the timer the `SetTimer` function(page 10-14) or the `SetInterruptTimer` function (page 10-16).
`timeRemaining`	On output, a pointer to an absolute time value (page 10-4) indicating the time remaining before the timer would have expired. This value is the difference between the expiration time set for the timer and the absolute time when the timer was cancelled by the `CancelTimer` function.
function result	The result code `kernelIDerr` indicates that the timer has either expired or has already been canceled. See “Result Codes” (page 10-42) for a list of other result codes that can be returned.

DISCUSSION

You use the `CancelTimer` function to cancel an outstanding asynchronous timer or interrupt timer. Any kernel notification or secondary interrupt handler designated to run when the timer expires does not execute if the `CancelTimer` operation occurs before the expiration time of the timer.

SPECIAL CONSIDERATIONS

When you attempt to cancel an asynchronous timer, a race condition begins between your cancellation request and the expiration time of the timer. For example, you might test to see whether the timer has expired and if it has not, you cancel the timer. However, the timer might actually expire and cause a handler to execute between the time you perform the test and the time you cancel the timer.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers.

SEE ALSO

You can use the `SetTimer` function (page 10-14) to create an asynchronous timer to deliver a kernel notification upon expiration.

You can change the expiration time of the timer created by the `SetTimer` function or the kernel notification to be delivered by calling the `ResetTimer` function (page 10-18).

You can call the `SetInterruptTimer` function (page 10-16) to create interrupt timers to run secondary interrupt handler routines upon expiration.

Getting Time Base Information

GetTimeBaseInfo

Returns the rate at which the absolute time is incremented.

```
void GetTimeBaseInfo (
            UInt32 *theMinAbsoluteTimeDelta,
            UInt32 *theAbsoluteTimeToNanosecondNumerator,
            UInt32 *theAbsoluteTimeToNanosecondDenominator,
            UInt32 *theProcessorToAbsoluteTimeNumerator,
            UInt32 *theProcessorToAbsoluteTimeDenominator);
```

`theMinAbsoluteTimeDelta`
: On output, a pointer to a value that indicates the minimum number of absolute time units that can change at any given time. For example, if the Power Macintosh hardware increments the time in absolute time units of 128, then the value returned in the `minAbsoluteTimeDelta` parameter would be 128.

`theAbsoluteTimeToNanosecondNumerator`
: On output, a pointer to a returned value that represents the numerator portion of the fraction used to convert an absolute time value into nanoseconds.

`theAbsoluteTimeToNanosecondDenominator`
: On output, a pointer to a value that represents the denominator portion of the fraction used to convert an absolute time value into nanoseconds.

`theProcessorToAbsoluteTimeNumerator`
: On output, a pointer to a value that is part of the information required to convert a time base register value to an absolute time value.

`theProcessorToAbsoluteTimeDenominator`
: On output, a pointer to a value that is part of the information required to convert a time base register value to an absolute time value.

function result See “Result Codes” (page 10-42) for a list of the result codes that can be returned.

DISCUSSSION

Most of the values returned by the `GetTimeBaseInfo` function are for use by the system software and the functions described under “Timing Conversion Functions,” beginning on page 10-25.

The one value returned by the `GetTimeBaseInfo` function that is of direct use to you is that returned in the parameter `theMinAbsoluteTimeDelta`. This value specifies the minimum number of absolute time units by which the time is incremented. In effect, this represents the limit of accuracy in measuring times.

The remaining values returned by the `GetTimeBaseInfo` function are used to compute the value referenced by the parameter `theMinAbsoluteTimeDelta` and to allow the time conversion functions to convert a time base register value to nanoseconds.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers.

SEE ALSO

See the timing conversion functions described beginning on page 10-25 for a list of routines that you can use to convert between various units of time.

Timing Conversion Functions

You use the conversion routines described in this section to convert between nanoseconds, durations, absolute time, and ticks.

AbsoluteToNanoseconds

Converts absolute time to nanoseconds.

```
Nanoseconds AbsoluteToNanoseconds (AbsoluteTime theAbsoluteTime);
```

`theAbsoluteTime`	The absolute time (page 10-4) value to be converted into nanoseconds (page 10-5).
function result	The value of the specified absolute time converted into nanoseconds. The margin of error due to rounding is .5 nanoseconds.

DISCUSSION

You may find it necessary to convert an absolute time value to nanoseconds when using System 7 and earlier Time Manager functions that do not accept absolute time values as input.

SPECIAL CONSIDERATIONS

The conversion from absolute time to nanoseconds can result in rounding errors, which are typically limited to not more than .5 nanoseconds.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

SEE ALSO

You can obtain the current absolute time by calling the `UpTime` function (page 10-7).

You use the `NanosecondsToAbsolute` function (page 10-28) to convert from nanoseconds to an absolute time value.

AbsoluteToDuration

Converts absolute time to a duration value.

```
Duration AbsoluteToDuration (AbsoluteTime theAbsoluteTime);
```

`theAbsoluteTime`	The absolute time value (page 10-4) to be converted to a duration value (page 10-6).
function result	The duration value to which the absolute time value is converted. A negative number designates microseconds; a positive number designates milliseconds. If an absolute time value is too large to be expressed in units of microseconds, the

result is rounded up into the less-precise units of milliseconds. If the result is too large to be expressed in milliseconds then the value of `7FFFFFFE` is returned to indicate an overflow condition.

DISCUSSION

You might find it necessary to convert from an absolute time value to a duration value prior to calling the `DelayFor` function (page 10-12), or when using System 7 and earlier Time Manager functions that require time to be expressed in terms of microseconds or milliseconds.

SPECIAL CONSIDERATIONS

The conversion from absolute time to a duration value can result in rounding errors, which are typically limited to not more than .5 microseconds.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

SEE ALSO

You can obtain the current absolute time by calling the `UpTime` function (page 10-7).

Use the `DurationToAbsolute` function (page 10-29) to convert a duration value to an absolute time value.

AbsoluteToTicks

Returns the number of ticks for the specified absolute time value.

```
extern Ticks AbsoluteToTicks(AbsoluteTime theAbsoluteTime);
```

`theAbsoluteTime`The absolute time value to be converted to ticks.

function result The ticks to which the absolute time value has been converted. One tick equals 1/60 of a second.

DISCUSSION

Some existing software uses ticks to measure time. If you are writing new code that needs to interface with such software, you can use this routine to convert absolute time to ticks.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

SEE ALSO

Use the `TicksToAbsolute` function (page 10-33) to convert from ticks to absolute time value.

NanosecondsToAbsolute

Converts nanoseconds into absolute time.

```
AbsoluteTime NanosecondsToAbsolute (Nanoseconds theNanoseconds);
```

`theNanoseconds`
The nanoseconds (page 10-5) value to be converted to an absolute time (page 10-4) value.

function result The absolute time value to which the nanoseconds are converted.

DISCUSSION

Most timing services functions require that you specify expiration times in absolute time. To use these functions, you must convert any values that are expressed in your application as nanoseconds into absolute time.

SPECIAL CONSIDERATIONS

The conversion from nanoseconds into absolute time can result in rounding errors, which are typically limited to not more than .5 of an absolute time unit.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

SEE ALSO

Use the `AbsoluteToNanoseconds` function (page 10-25) to convert from absolute time to nanoseconds.

Use the absolute time arithmetic functions, described beginning on page 10-33, to obtain absolute time values by adding and subtracting various combinations of absolute time, nanoseconds, and duration values.

DurationToAbsolute

Converts a duration value into absolute time.

```
AbsoluteTime DurationToAbsolute (Duration theDuration);
```

`theDuration`	The 32-bit `Duration` (page 10-6) value to be converted to a 64-bit `AbsoluteTime` (page 10-4) value.
function result	The value of `theDuration` converted into `AbsoluteTime`. If you specify `kDurationForever` for the value to be converted, the function returns the largest signed 64-bit value: `0x7FFFFFFF:FFFFFFFF`.

DISCUSSION

You can use the `DurationToAbsolute` function to convert a `Duration` value expressed as microseconds or milliseconds into an `AbsoluteTime` value.

You must convert any values that are of type `Duration` into an `AbsoluteTime` value before using any of the Timing Services functions that require `AbsoluteTime` as input.

SPECIAL CONSIDERATIONS

The conversion from a `Duration` value into `AbsoluteTime` can result in rounding errors, which are typically limited to not more than one half a unit of absolute time.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

SEE ALSO

See the `AbsoluteToDuration` function (page 10-26) for information on how to convert from `AbsoluteTime` to a `Duration` value.

See the absolute time arithmetic functions, described beginning on page 10-33, for information on functions that allow you to obtain absolute time values by adding and subtracting various combinations of absolute time, nanoseconds, and duration values.

DurationToNanoseconds

Converts a duration value into nanoseconds.

```
Nanoseconds  DurationToNanoseconds (Duration theDuration);
```

`theDuration`	The 32-bit `Duration` value (page 10-6) to be converted to a 64-bit `Nanoseconds` value (page 10-5).

function result The value of `theDuration` converted into nanoseconds. If you specify `kDurationForever` for the value to be converted, the function returns the largest signed 64-bit value: `0x7FFFFFFF:FFFFFFFF`.

SPECIAL CONSIDERATIONS

The conversion from a time `Duration` value to `Nanoseconds` can result in rounding errors, which are typically limited to not more than .5 nanoseconds.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

SEE ALSO

See the `NanosecondsToDuration` function (described next) for information on how to convert from `Nanoseconds` to a `Duration` value.

See the absolute time arithmetic functions, described beginning on page 10-33, for information on functions that allow you to obtain absolute time values by adding and subtracting various combinations of absolute time, nanoseconds, and duration values.

NanosecondsToDuration

Converts nanoseconds into a duration value.

```
Duration NanosecondsToDuration (Nanoseconds theNanoseconds);
```

`theNanoseconds`
The 64-bit `Nanoseconds` value (page 10-5) to be converted to a 32-bit `Duration` value (page 10-6).

function result The 32-bit enumerated `Duration` value to which the `Nanoseconds` value is converted. Results in microseconds are returned as negative numbers; milliseconds are returned as positive numbers. If a 64-bit `Nanoseconds` value is too large to be expressed in units of `kDurationMicrosecond`, then the result is rounded up into less-precise units of `kDurationMillisecond`. If the result is too large to be expressed in 32-bit units of `kDurationMillisecond`, then an overflow value of `7FFFFFFE` (the largest non-infinite number that can be expressed in 32 bits) is returned.

DISCUSSION

You can use the `NanosecondsToDuration` function to convert a 64-bit `Nanoseconds` value into a 32-bit `Duration` value.

You may find it necessary to convert from an `Nanoseconds` value to a `Duration` value prior to calling the `DelayFor` function (page 10-12), or when using System 7.x and earlier Time Manager functions that require time to be expressed in terms of microseconds or milliseconds.

SPECIAL CONSIDERATIONS

The conversion from `Nanoseconds` to a time `Duration` value can result in rounding errors, which are typically limited to not more than 1 microsecond if the resulting duration is negative or 1 millisecond if the duration is positive.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

SEE ALSO

See the `DurationToNanoseconds` function (page 10-30) for information on how to convert a `Duration` value into `Nanoseconds`.

See the absolute time arithmetic functions, described beginning on page 10-33, for information on functions that allow you to obtain absolute time values by adding and subtracting various combinations of absolute time, nanoseconds, and duration values.

TicksToAbsolute

Converts ticks into absolute time.

```
extern AbsoluteTime TicksToAbsolute(Ticks theTicks);
```

`theTicks`	The number of ticks to be converted into an absolute time value. One tick equals 1/60 of a second.
function result	The absolute time value that is equivalent to the tick value specified for the parameter `theTicks`.

DISCUSSION

Some existing software uses ticks to measure time. If you are writing new code that needs to interface with such software, you can use this routine to convert ticks to absolute time.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

SEE ALSO

Use the `AbsoluteToTicks` function (page 10-27) to convert from absolute time to ticks.

AbsoluteTime Arithmetic Functions

Because absolute time and nanosecond values are stored as a 64-bit values and the C language does not include arithmetic functions that can manipulate integers of this size, the timing services library provides the following functions that you can use to perform additions and subtractions involving these 64-bit time units.

If you need additional arithmetic operations that manipulate 64-bit values, please use the functions declared in the file `Math64.h`.

AddAbsoluteToAbsolute

Returns the sum of two absolute time values.

```
AbsoluteTime AddAbsoluteToAbsolute (AbsoluteTime  theAbsoluteTime1,
                                    AbsoluteTime  theAbsoluteTime2);
```

`theAbsoluteTime1`	An absolute time value (page 10-4) to be added to the time specified by the parameter `theAbsoluteTime2`.
`theAbsoluteTime2`	An absolute time value (page 10-4) to be added to the time specified by the parameter `theAbsoluteTime1`.
function result	The absolute time resulting from the addition of the values specified by the parameters `AbsoluteTime1` and `theAbsoluteTime2`.

DISCUSSION

Use the `AddAbsoluteToAbsolute` function to add two absolute time values. Please note that one of these values must be obtained by finding the difference between two absolute time values. That is, you are finding the result of adding an absolute time delta to an absolute time.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

SEE ALSO

See the `SubAbsoluteFromAbsolute` function (page 10-35) for details on subtracting two `AbsoluteTime` values.

SubAbsoluteFromAbsolute

Returns the difference between two absolute time values.

```
AbsoluteTime SubAbsoluteFromAbsolute (
                    AbsoluteTime  theLeftAbsoluteTime,
                    AbsoluteTime  theRightAbsoluteTime);
```

`theLeftAbsoluteTime`
The absolute time value (page 10-4) from which the value specified by the parameter `theRightAbsoluteTime` is to be subtracted.

`theRightAbsoluteTime`
The absolute time value to be subtracted from the value specified by the parameter `theLeftAbsoluteTime`.

function result The difference in absolute time obtained from subtracting the value specified by the parameter `theRightAbsoluteTime` value from the value specified by the parameter `theLeftAbsoluteTime` value.

DISCUSSION

The difference between to absolute time values is called the absolute time delta. You can use the `SubAbsoluteFromAbsolute` function to calculate an absolute time delta. This function is useful when determining how much time has passed between two absolute time values, such you can obtain from invoking the `UpTime` function (page 10-7) at two different times.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

SEE ALSO

Use the `AddAbsoluteToAbsolute` function (page 10-34) to add two absolute time values.

If you need to convert the result returned by the `SubAbsoluteFromAbsolute` function to a value of type duration or nanoseconds, you should use the function `AbsoluteDeltaToNanoseconds` (page 10-40) or the function `AbsoluteDeltaToDuration` (page 10-41) instead. These functions find the difference between two absolute time values and return the difference in the units of your choice, thus combining two function calls into one.

AddNanosecondsToAbsolute

Returns the sum (in absolute time) of a nanosecond value and an absolute time value.

```
AbsoluteTime AddNanosecondsToAbsolute (Nanoseconds  theNanoseconds,
                   AbsoluteTime  theAbsoluteTime);
```

`theNanoseconds`
The value in nanoseconds (page 10-5) to be added to the absolute time specified by the parameter `theAbsoluteTime`.

`theAbsoluteTime`
The absolute time value (page 10-4) to be added to the value specified by the parameter `theNanoseconds`.

function result The sum (in absolute time) of the values specified by the parameters `theNanoseconds` and `theAbsoluteTime`.

DISCUSSION

You can use this function to compute some future time when you want a task to execute. You can then specify the result as the `expirationTime` parameter to the `DelayUntil` function or to one of the functions the defines an asynchronous timer.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

SEE ALSO

Use the `DelayUntil` function (page 10-10) to set up a synchronous timer.

Use the functions described beginning on page 10-14 to define asynchronous and interrupt timers.

AddDurationToAbsolute

Returns the sum (in absolute time) of a duration value and an absolute time value.

```
AbsoluteTime AddDurationToAbsolute (Duration theDuration,
                          AbsoluteTime  theAbsoluteTime);
```

`theDuration`	The duration value (page 10-6) to be added to the absolute time value specified by the parameter `theAbsoluteTime`.
`theAbsoluteTime`	The absolute time value (page 10-4) to be added to value specified by the parameter `theDuration`.
function result	The sum (in absolute time) of the values specified by the parameters `theDuration` and `theAbsoluteTime`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

SEE ALSO

Use the function `SubDurationFromAbsolute` (page 10-39) to subtract a duration from an absolute time value.

Use the timing conversion functions, described beginning on page 10-25, to convert absolute time values to nanoseconds, microseconds, milliseconds, or ticks.

SubNanosecondsFromAbsolute

Subtracts a value in nanoseconds from an absolute time value.

```
AbsoluteTime SubNanosecondsFromAbsolute (Nanoseconds theNanoseconds,
                                         AbsoluteTime theAbsoluteTime);
```

`theNanoseconds`
The nanosecond value (page 10-5) to be subtracted from the absolute time value specified by the parameter `theAbsoluteTime`.

`theAbsoluteTime`
The absolute time value (page 10-4) from which the value specified by the parameter `theNanoseconds` is to be subtracted.

function result The absolute time value obtained from subtracting the value specified by the parameter `theNanoseconds` value from the value specified by the parameter `theAbsoluteTime`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

SEE ALSO

Use the `AddNanosecondsToAbsolute` function (page 10-36) to add nanoseconds to an absolute time value.

Use the timing conversion functions, described beginning on page 10-25, to convert absolute time values to nanoseconds, microseconds, milliseconds, or ticks.

SubDurationFromAbsolute

Subtracts a duration value from an absolute time value.

```
AbsoluteTime SubDurationFromAbsolute (Duration theDuration,
                     AbsoluteTime  theAbsoluteTime);
```

`theDuration`	The duration value (page 10-6) to be subtracted from the value specified by the parameter `theAbsoluteTime`.
`theAbsoluteTime`	The absolute time value (page 10-4) from which the value specified by the parameter `theDuration` is to be subtracted.
function result	The absolute time value obtained by subtracting the value specified by the parameter `theDuration` from the value specified by the parameter `theAbsoluteTime`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

SEE ALSO

Use the `AddDurationToAbsolute` function (page 10-37) to add a duration value to an absolute time value.

Use the timing conversion functions, described beginning on page 10-25, to convert absolute time values to nanoseconds, microseconds, milliseconds, or ticks.

Timing Delta Functions

The functions described in this section find the difference between two absolute time values and convert the result to nanoseconds or duration values.

AbsoluteDeltaToNanoseconds

Subtracts one absolute time from another and returns the difference in nanoseconds.

```
Nanoseconds AbsoluteDeltaToNanoseconds (
                    AbsoluteTime theLeftAbsoluteTime,
                    AbsoluteTime theRightAbsoluteTime);
```

`theLeftAbsoluteTime`
: The absolute time value (page 10-4) from which the value specified by the parameter `theRightAbsoluteTime` is to be subtracted.

`theRightAbsoluteTime`
: The absolute time value to be subtracted from the value specified by the parameter `theLeftAbsoluteTime`

function result The difference (expressed in nanoseconds) resulting from subtracting the value specified by the parameter `theRightAbsoluteTime` from the value specified by the parameter `theLeftAbsoluteTime`. Because there is no concept of negative time in this system, if the difference is negative, the function returns 0.

DISCUSSION

Use this function to combine the action of the functions `SubAbsoluteFromAbsolute` and `AbsoluteToNanoseconds` into one.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

SEE ALSO

Use the `AbsoluteDeltaToDuration` function (page 10-41) to find the difference between two absolute time values and to convert it to microseconds or milliseconds.

AbsoluteDeltaToDuration

Subtracts one absolute time from another and returns the difference as a duration value.

```
Duration AbsoluteDeltaToDuration (AbsoluteTime  theLeftAbsoluteTime,
      AbsoluteTime theRightAbsoluteTime);
```

`theLeftAbsoluteTime`
The absolute time value (page 10-4) from which the value specified by the parameter `theRightAbsoluteTime` is to be subtracted.

`theRightAbsoluteTime`	The absolute time value to be subtracted from the value specified by the parameter `theLeftAbsoluteTime`
function result	The difference (expressed as a duration value) resulting from subtracting the value specified by the parameter `theRightAbsoluteTime` from the value specified by the parameter `theLeftAbsoluteTime`. Because there is no concept of negative time in this system, if the difference is negative, the function returns 0.

DISCUSSION

Use this function to combine the action of the functions `SubAbsoluteFromAbsolute` and `AbsoluteToDuration` into one.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

SEE ALSO

Use the `AbsoluteDeltaToNanoseconds` function (page 10-40) to find the difference between two absolute time values and to convert it to nanoseconds.

Result Codes

Many of the Timing Services return the result codes listed below.

`noErr`	0	No error
`paramErr`	–50	Error in parameter list
`memFullErr`	–108	Out of memory
`kernelIDerr`	–2419	You specified an ID that was never created or that has been destroyed

Glossary

absolute time delta A 64-bit value resulting from subtracting one absolute time value from another.

Debugger Services

Contents

Debugger Services Reference

This chapter describes data structures and functions that are declared in the file `DebuggerSupport.h`. You use these functions and structures to

- register a debugger
- receive an exception
- examine and modify the machine state
- examine and control task execution
- examine and modify memory
- obtain information about data breakpoint support
- set and clear data breakpoints
- unregister a debugger

The registered debugger can only handle exceptions that occur in normal tasks and software interrupts.

▲ **WARNING**
There is no special attempt to protect the system against debuggers. Please make judicious use of the calls provided to change the state of the processor and of memory. ▲

Constants and Data Types

The constants and data types described in this section describe the state of the currently executing task and kernel process, specify the precise nature of a data breakpoint operation, and provide state information associated with the current exception.

Kernel State Enumeration

The function `DSGetTaskState` (page 11-17) returns a kernel state enumeration value in the `kernelState` field of the kernel state structure (page 11-8). This value describes the execution state of a task or of its associated software interrupt if any.

```
enum {
    kInactiveState                   = '    ',
    kPageFaultState                  = 'PFLT',
    kMessageSendState                = 'MSND',
    kMessageReceiveState             = 'MRCV',
    kHeldState                       = 'HELD',
    kEventFlagState                  = 'EFLG',
    kTerminatingOtherProcessState    = 'OTRM',
    kTerminatingCurrentProcessState  = 'CTRM',
    kTimerDelayState                 = 'DLYQ',
    kKernelQueueState                = 'QBLK',
    kRunState                        = 'RUN ',
    kSemaphoreReadState              = 'MSRD',
    kSemaphoreWriteState             = 'MSWR',
    kTaskStartingState               = 'STRT',
    kTaskTerminatingState            = 'TERM'
};
```

Enumeration descriptions

`kInactiveState`	The software interrupt is inactive; it might be queued, but it is not yet eligible for execution.
`kPageFaultState`	The task or software interrupt is blocked and waiting for a page fault to be processed.
`kMessageSendState`	The task or software interrupt is blocked, waiting for a message to be sent.
`kMessageReceiveState`	The task or software interrupt is blocked, waiting for a message to be received.
`kHeldState`	The task or software interrupt is blocked as a result of a call to the function `DSHoldTasks` (page 11-18).
`kEventFlagState`	The task or software interrupt is blocked, waiting for an event flag.
`kTerminatingOtherProcessState`	The task or software interrupt is blocked, waiting for another process to terminate.

`kTerminatingCurrentProcessState`	The task or software interrupt is blocked, waiting for the current process to terminate.
`kTimerDelayState`	The task or software interrupt is blocked, waiting for a timer to expire.
`kKernelQueueState`	The task or software interrupt is blocked, waiting in a kernel queue.
`kRunState`	The task or software interrupt is running or it is eligible to be run.
`kSemaphoreReadState`	The task or software interrupt is blocked, waiting for a semaphore to be read.
`kSemaphoreWriteState`	The task or software interrupt is blocked, waiting for a semaphore to be written to.
`kTaskStartingState`	The task or software interrupt is blocked, waiting for a task to be started.
`kTaskTerminatingState`	The task or software interrupt is blocked, waiting for a task to be terminated.

Data Breakpoint Option Enumeration

You use the `options` parameter of the `DSSetDataBreakpointInformation` function (page 11-28) to clear a data breakpoint or to specify the type of operation (read or write) on which the data breakpoint should be set.

The data breakpoint option enumeration species the possible values you can specify for the `options` parameter.

```
typedef OptionBits DSDataBreakpointOptions;
enum {
    kDSBreakDIsable            = 0x00000000,
    kDSBreakOnReadAccess       = 0x00000001,
    KDSBreakOnWriteAccess      = 0x00000002
};
```

Enumeration descriptions

`kDSBreakDIsable` Set this bit to clear a data breakpoint. See the description of the `DSSetDataBreakpoint` function (page 11-28) to find out how you specify the address of the breakpoint to be cleared.

`kDSBreakOnReadAccess`
Set this bit to raise an exception when the address specified for the data breakpoint is accessed during a read operation.

Not all implementations are able to distinguish between read and write operations. As a result, the debugger might have to do additional work to determine whether the address reference that raised the exception did indeed occur during a read operation.

`KDSBreakOnWriteAccess`
Set this bit to raise an exception when the address specified for the data breakpoint is accessed during a write operation.

Not all implementations are able to distinguish between read and write operations. As a result, the debugger might have to do additional work to determine whether the address reference that raised the exception did indeed occur during a write operation.

Exception Structure

The `DSWaitForException` function (page 11-13) returns information about the current exception in an exception structure. Your debugger can use this information to display current state information to the user using memory display windows, register display windows, and so on.

The `DSExceptionRecord` data type defines an exception structure.

```
struct DSExceptionRecord {
    Task ID                 exceptedTaskID;
    KernelProcessID         exceptedKernelProcessID;
    AddressSpaceID          exceptedAddressSpaceID;
    MessageID               exceptionID;
    ExceptionInformation    exception;
};
```

Field descriptions

`exceptedTaskID`	The ID of the task that caused the exception. If the exception is generated by an interrupt handler, this field specifies the ID of the task that is currently executing and which was interrupted.
`exceptedKernelProcessID`	The ID of the process containing the task that caused the exception.
`exceptedAddressSpaceID`	The ID of the address space belonging to the task that caused the exception.
`exceptionID`	The ID of the exception.
`exception`	An exception information structure that describes the state of the processor at the time the exception occurred. The exception information structure is described in the chapter “Exception Handling Reference”

Task State Structure

The `DSGetTaskState` function (page 11-17) returns information in a task state structure about the execution state of a task.

The `DSTaskState` data type defines a task sate structure.

```
struct DSTaskState {
    DSKernelState   taskState;
    DSKernelState   swiState;
};
```

Field descriptions

`taskState`	A kernel state structure (page 11-8) describing the execution state for the task whose state is being queried.
`swiState`	A kernel state structure (page 11-8) describing the execution state for the software interrupt targeted at the task whose state is being queried.

Kernel State Structure

The `DSGetTaskState` function (page 11-17) returns a task state structure (page 11-7) that contains two kernel state structures: one of the structures describes the execution state of the task and the other describes the execution state of the software interrupt currently targeted at that task.

The `DSKernelState` data type defines a kernel state structure.

```
struct DSKernelState {
    SchedulerState      kernelState;
    LogicalAddress      PC;
    LogicalAddress      SP;
};
```

Field descriptions

`kernelState`	A value specified by the kernel state enumeration (page 11-3) that describes the execution state of a task or of a software interrupt.
`PC`	The value stored in the Program Counter register at the time that the `DSGetTaskState` function executes.
`SP`	The value stored in the Stack Pointer register at the time that the `DSGetTaskState` function executes.

The values returned in the `PC` and `SP` fields depend on the state of the task or its associated software interrupt. If the state is `kRunState` or `kInactiveState`, the contents of the Program Counter and Stack Pointer are 0. This means that the values are either impossible to determine or that if they could be determined, they would be without meaning. For example, if the task whose state you are querying is currently running, the contents of the PC would be continually changing.

If the task or software interrupt is blocked—that is, if it is in any state other than `kRunState` or `kInactiveState`, the values stored in the PC and SP registers are the values that were current when the function that caused the task or software interrupt to block was called.

Data Breakpoint Information Structure

The `DSGetDataBreakpointInformation` function (page 11-26) returns a pointer to a data breakpoint information structure that you can examine to determine whether the current microkernel implementation supports data breakpoints and, if it does, the maximum number of such breakpoints that it supports and the resolution (margin of error) for any given exception that is raised.

The `DSDataBreakpointInformation` data type defines a data breakpoint information structure.

```
struct DSDataBreakpointInformation {
    ItemCount   maxBreakpoints;
    ByteCount   breakpointResolution;
};
```

Field descriptions

`maxBreakpoints`	On output, the maximum number of breakpoints that this implementation of the microkernel supports. A value of 0 means that data breakpoints are not supported.
`breakpointResolution`	On output, the memory range within which the microkernel will raise a data breakpoint exception for a given address reference. See the discussion that follows for additional information.

DISCUSSION

In the worst case, for some breakpoint address *addr*, the range of addresses to which a data breakpoint applies is given by the following formula

```
[(addr & ~ (breakpointResolution -1))..
        (addr & ~ (breakpointResolution -1)) + breakpointResolution -1]
```

Depending on the implementation, the resolution might range from 8 bytes to 1 page. (Pages vary in size; use the `GetSystemInformation` function to determine the size of a page.) It is up to the debugger to do additional filtering to determine whether the exception that is raised corresponds to the data breakpoint that was set or whether it was raised simply because it occurred within the memory range specified by the `breakpointResolution` field.

Functions

You use the functions described in this section to register and unregister a debugger, and to ease the debugger's ability to handle exceptions, control task scheduling. change the state of the machine, and manage data breakpoints.

Registering and Unregistering a Debugger

You use the functions described in this section to register and unregister a debugger. In order for the microkernel to pass exceptions to the debugger and to provide services to the debuggers, the debugger must register itself. When you no longer want the debugger to receive exceptions, you can explicitly unregister it or you can allow the operating system to do it when the process associated with the debugger terminates. It is recommended that you unregister the debugger yourself.

DSRegisterDebugger

Registers the calling task as a debugger.

```
OSStatus DSRegisterDebugger (void);
```

function result If another debugger is already registered, the function returns the result `kernelIDErr`.

Once a task is registered as a debugger, any call made to the microkernel debugger services by a task not belonging to the registered debugger's task family also returns the result `kernelIDErr`.

DISCUSSION

The microkernel allows at most one debugger to register itself at any one time. The microkernel passes exceptions to the currently registered debugger and allows it access to debugger services.

The microkernel passes all exceptions to the registered debugger except for exceptions that occur

- in kernel tasks or in microkernel code running on behalf of user tasks
- in privileged tasks with hardware interrupts disabled
- during execution at secondary interrupt level
- while running message system accept functions
- in the 68K emulator
- in the debugger itself

Once you use this function to register a debugger, the microkernel associates the calling task and all other tasks in the task's family with the debugger. As a result, the microkernel does not pass any exception occurring in these tasks back to the debugger. To handle these exceptions, the debugger must install an exception handler using the `InstallExceptionHandler` function.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

Call the `DSUnregisterDebugger` function (page 11-12) to unregister the debugger.

After registering the debugger, call the `DSWaitForException` function (page 11-13) to receive an exception.

To install an exception handler for exceptions that occur in the debugger use the `InstallExceptionHandler` function described in "Exception Handling Reference."

For additional information about tasks and a task's family, see the chapter "Tasks, Processes, and Scheduling."

DSUnregisterDebugger

Unregisters a debugger.

```
OSStatus DSUnregisterDebugger (void);
```

function result If the caller is not a task belonging to the family of the registered debugger, the function returns the result `kernelIDErr`.

DISCUSSION

The microkernel unregisters the currently installed debugger automatically when the debugger terminates and it does this whether or not the debugger terminates normally. However, it is recommended that you call the `DSUnregisterDebugger` function explicitly to unregister the debugger. If you call this function and another debugger is available, the microkernel installs it immediately once your debugger has unregistered itself and, in that case, it is less likely that any exceptions are lost by the time the new debugger is installed.

If any tasks are blocked, waiting for exceptions to be processed, they are resumed when the debugger unregisters.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

Call the `DSRegisterDebugger` function (page 11-10) to register the debugger.

Handling Exceptions

You use the functions described in this section to obtain information about an exception and to resume execution after responding to the exception.

DSWaitForException

Waits a specified amount of time to receive an exception.

```
OSStatus DSWaitForException (DSExceptionRecord * exceptionRecord,
                    Duration timeLimit);
```

`exceptionRecord`	A pointer to an exception structure (page 11-6) that identifies the exception, the task, the process, and the memory space in which the exception occurred and that passes additional information about the state of the machine when the exception occurred.
`timeLimit`	An integer specifying the amount of time this function should wait to receive an exception. To specify the time limit value in milliseconds, use a positive number; to specify the time limit value in microseconds, use a negative number. You can specify `kDurationForever` to wait indefinitely.
function result	If the caller is not a task belonging to the family of the registered debugger, the function returns the result `kernelIDErr`.

DISCUSSION

The `DSWaitForException` function waits for an exception to occur. It is recommended that your debugger dedicate one task to receiving exceptions. This task can call the `DSWaitForException` function with the `timeLimit` parameter set to `kDurationForever`. When it receives an exception the waiting task can place it on a queue, from where another task can retrieve it and process it. In the meanwhile the receiving task can continue to receive exceptions.

The `exceptionRecord` parameter passes complete information about the exception that caused the debugger to be invoked and the state of the machine when the exception occurred. This information includes:

- The ID of the task that caused the exception
- The ID of the process containing the task that caused the exception
- The ID of the address space belonging to the task that caused the exception
- The ID of the exception
- The type of exception (memory, illegal instruction, and so on)
- The contents of the CPU's general purpose registers, floating point registers, and special registers when the exception occurred
- The type of memory access operation that caused the exception if the exception was a memory access type

SPECIAL CONSIDERATIONS

The exception `kTaskCreationException` is only seen by the debugger. When this exception is raised, the task is ready to execute, all libraries are loaded, and the Program Counter points to the task's entry point.

EXECUTION ENVIRONMENT

Reentrant?	**Call at secondary interrupt level?**	**Call at hardware interrupt level?**
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

After an exception has caused a task to be halted, you use the `DSResumeFromException` function (page 11-15) to resume execution of the task or to propagate the exception to an exception handler.

See the “Timing Services” chapter for a description of the duration data type.

DSResumeFromException

Resumes the execution of a task that has been halted in order to process an exception or propagates the exception to another exception handler.

```
OSStatus DSResumeFromException (MessageID exceptionID, OSStatus
                  exceptionReturnStatus);
```

`exceptionID` The ID of the exception that has been received using the `DSWaitForException` function.

`exceptionReturnStatus`
A value that indicates whether the microkernel should resume task execution or whether it should propagate the exception to the installed exception handler. Specify `noErr` to resume task execution. Specify any nonzero error code to propagate the exception.

This parameter is ignored for task creation exceptions.

function result To be provided.

DISCUSSION

The setting of the `exceptionReturnStatus` parameter allows you to determine whether the debugger handles an exception first or last—that is, before or after the installed exception handler.

- If you set the parameter `exceptionReturnStatus` to `noErr`, the microkernel assumes that the debugger has handled the exception and it resumes execution of the excepted task using whatever state information is current for that task. If the debugger has changed the state of the processor while the task was halted, the microkernel attempts to resume task execution with the new state setting.
- If you set the parameter `exceptionReturnStatus` to `kernelUnrecoverableErr`, the microkernel terminates the excepted task without running any task exception handler. The microkernel also terminates the process to which the task belongs if the task is the main task.

- If you set the `exceptionReturnStatus` parameter to a nonzero value other than `kernelUnrecoverableErr`), the microkernel passes the exception to the installed exception handler for that task, if any. If there is no installed handler or if the handler fails to resolve the exception, the microkernel presents the exception to the debugger a second time. This second instance of the exception has a different exception number than the first instance. If the debugger chooses to propagate the exception again, the microkernel terminates the excepted task. The microkernel also terminates the process to which the task belongs if the task is the main task.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

You use the `DSWaitForException` function (page 11-13) to receive an exception.

You use the `InstallExceptionHandler` function described in "Exception Handling Reference" to install an exception handler for a task.

Controlling Task Scheduling

You use the functions described in this section to obtain information about a task's execution, to hold a task, and to resume task execution after holding. Controlling task scheduling can help you debug synchronization problems.

DSGetTaskState

Returns information about the execution state of a task.

```
OSStatus DSGetTaskState (TaskID taskID, DSTaskState * state);
```

`taskID`	The ID of the task whose execution state you are querying.
`state`	A pointer to a task state structure (page 11-7) that describes the execution state for the task specified by the `taskID` parameter. On return, the microkernel places information about the task in this structure.
function result	If you specify an invalid task ID, this function returns the result code `kernelIDErr`.

DISCUSSION

The `state` parameter points to a task state structure that contains two kernel state structures, one that describes the execution state of the task and one that describes the execution state of the software interrupt currently targeted at that task. The possible combinations of execution states are as follows:

- The task is eligible for execution or is executing and there are no software interrupts pending for the task (`taskState.kernelState` is `kRunState`; `swiState.kernelState` is `kInactiveState`.)
- The task is eligible for execution and the software interrupt is either executing or it is eligible for execution (`taskState.kernelState` and `swiState.kernelState` are `kRunState`.)
- The task is blocked; the software interrupt is eligible for execution or executing (`taskState.kernelState` is one of the blocked values; `swiState.kernelState` is `kRunState`.
- Both the task and the software interrupt are blocked. In this case `taskState.kernelState` and `swiState.kernelState` both contain blocked values.

You must exercise caution when using this function to hold a privileged task. All privileged tasks belong to the microkernel process. Consequently, if you use this function to hold a privileged task and specify `kTaskFamily` or `kTaskKernelProcess` for the task's scope, you will cause all tasks belonging to the microkernel process to be held.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

You use the `DSHoldTasks` function (page 11-18) to make a task ineligible for execution.

You use the `DSReleaseTasks` function (page 11-19) to make a task eligible for execution.

DSHoldTasks

Makes a task or set of tasks ineligible for execution.

```
OSStatus DSHoldTasks (TaskID taskID, TaskRelationship scope);
```

`taskID`	The ID of the task that you want the microkernel to hold.
`scope`	A constant specifying the scope of the task you want held. This scope indicates whether you want the microkernel to hold the task only, the task and its children, the task family, or the process to which the task belongs.
function result	If you specify an invalid task ID, this function returns the result code `kernelIDErr`.

DISCUSSION

The microkernel holds the specified task or tasks until you call the `DSReleaseTasks` function.

If a task being held is already blocked, for example, because of a page fault, I/O operation, or message send, it remains ineligible for execution even after it is unblocked.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

You use the `DSReleaseTasks` function (page 11-19) to make a held task eligible for execution.

You use the `DSGetTaskState` function (page 11-17) to obtain information about the execution state of a task.

Possible values for a task's scope are given by the task scope enumeration in **XREF**.

DSReleaseTasks

Makes a task or tasks that have been held, eligible for execution.

```
OSStatus DSReleaseTasks (TaskID taskID, TaskRelationship scope);
```

`taskID`	The ID of the task that you want the microkernel to release.
`scope`	A constant specifying the scope of the task you want released. This scope indicates whether you want the microkernel to release the task only, the task and its children, the task family, or the task team.

function result If you specify an invalid task ID, this function returns the result code `kernelIDErr`.

DISCUSSION

The value that you specify for the `scope` parameter for this function does not have to be the same as that of the `scope` parameter for the `DSHoldTasks` function. For example, if you chose to hold a task's team, you can release the hierarchical components of the team using several calls to the `DSReleaseTasks` function, each call releasing another level.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

You use the `DSHoldTasks` function (page 11-18) to make a task ineligible for execution.

You use the `DSGetTaskState` function (page 11-17) to obtain information about the execution state of a task.

Possible values for a task's scope are given by the task scope enumeration in **XREF**.

Reading and Modifying Task Memory

You use the functions described in this section to read or modify task memory. Use the `DSCreateMemoryAccess` function to obtain permission to write to memory; use the `DSWriteMemory` function to write to memory, and use the

`DSDeleteMemoryAccess` function to inform the microkernel that you no longer need to write to a given area in memory.

DSReadMemory

Reads memory within the specified address space.

```
OSStatus DSReadMemory (AddressSpaceID addrSpaceID,
                       LogicalAddress srcAddr, void * dstDataPtr,
                       ByteCount numBytes);
```

`addrSpaceID`	The ID of the address space that contains the range of memory to be read.
`srcAddr`	The beginning address of the range of memory to be read.
`dstDataPtr`	A pointer to a buffer that you create. On return, the microkernel copies the contents of the memory being read into this buffer.
`numBytes`	An integer specifying the number of bytes you want to read.
function result	If the function succeeds it returns the result code `noErr. If` the value specified for the `addrSpaceID` parameter is invalid, it returns the result `kernelIDErr`. If there is a problem with reading from the address specified by the `srcAddr` parameter or writing to the address referenced by the `dstDataPtr` parameter, the function returns a memory system error.

DISCUSSION

You can obtain the address space belonging to a task either by examining the `exceptedAddressSpaceID` field of the exception structure (page 11-6) or by calling the `GetKernelProcessInformation` function for the task's process.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

SPECIAL CONSIDERATIONS

This function is reentrant and can be run in privileged mode. You can call this function from task level.

SEE ALSO

The `GetKernelProcessInformation` function is described in (**XREF**).

DSCreateMemoryAccess

Obtains permission to write to the specified range of memory.

```
OSStatus DSCreateMemoryAccess (AddressSpaceID addrSpaceID,
                        LogicalAddress dstAddr,
                        DSMemoryAccessID * memAccessID);
```

`addrSpaceID`	The ID of the address space to which you want to write.
`dstAddr`	The logical address to which you want to write.
`memAccessID`	A pointer to a memory access ID. On return, this points to an ID used to specify the write access right to a particular page in memory. You specify this value as the `memAccessID` parameter to the `DSWriteMemory` function.
function result	If the function succeeds it returns the result code `noErr. If` the value specified for the `addrSpaceID` parameter is invalid, it returns the result `kernelIDErr`. If there is a problem with accessing the address specified by the `dstDataPtr` parameter, the function returns a memory system error.

DISCUSSION

You must call the `DSCreateMemoryAcces` function before you can call the `DSWriteMemory` function. By explicitly obtaining the right to access memory, your debugger can make sure that modifications made to code in memory are coordinated with backing storage. Thus, for example, if the user modifies code using the memory display or if the debugger modifies code in the process of setting a breakpoint, these changes are not lost during paging activity nor written back to the file to which the code is mapped.

After the debugger obtains a memory access ID, the page containing the logical address specified by the `dstAddr` parameter becomes physically resident. This prevents subsequent reads or writes of the page from or to backing storage.

You can obtain the address space belonging to a task either be examining the `exceptedAddressSpaceID` field of the exception structure (page 11-6) or by calling the `GetKernelProcessInformation` function for the task's process.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

The `GetKernelProcessInformation` function is described in (**XREF**).

To delete a memory access ID use the `DSDeleteMemoryAccess` function (page 11-25).

To write to memory use the `DSWriteMemory` function (page 11-24).

DSWriteMemory

Modifies memory to which you have obtained an access right.

```
OSStatus DSWriteMemory (DSMemoryAccessID memAccessID,
                    LogicalAddress dstAddr,
                    void * srcDataPtr, BYteCount numBytes);
```

`memAccessID`	A memory access ID obtained using the `DSCreateMemoryAccess` function.
`dstAddr`	The beginning address where the data is to be written.
`srcDataPtr`	A pointer to the data that is to be written.
`numBytes`	An integer specifying the number of bytes to be written.
function result	If the function succeeds, it returns the result code `noErr. If` the value specified for the `memAccessID` parameter is invalid, it returns the result `kernelIDErr.` If there is a problem with accessing the address referenced by the `srcDataPtr` parameter or specified by the `dstAddr` parameter, the function returns a memory system error.

DISCUSSION

The range specified by

```
    dstAddr...dstAddr + numBytes - 1
```

must lie entirely within the page specified by the access right. Use the `GetSystemInformation` function to obtain the current page size. To modify memory that spans more than a single page, you need to perform multiple operations.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

You must call the `DSCreateMemoryAccess` function (page 11-22) to obtain permission to write to memory before you call the `DSWriteMemory` function.

When you are finished writing you should call the `DSDeleteMemoryAccess` function (page 11-25) to inform the microkernel that it can release the page that has been made resident using the `DSMemoryAccess` function.

DSDeleteMemoryAccess

Deletes an access right and resumes normal backing storage activity for a given page.

```
OSStatus DSDeleteMemoryAcces (DSMemoryAccessID memAcessID);
```

`memAccessID`	A value returned by the `DSCreateMemoryAcces` function.
function result	If the function succeeds, it returns the result code `noErr. If` the value specified for the `memAccessID` parameter is invalid, it returns the result `kernelIDErr.`

DISCUSSION

Before writing to memory, you call the `DSCreateMemoryAccess` function, which returns a memory access ID. Associated with that ID is a page that the microkernel makes physically resident in order to prevent normal backing storage activity for that page. When you are done writing, you should call the

DSDeleteMemoryAccess function to inform the microkernel that it is now safe to resume normal paging activity. Changes that you have made to memory up to the time that you call the DSDeleteMemoryAccess function are not written to backing storage. Consequently, the changes that you make never become permanent.

When the debugger unregisters itself or terminates, the microkernel deletes all outstanding memory access rights.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

You call the DSCreateMemoryAccess function (page 11-22) to obtain permission to write to memory.

Supporting Data Breakpoints

You use the functions described in this section to determine the type of support provided by the current processor and microkernel implementation for data breakpoints and to set and clear such breakpoints.

DSGetDataBreakpointInformation

Returns information about the current support provided for data breakpoints.

```
OSStatus DSGetDataBreakpointInformation (PBVersion version,
      DSDataBreakpointInformation * info);
```

`version`	The constant `kDataBreakpointInformationVersion`.
`info`	A pointer to a data breakpoint information structure (page 11-8).
function result	If the function succeeds, it returns the result `NoErr`. If the pointer specified for the `info` parameter is invalid, the function fails and returns the result `paramErr`.

DISCUSSION

The `DSGetDataBreakpointInformation` function fills in a data breakpoint information structure that you provide and that you can examine to determine whether data breakpoints are supported and, if they are, the extent of such support. One or more of the following conditions are possible:

- data breakpoints are not supported. The `maxBreakpoints` field of the data breakpoint information structure is set to 0 to indicate this.
- the memory exception raised by the data breakpoint might be within a certain range of the breakpoint address but might not be at the address you specify. For example, if you set a breakpoint at address X, an exception might be raised by an instruction attempting to access address x + 7. The possible range of memory is indicated by the `breakpointResolution` field of the data breakpoint information structure.
- the memory exception raised by the data breakpoint might return an instruction referencing the breakpoint address, but this address might lie in the wrong address space.

Before alerting the user that the data breakpoint has been reached, the debugger might have to do additional work to determine that it is the correct reference. This additional processing is described beginning on page 11-28.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

You call the `DSSetDataBreakpoint` function (page 11-28) to set or clear a data breakpoint.

DSSetDataBreakpoint

Sets or clears a data breakpoint.

```
OSStatus DSSetDataBreakpoint (AddressSpaceID addrSpaceID,
      LogicalAddress breakAddr, ByteCount numBytes,
      DSDataBreakpointOptions options);
```

`addrSpaceID`	The ID of the address space in which the data reference is made.
`breakAddr`	The logical address of the data being referenced.
`numBytes`	The size of the data that is being referenced. This value should not exceed the value returned in the `breakpointResolution` field of the data breakpoint information structure (page 11-8).
`options`	One of the values specified by the data breakpoint options enumeration (page 11-5).
function result	If the function succeeds, it returns the result `NoErr`. Otherwise, it returns one of the following result codes.

If data breakpoints are not supported by this particular implementation or cpu, the function returns the result `kernelUnsupportedErr`.

If you use this function to clear a data breakpoint when none is set, the function returns the result `kernelAlreadyFreeErr`.

If you attempt to disable a data breakpoint for which you specify an bad address or address space ID, the function returns the result `paramErr`.

If there is not enough memory available for the microkernel to allocate internal data structures used to manage the data breakpoint, the function returns the result `memFullErr`.

If you specify a value for the `breakAddr` parameter that is not allowed, the function returns the result `kernelAttributeErr`.

DISCUSSION

To clear a data breakpoint specify the location of the data reference that you have previously set a data breakpoint for, and then specify `kDSBreakDisable` for the `options` parameter.

To set a data breakpoint, specify the location and size of the data reference you are interested in, and then specify either `kDSBreakOnReadAccess` or `kDSBreakOnWriteAccess` for the `options` parameter. Not all implementations are able to distinguish between the two types of access, and it is possible that the debugger might have to do additional work to determine whether the memory reference that raised the exception did indeed occur as the result of the specified operation.

If the current implementation supports data breakpoints, the microkernel will raise an exception of the type `kDataBreakpointException` whenever it encounters an instruction referencing a memory location that is within a certain range of the address specified for the data breakpoint. In the worst case, for some breakpoint address *addr*, the range of addresses to which a data breakpoint applies is given by the following formula

```
[(addr & ~ (breakpointResolution -1))..
        (addr & ~ (breakpointResolution -1)) + breakpointResolution -1]
```

Before alerting the user that the breakpoint has been reached, the debugger must examine the exception information structure returned by the

`DSWaitForException` function and determine whether the correct reference has raised the current exception. Possible discrepancies might arise because the reference is in the wrong address space, because it is not the exact address sought, or because it did not occur as a result of the type of operation specified by the `options` parameter of the `DSSetDataBreakpoint` function.

If the correct address reference has been found, the debugger can present it to the user. If the correct address reference has not been found the debugger must do the following

3. Emulate the instruction currently referenced by the PC (Program Counter).

   This is the instruction whose memory access raised the data breakpoint exception.

   - If this was a load instruction, the debugger must use the `DSReadMemory` function (page 11-21) to emulate the read operation and then it must place the data read into the appropriate processor register value field of the exception information structure.
   - If this was a store instruction, the debugger must examine the instruction to determine from which register it must read the data. It can then examine the exception information structure to find the correct value and use the `DSCreateMemoryAccess` function (page 11-22) and the `DSWriteMemory` function (page 11-24) to write that value to memory.

4. Advance the program counter to the next instruction.
5. Use the `DSResumeFromException` function to resume task execution.

This procedure is the only way to resume task execution that does not result in the same (erroneous) exception being raised. It is especially important to use the `DSReadMemory` and `DSWriteMemory` functions to emulate the instruction that caused the break, because these functions are implemented in way that will not cause the debugger's memory access to trigger another data breakpoint exception.

SEE ALSO

The exception information structure is described in the chapter "Exception Handling Services."

Use the `DSWaitForException` function (page 11-13) to obtain information about an exception.

Use the `DSGetDataBreakpointInformation` function (page 11-26) to determine what support the current implementation provides for data breakpoints.

Use the `DSResumeFromException` function (page 11-15) to resume task execution.

The Patch Manager

Contents

This chapter describes the Patch Manager and explains data-driven patching, a new patching model that you should use if you are writing patching code that is meant to run in Mac OS 8 or in any subsequent Mac OS release.

A **patch** is a piece of code that intercepts the transaction between a client and a service. You can use a patch to monitor the use of this service or to modify or replace the service. The Patch Manager is a shared library containing routines that return information about existing patches and that enable and disable patches. You should read this chapter if you are writing a program that must monitor, modify, or replace a routine residing in another fragment and if you cannot find any means to do so other than by patching that routine.

The Patch Manager is a new component of the Macintosh OS. It is intended to replace the routines used to install patches documented in the Trap Manager chapter of *Inside Macintosh: Operating System Utilities*. The programmatic patching model, referred to throughout this chapter, is based on the use of these older routines. The Patch Manager is based on a new data-driven patching model that offers many advantages over the programmatic patching model used in System 7. Creating patches using the new model allows you to

- use a single patching model for head, tail, and surround patching
- patch any imported routine, not just operating system and toolbox routines
- create patches that have local or global effect
- enable and disable existing patches
- control the order in which patches execute
- obtain information about currently installed patches

Data-driven patching is designed to work for applications and server programs that rely on being prepared and launched automatically by the system. You cannot use data-driven patching if you are using low-level calls to the Code Fragment Manager to launch your program.

Mac OS 8 also supports the patching API documented in the Trap Manager chapter of *InsideMacintosh:Operating System Utilities*. Note however, that this API will not be supported in future versions of the Mac OS. Thus, if you are certain that you need to patch, you might want to modify your patching code using the model described in this chapter to ensure compatibility with future operating system releases.

Although this chapter describes patching in some depth, you should rarely, if ever, need to use patches in a program. Historically, Apple has used patches to fix problems and augment routines in ROM code. The packaging of system services as a set of updatable shared libraries has eliminated this need for patching. Application developers have also used patching to get information about system activity, to schedule services, or to customize the behavior of the system. Mac OS 8 includes widely expanded notification services, new scheduling services, and many additional routines that you can use to customize the behavior of the system. Inasmuch as patching a routine is less economical and less dependable than using these newer services, you should seriously consider using these rather than patching system routines. However, because it is impossible to anticipate every need, the patching mechanism described in this chapter has been provided for your use.

About Patching and the Patch Manager

Patching a routine allows you to assume control every time the routine is called from any task within a particular process. If you assume control in order to do some preparatory processing before the routine executes, the patch is called a **head patch.** If you want to do additional processing after the routine executes, the patch is called a **tail patch**. If you want to do both, the patch is called a **surround patch.** It is also possible, though it is strongly discouraged, to write a **replacement patch,** which is executed instead of the routine being called.

When several fragments patch the same routine, the result is a daisy chain of patches, or **patch chain,** with each patch in the chain executing in turn before the patched routine is called. Each patch in the chain can do some preprocessing and then call the patched routine, which, depending on the patch's position in the chain, might result in the next patch executing or might call the patched routine. After the patched routine executes, control is returned successively back through the patch chain so that each patch in the chain can do any needed postprocessing. Using the data-driven patching model defined for the Patch Manager, you can create every type of patch, specify when you want your patch to execute (relative to other patches in the patch chain), and have these patches automatically installed when the fragment owning the patches is instantiated. In addition, you can use Patch Manager functions

- to obtain information about all currently installed patches
- to determine which routines are being patched
- to enable or to disable patches

The following sections summarize the differences between programmatic and data-driven patching models, explains the special problems posed by patches with global effect, and examines the data-driven patching model in greater detail.

Programmatic and Data-Driven Patching

The patching model used in Mac OS 8 differs markedly from the model used in System 7. This section describes these differences and examines the patching model used in Mac OS 8 in greater detail.

The **programmatic patching** model defined for System 7 allows you to patch a system routine by replacing its address in the trap dispatch table with the address of your patch routine. If you need to call the patched routine from your patch, you are required to save the original address and then, in most instances, to write assembly language code that sets up the stack properly and then jumps to the saved address. Programmatic patching is in many ways limiting and costly to the programmer: It is limited to patching system software, it is difficult to implement, and it provides no control over the order in which patches execute. In addition, using this method it is not possible to examine the patch chain, which makes it next to impossible to identify and resolve patching conflicts.

The **data-driven patching** model defined for Mac OS 8 and future Mac OS releases differs markedly from the programmatic model. In Mac OS 8 any fragment (application or shared library) can define a patch by using a patch description fragment which is stored in the same file with that fragment. A fragment defining a patch in this way is said to own the patch and is called the patch's **owning fragment**. When the owning fragment is prepared for execution, the system looks for and installs the patches owned by the fragment as part of the fragment's launch process. (Because patches are installed at launch time, an import library brought in later, for example to support a plug in, will not have its patches installed.)

The **patch description fragment** contains a patch description structure for each patch that you want to install. The patch description structure specifies a reference to the patch, a reference to the patched routine, the name of the patch, and other information used to control the execution of the patch. You can store the patch routines in the fragment containing the patch description structures, but you are not required to do so.

Note
Programmatic patching takes place at the trap vector level. Because there are more system calls than trap vectors, many traps are dispatched, so that you were forced to patch many system routines when you only wanted to patch one. Data driven patching eliminates this problem by using shared library entry points in place of the more limited trap vectors. ◆

The main advantage of the data-driven patching model is that it gives you more control over the execution of your patches at the same time that it does more of the programming work: Once you write the patch routine and create the patch descriptions, you have no additional programming to do. The system assumes all responsibility for installing your patches in the patch chain in the order that you specify, for advising you about any conflicting patches, and for maintaining information about the patches in a chain. In addition, the Patch Manager provides routines that you can use to obtain information about all installed patches for all processes on a machine and to enable or disable any patch.

Patch Scope

A patch can have local or global scope. If you want to assume control only when a routine is called by your program (directly or indirectly), you need to create a patch that has local effect. This kind of patch is called a **local patch.** If you want to assume control when the routine is called by any process, you need to create a patch that has global effect. This kind of patch is called a **global patch.**

True global patching requires that a patch be installed at system boot time. Under Mac OS 8, all patches are local patches that are installed within a particular process. Because the Patch Manager installs a patch when it launches the patch's owning fragment, it is only possible to achieve global-effect patching by having the operating system install a different instance of a local patch in all processes as they are created. Any program that is launched before you install the global-effect patch is not affected by the patch.

Under System 7, a patch that has global scope is installed by INIT-type code and loaded in the system heap at boot time: the same instance of the patch and of any data initialized by the patch are visible and accessible to all processes that call the patched routine. The Mac OS 8 runtime architecture cannot support INIT-type code and does not support global programmatic patches.

To install a patch that has global effect in Mac OS 8, you must create a shared library fragment that uses per-process instantiation and a patch description fragment that contains the patch description structure. The patch code can reside in either fragment. Then you must place the shared library and the patch description fragment in a special folder in the Mac OS folder. The operating system instantiates such a library for each program, and the iteration of the local patch across all processes that are launched after the patch is so placed creates a patch with global effect. Each process contains its own copy of the patch and of any data initialized by that patch. For more information, see "Creating a Global-Effect Patch" on page 12-25.

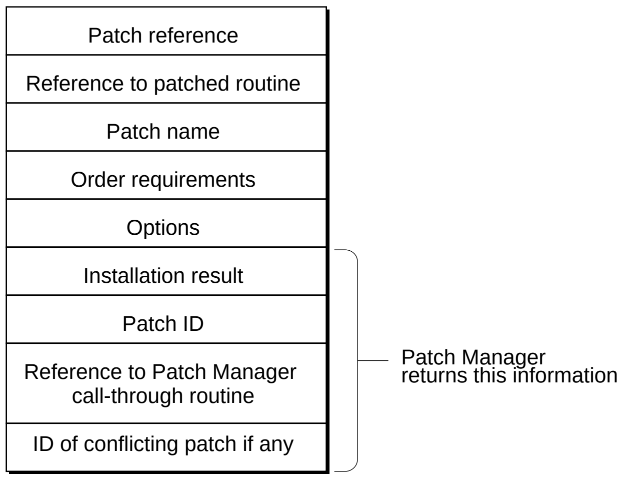
Because data-driven global-effect patches are actually iterated local patches, if you create a patch that needs to share data or communicate with other instances of the patch, you must explicitly use standard Mac OS 8 mechanisms for sharing data and synchronization.

Data-Driven Patching

This section describes the data-driven patching model in greater detail. It describes the data structure used to define a patch, explains the structure of the special fragment containing these structures, and discusses the ordering and execution of patches when multiple patches are applied to a single routine.

You use a data structure like the one shown in Figure 12-1 to describe each patch that you want to install.

Figure 12-1 The patch description structure


Patch reference
Reference to patched routine
Patch name
Order requirements
Options
Installation result
Patch ID
Reference to Patch Manager call-through routine
ID of conflicting patch if any
Patch Manager returns this information

You initialize the patch description structure to contain the address of the patch and of the patched routine, the name of the patch, and any order requirements and options you specify for the patch. You use the order requirements field to describe the order in which you would like the patch to execute relative to other patches. You use the options field to specify whether the patch is initially enabled or disabled and whether the patch must be successfully installed in order for the program to be launched.

Every patch must have a name that you assign to it when you create the patch. A patch name is composed of two parts: its signature and its type expressed as four-character literals. For example, `'SRFW' 'SpCh'`. The signature should be the same as the registered creator code for the program or shared library installing the patch. The type part of the name is a four-character literal of your choice.

You specify the order in which you want patches to execute by using the patch name: one of the fields of the patch description structure can specify the name of a patch you want to execute before or after your patch. By using wildcard characters to specify either part of the patch name when specifying ordering requirements, you can cause the execution of your patch to be ordered relative to sets of patches. For example, you can ask that your patch execute before all patches installed by a given program, or you can ask that your patch execute after all patches of a specific type.

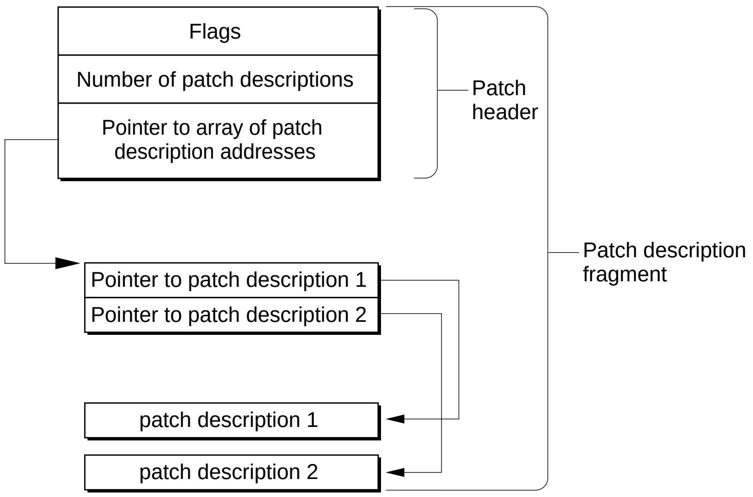
When your program or shared library is prepared for execution, the Code Fragment Manager also prepares any associated patch description fragment. The Code Fragment Manager performs any required relocations and fills in actual addresses for the patch address and the patched routine address. The Patch Manager examines the ordering requirements you specified for each patch and attempts to place the patch in a patch chain according to those requirements. If it is able to do so, it returns a unique **patch ID** to identify the patch. If it is not able to do so, it returns an error code in the installation result field of the patch description structure and it also returns the ID of the patch that caused the installation of your patch to fail if the failure was due to an ordering conflict.

The Patch Description Fragment

In order to install a patch, you must create a special patch description fragment and store this fragment in the same file as that of the owning fragment. The patch description fragment's `'cfrg'` resource should have the same name and usage code as that of the owning fragment, but it must have the update level 255. If any of these conditions are not met, the system will be unable to recognize and to install your patch.

The patch description fragment includes a patch header and one or more patch description structures. Figure 12-2 shows the structure of a patch description fragment that contains two patch description structures.

Figure 12-2 Patch description fragment.


Flags
Number of patch descriptions
Pointer to array of patch description addresses
Patch header
Patch description fragment
Pointer to patch description 1
Pointer to patch description 2
patch description 1
patch description 2

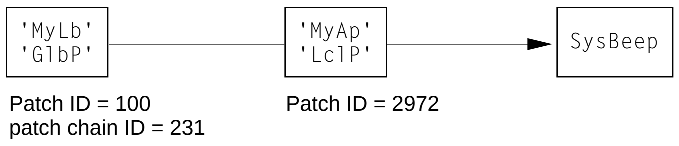
The patch header specifies the number of patch description structures in the fragment and points to an array containing the address of each structure. This array is called the **patch description reference array**. Referencing the patch descriptions by means of this array allows each patch description structure to have a proper variable name, thus helping to avoid programming mistakes.

The Code Fragment Manager and Patch Manager use the information provided by the patch header to locate all patch description structures in the fragment.The order in which patch description structures appear in the patch fragment affects the order in which the patches are installed but does not affect the order in which they execute. The order of execution is affected only by the values you specify for a patch's ordering requirements.

Applying Several Patches to the Same Routine

If several fragments patch the same routine, each patch is successively applied to the routine. Figure 12-3 shows two patches being applied to the routine `SysBeep`.

Figure 12-3 A patch chain



A **patch chain** is an ordered list of patches, all on the same instance of the same entry point. Figure 12-3 shows a patch chain for `SysBeep`. A patch chain includes a **root,** which is an instance of the routine being patched, and one or more patches. In Figure 12-3, the root of the patch chain is an instance of the routine `SysBeep`. The patch chain shown includes two patches: one named `'MyLb' 'GlbP'` and the other named `'MyAp' 'LclP'`. The smallest patch chain contains one patch and the patch root.

Each patch chain is identified by a **patch chain ID.** This identifier is a system-wide identifier. If two or more patch chains patch a different instance of the same routine, the patch chain that executes is the patch chain that is current for a given process.

Because the system does not differentiate between local-effect and global-effect patches, it is not possible to distinguish them in a patch chain, nor is it possible to obtain the IDs of all instances of a global-effect patch.

The order in which patches execute is determined by the ordering requirements you specify in the patch description structure used to describe the patch. Otherwise, no ordering hierarchy exists. Once the system has installed a patch in a patch chain, you can cause a patch not to execute by disabling it, but you cannot change its ordering requirements.

The Patch Manager provides routines that you can use to enable or disable a patch in a patch chain. It is important to understand that enabling a patch does not cause it to be added to the chain and that disabling a patch does not cause it to be removed from a chain. Enabling or disabling a patch simply determines whether the patch is executed when the patched routine is called. For this reason, it is not possible to remove an ordering conflict simply by disabling the offending patch. The Patch Manager must honor the ordering requirements of all patches, whether or not they are enabled.

The Structure of Patch Code

The structure of patch code used with data-driven patching is that of the surround patch. This type of patch

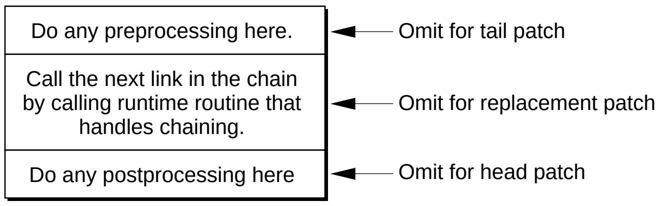
- does some preprocessing
- calls the patched routine
- does some post processing

Using the programmatic patching model, the call to the patched routine was the most difficult to implement. It involved saving the original address of the patched routine and then, in most cases, writing assembly-language code by means of which the stack and registers were properly set up, and then a jump or branch was effected to the saved routine address.

The data-driven patching model replaces the troublesome second step with a simple standard runtime call to the Patch Manager that you write in a high-level language. This call tells the Patch Manager to call the next link in the patch chain and supplies the information it needs (return result type, routine parameters) to make the call. If the next link in the chain is a patch, the Patch Manager passes control to that patch. If the next link is the chain root, the patched routine actually executes.

The structure of data-driven patching code is shown in Figure 12-4.

Figure 12-4 The structure of a patch


Do any preprocessing here.
Call the next link in the chain by calling runtime routine that handles chaining.
Do any postprocessing here
Omit for tail patch
Omit for replacement patch
Omit for head patch

As mentioned before, the patch code does not need to include any assembly code. It need only call the routine that handles the chaining. This routine ensures that the runtime environment is set up properly and that control passes smoothly to the next link in the chain. The Patch Manager returns the address of a call-through procedure that handles the chaining in the patch description structure. It is extremely important that each patch use the address returned to it during its installation and that patches do not share the address for this call-through procedure.

A **replacement patch** is a patch that does not include a call to the chaining routine. To implement such a patch, you define a patch description structure for it, but you never call the chaining routine. When the system launches your program, the Patch Manager automatically installs the patch by placing it in the patch chain for the patched routine. It is important to understand that because a replacement patch does not call the chaining routine, no patch following the replacement patch in the chain executes, including the routine being patched. For example, if `'MyAp' 'CPRS'` in Figure 12-3, is a replacement patch, neither `'WApp' 'spel'` nor `'DApp' 'graf'`, nor `SysBeep` are ever executed.

Order Requirements

You use the order requirements field of the patch description structure to specify when your patch should execute. You use this field to specify the name of the patch after which you want your patch to execute and to specify the name of the patch before which you want your patch to execute.

The Patch Manager allows you to specify ordering requirements relative only to other patches, not relative to the patched routine. For example, you can specify that your patch execute

- any time before one patch and after another patch
- immediately before one patch and any time after another patch
- any time before one patch and immediately after another patch

Because you can use wildcard characters to specify a patch name or type, it is also possible to order your patch relative to a set of other patches.

It is possible (though not recommended) to specify order requirements that cause a replacement patch to be placed last in the chain (right before the patched routine). In this way all other patches in the patch chain execute before the replacement patch. However, in general, it is not a good idea to impose order requirements unless they are crucial to the performance of your code. If the Patch Manager cannot resolve conflicting order requirements, it is unable to install your patch or that of another program. This might result in unpredictable behavior and poor user experience.

IMPORTANT
You cannot cause your patch to be installed by disabling the patch whose order requirements conflict with yours. The Patch Manager must honor the ordering requirements of a patch even when that patch is disabled. Thus, the only way to eliminate ordering conflicts is to change your own ordering requirements. ▲

Limitations on Patching

If you use data-driven patching, you must observe the following restrictions:

- Patch code must be native.
- A library that owns a patch description fragment must use per-process instantiation.
- You can use only the generic routine that calls the next link in a patch chain to call patch code. That is, you can only call patch code as part of a patch chain.
- You must call the procedure that handles the chaining from the main patch routine, not from a subroutine.
- You cannot use the same patch code to patch two different routines.
- A fragment can have only one patch fragment associated with it.
- You must place the patch fragment and owning fragment in the same file.

Compatibility

The Patch Manager ensures that the trap patching API defined with System software 7 is supported in Mac OS 8. Support is limited to local patching. Table 12-1 lists the names of the functions that continue to be supported in Mac OS 8.

Table 12-1 Programmatic patching calls supported under Mac OS 8

Function
`GetTrapAddress`
`SetTrapAddress`
`NGetTrapAddress`
`NSetTrapAddress`
`GetOSTrapAddress`
`SetOSTrapAddress`
`GetToolTrapAddress`
`SetToolTrapAddress`
`GetToolboxTrapAddress`
`SetToolboxTrapAddress`

These functions are fully described in the Trap Manager chapter of *Inside Macintosh: Operating System Utilities*.

If you use these functions to install a programmatic patch, you can still use all the functions described in this chapter to obtain information about patches (including the programmatic patch) and to enable or disable a patch.

Using the Patch Manager

The Patch Manager provides data structures that you use to describe your patch and the order in which you would like it to execute. The Patch Manager also provides functions that you can use to

- obtain a list of all the patches in a patch chain
- obtain information about any patch in a chain
- obtain a list of all the patch chains associated with a process
- determine the process to which a patch chain belongs
- determine the root of a patch chain
- determine the patch chain to which a routine belongs (if any)
- enable or disable any patch in a chain

This section explains how you use Patch Manager data structures and functions to create a patch, to control its execution, and to obtain information about currently installed patches.

Creating a Patch

This section explains the steps required to create a patch. Creating a patch involves

- Creating the source for the patch description fragment

  The source includes initializing a patch description header and one patch description for each patch.

  The patch description header specifies the number of patch descriptions you are going to include in the patch description file and provides a pointer to the array referencing the patch description structures.

  The patch description specifies the patch, the patched routine, the patch name, and other options that govern the installation and execution of the patch.

- Writing the patch routine and placing it either in the patch description fragment or in any fragment that you are going to link with the patch description fragment.
- Compiling and linking the source files containing the patch description header and the patch descriptions into a patch description fragment.

  In order for the patch to be recognized and installed, the patch description fragment's `'cfrg'` resource must have the same values for the name and usage type fields as those in the owning fragment's `'cfrg'` resource, but the update level field of the patch description fragment's `'cfrg'` resource must be 255.
- Add the patch fragment to the owning fragment's file.

The following two sections explain how you write a patch header and a patch description.

The Patch Description Header

The patch description header specifies the number and location of patch description structures. The patch description header must be the main symbol of the patch description fragment.You use the `PatchHeader` data type (page 12-32) to define a patch description header structure. The Patch Manager uses the first two fields of the structure for version control. You use the `count` field to specify the maximum number of patch description structures in the fragment. This number can exceed the actual number of patch descriptions; the Patch Manager strips trailing null pointers. You use the `patches` field to reference the patch descriptions.

Listing 12-1 shows a sample patch header. It uses the constant `kMyPatchCount` to specify the number of patch descriptions and the pointer `&gPatchDescriptionList[0]` to reference the array containing the addresses of the patch description structures.

Listing 12-1 Sample patch header

```
PatchHeader gPatchData = {
        kPatchHeaderTag,
        kPatchHeaderVersion,
        kNilOptions,
        kMyPatchCount,
        &gPatchDescriptionList[0]
        };
```

The system uses the `flags` field of the patch header to let you know whether it has failed to install any of your required or optional patches. If you are installing many patches, knowing that all your patches have been successfully installed can save you the trouble of examining each patch description structure to obtain the same information.

The Patch Description Structure

You must create a patch description structure for each patch that you want to install. The `PatchDescription` data type (page 12-33) defines a patch description structure. You use the first five fields of the structure to describe your patch: you provide a reference to the patched routine, a reference to the patch, a name for the patch, a constant value indicating the order in which you want the patch to execute relative to other patches, and a constant value indicating whether the patch should be initially enabled or disabled. The section "Order Requirements," beginning on page 12-13 explains how you specify a value for the field `thisPatchOrdering`.

The Patch Manager uses the last four fields of the patch description structure to return information about the installation of the patch. It tells you whether the installation succeeded, it returns the ID of the patch, it returns a pointer to the Patch Manager routine that handles the chaining, and, if your patch could not be installed because its ordering requirements conflicted with those of another patch, it returns the ID of the conflicting patch. A good time to check for these returned values would be in the beginning of the owning fragment's initializing routine. Listing 12-2 shows a sample patch description. It specifies the name of the patch to be `'wild',demo';` it specifies that the patch can be placed anywhere in the patch chain and that it should be initially enabled.

Listing 12-2 Sample patch description

```
PatchDescription gSysBeepPatchDescription = {
    &SysBeep,
    &MySysBeepPatch,
    {   'wild',
        'demo'},
    {   kNilOptions,
        {   kDoNotMatchAnyOrderedItemService,
            kDoNotMatchAnyOrderedItemSignature},
        {   kDoNotMatchAnyOrderedItemService,
            kDoNotMatchAnyOrderedItemSignature}
    },
    kPatchEnabledMask,
    paramErr,           /* install result returned here */
    kInvalidID,         /* ID of patch returned here */
    NULL,               /* pointer to call through proc returned here */
    kInvalidID          /* conflicting patch ID returned here */
};
```

The initial values of the four output fields are not critical; however, using the ones shown in Listing 12-2 can help ensure that errors are detected.

Listing 12-3 shows a sample surround patch for the routine `SysBeep`.

Listing 12-3 Sample patch code

```
void MySysBeepPatch(SInt16 duration)
{   DoSomePreprocessing();

/* Call through */
    (* (SysBeepPatchProcPtr)
            gSysBeepPatchDescription.thisCallThroughProc) (duration);

    DoSomePostprocessing();
    return;
}
```

Listing 12-4 shows the contents of a patch description fragment: the patch description, patch description header, and the patch code that has already been described in the previous listings. In addition, it contains an initialization routine that halts processing if the patch cannot be installed.

Listing 12-4 Patch Description Fragment

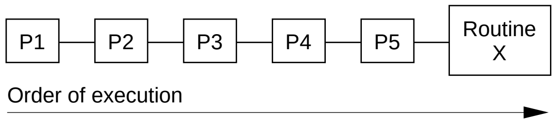
```
#include    <Types.h>
#include    <Patches.h>
#include    <CodeFragments.h>
#include    <Sound.h>

void MySysBeepPatch(SInt16 duration);
void DoSomePreprocessing(void);
void DoSomePostprocessing(void);

enum {
    kMyPatchCount=1 /* Number of patches in my list */
};

/* One patch description per patch */
PatchDescription gSysBeepPatchDescription = {
    &SysBeep,
    &MySysBeepPatch,
    {'wild',
        'demo'},
        {
            kNilOptions,
            { kDoNotMatchAnyOrderedItemService,
                kDoNotMatchAnyOrderedItemSignature},
            { kDoNotMatchAnyOrderedItemService,
                kDoNotMatchAnyOrderedItemSignature}
        },
    kPatchEnabledMask,
    paramErr,
    kInvalidID,
    NULL,
    kInvalidID
};

/* The table with the list of patch descriptions; this one includes
    only one patch */
PatchDescription * gPatchDescriptionList[kMyPatchCount] = {
                                &gSysBeepPatchDescription};

/* The Patch header. This is exported as the patch fragment's main entry
```

```
point. */
PatchHeader gPatchData =
    {
        kPatchHeaderTag,
        kPatchHeaderVersion,
        kNilOptions,
        kMyPatchCount,
        &gPatchDescriptionList[0]
    };



/* Do some preprocessing here */
void DoSomePreprocessing(void)
{
/*  Put preprocessing code here */
    Debugger();
}

void DoSomePostprocessing(void)
{
/*  Put postprocessing code here */
    Debugger();
}

typedef void (*SysBeepPatchProcPtr)(SInt16 duration);

/*  The patch itself.
    Does some preprocessing,
    calls through the rest of the patch chain, and
    then does some postprocessing. */
void MySysBeepPatch(SInt16 duration)
{   DoSomePreprocessing();

/* Call the routine that handles the chaining*/
    (* (SysBeepPatchProcPtr)
        gSysBeepPatchDescription.thisCallThroughProc) (duration);

    DoSomePostprocessing();

    return;
}

/* CFM init routine.
    Do not proceed if the patch could not be installed. */

OSErr InitMyPatch(const CFragInitBlock *initBlock)
{
    return (gSysBeepPatchDescription.installResult);
}
```

Specifying Order Requirements

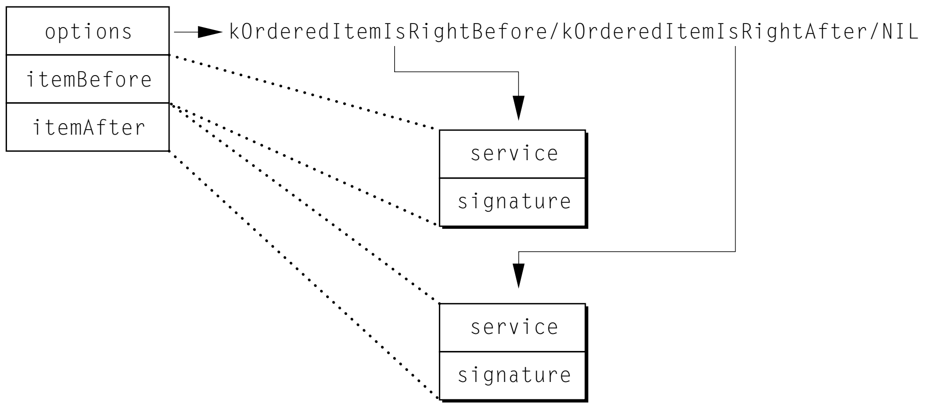
You specify the order in which you want your patch to execute relative to the name of other patches in the same patch chain. You use the field `thisPatchOrdering` in the patch description structure to specify the order in which you want your patch to execute. It is easiest to explain how you use this structure by referring to a sample patch chain like the one shown in Figure 12-5.

Figure 12-5 Sample patch chain



The figure shows a set of five patches, in positions P1 through P5, being applied to a routine. The structure you use to control the placement of your patch in that chain is shown in Figure 12-6.

Figure 12-6 Patch order requirements structure



The structure contains three fields. You use the field `itemBefore` to specify the name of the patch after which you want your patch to execute; for example, if you want your patch to execute after the patch in position P1, you can specify the name of the patch in position P1 in the `itemBefore` field when you create the patch. You use the field `itemAfter` to specify the name of the patch before which you want your patch to execute. For example, if you want your patch to execute before the patch in position P4, you can specify the name of the patch in position P4 in the `itemAfter` field when you create the patch.

If you specify that your patch should execute relative to a patch that does not exist, the Patch Manager still executes your patch. Thus, requiring that your patch execute after patch X is the same as requiring that *if* patch X exists, your patch should execute after it. It is not the same as requiring *that* patch X exist.

You use the `options` field of the order requirements structure to make the selection specified with either of the other two fields more precise:

- If you want the patch specified in the `itemBefore` field to execute immediately before your patch, specify the constant `kOrderedItemIsRightBefore` in the `options` field.
- If you want the patch specified in the `itemAfter` field to execute immediately after your patch, specify the constant `kOrderedItemIsRightAfter` in the `options` field.

Note that this scheme allows you to order your patch immediately after or immediately before another patch, but not both. It is best, however, unless the execution of your patch absolutely requires you to specify one of these options, that you set the options field to `NIL`. This results in a looser ordering requirement and, consequently, in fewer ordering conflicts when the system installs your patch or other patches in the same patch chain.

You can also use the constant names listed in Table 12-2 in the `itemBefore` or `itemAfter` fields to specify your location relative to a *set* of patches.

Table 12-2 Wildcard specifiers

Constant name	Meaning
`kMatchAnyOrderedItemService`	Place my patch before/after every patch name whose signature field matches my signature field.
`kMatchAnyOrderedItemSignature`	Place my patch before/after every patch name whose service field matches that specified in my service field.
`kDoNotMatchAnyOrderedItemService`	I don’t care where you place my patch.
`kDoNotMatchAnyOrderedItemSignature`	I don’t care where you place my patch.

The following examples illustrate the use of the constant names shown in Table 12-2.

- To have your patch execute right before the patched routine, specify `kMatchAnyOrderedItemService` and `kMatchAnyOrderedItemSignature` in the `itemBefore` field. Specify `kDoNotMatchAnyOrderedItemService` and `kDoNotMatchAnyOrderedItemSignature` for the `itemAfter` field.
- To have your patch execute first, specify `kMatchAnyOrderedItemService` and `kMatchAnyOrderedItemSignature` in the `itemAfter` field. Specify `kDoNotMatchAnyOrderedItemService` and `kDoNotMatchAnyOrderedItemSignature` for the `itemBefore` field.

- To have your patch execute before a patch installed by another program, specify that program's creator code in the signature field of the `itemAfter` field, and set the service field to `kDoNotMatchAnyOrderedItemService`.

IMPORTANT

The more restrictive you make the order requirements for your patch, the more likely they are to conflict with the order requirements of other patches. As a result, it will not be possible to install your patch or other conflicting patches. ▲

Creating a Local Patch

The steps required to create a local patch are listed in "Creating a Patch," beginning on page 12-16. The owning fragment can be an application, a shared library, or any other kind of fragment except an update library.

Creating a Global-Effect Patch

The steps required to create a global-effect patch are exactly the same as those for creating a local patch except that the owning fragment must be a shared library using per-process instantiation (rather than an application). If the patch code resides in the patch description fragment, the shared library fragment can be empty.

To have the Patch Manager install the patch for all programs referencing the patched routine, you must place the shared library and the patch description fragment in the Co-operative Extension Libraries folder, which resides in the Extension Libraries folder, which resides in the Mac OS folder. Any programs that are launched after the shared library and patch description fragment are so placed, will see the patch.

Creating a Patchable Shared Library

If you want to create a patchable shared library, it is important to make sure that your development system generates all internal calls to your exported routines as indirect calls. Some compilers and linkers do not call through transition vectors for routines in the same compilation unit. The data-driven patching mechanism depends upon patchable routines being accessed through transition vectors. If your compiler and linker do not call through a transition vector for an internally referenced routine and such a routine is patched, the direct internal calls to the routine will not see the patch.

Obtaining Information About Patches

The Patch Manager supports a variety of functions that return information about installed patches. This section explains how these functions relate to one another and provides a more detailed discussion of the `GetPatchInformation` function, which returns information about a patch. In general, you should not do anything with patches that you don't own, except to look at them.

Table 12-3 shows the input and output parameters of the Patch Manager functions that return information.

Table 12-3 Patch Manager functions that return information

Function	Input	Output
`GetPatchChainsInKernelProcess`	Process ID	Patch chain IDs
`GetPatchChainInformation`	Patch chain ID	Process ID Chain root
`GetPatchChainFromProcPtr`	Routine	Patch chain ID
`GetPatchFromProcPtr`	Routine	Patch ID
`GetPatchesInPatchChain`	Patch chain ID	List of patch IDs
`GetPatchInformation`	Patch ID	Patch chain ID Patch address Options Patch name

As you can see, given any single piece of information, you can use these functions to determine how that piece fits into the current patching scheme for all installed patches. For example, for any given process, you can determine the IDs of the patch chains associated with it and consequently the IDs of all the patches in the patch chains. For any given routine, you can determine whether it is included in a patch chain and, if so, the process to which the patch chain belongs.

You use the `GetPatchInformation` function to obtain information about a single patch; the function returns this information in a patch information structure. This includes the ID of the patch chain to which the patch belongs, the address of the patch, the name of the patch, its order requirements, and the options that are currently set for the patch. Current option settings are returned in the `patchOptions` field; these determine whether

- the patch is optional or required

  An optional patch is a patch that does not have to be installed in order for the program that references the patched routine to be launched. A required patch is a patch that has to be installed in order for the program to be launched.

- the patch is a data-driven patch or a programmatic patch

  In Mac OS 8, some programs might call routines that are patched programmatically. Such patches are included in the patch chain and you can use Patch Manager functions to obtain information about these patches, but you should not try to manipulate them in any other way.

- the patch is currently enabled or disabled

  You can examine the `installOptions` field of the patch description structure to determine whether the patch is initially enabled or disabled.

Using Programmatic Patching

In Mac OS 8, the trap patching API defined for System 7 is implemented using data driven patching. This is why it is possible to obtain information about programmatic patches using the Patch Manager `GetPatchInformation` function.

If you are using programmatic patching, keep the following in mind:

- Mac OS 8 cannot support global programmatic patching because it does not support INIT type code.
- Optimizing compilers might assume that the contents of transition vectors are constant within a function and might prefetch their contents into nonvolatile registers. Because patching involves modification of the code address in a transition vector, installing a programmatic patch might invalidate the prefetched code address. To avoid such problems, you should not reference the patched routine from any routine that is active when the patch is installed, or you should make use of appropriate compilation pragmas, or you should compile the calling routines with reduced

optimizations. Data driven patches do not have this problem because your code is not running when the patch is installed.

Patch Manager Reference

This section describes the data types and functions you use to install, enable, disable, and obtain information about patches.

Constants and Data Types

You use the constants and data types described in this section to define the patch description header and the patch descriptions, which are contained in the patch description fragment.

Option Bits Mask Enumeration

The option bits mask enumeration specifies possible values for the `installOptions` field of the patch description structure (page 12-33) and for the `patchOptions` field of the patch information structure (page 12-36). When you install a patch, you use the patch description structure to describe the patch. After the patch is installed, the Patch Manager returns information about its current state in a patch information structure.

You can use the `bit OR` operator to combine two or more of the following values.

```
enum {
    kPatchEnabledMask       =   (1L << kPatchEnabledBit),
    kPatchCompatibilityMask =   (1L << kPatchCompatibilityBit),
    kPatchOptionalMask      =   (1L << kPatchOptionalBit)
};
```

Enumerator descriptions

`kPatchEnabledMask`	Before installing a patch, you use this bit in the `installOptions` field of the patch description structure to specify whether the patch is initially enabled. After installing the patch, you can use the `EnablePatch` function

(page 12-37) and `DisablePatch` function (page 12-38) to enable and disable the patch.

When you examine the `patchOptions` field of the patch description structure, the setting of this bit indicates the initial state of the patch: The bit is set if the patch was enabled; the bit is clear if the patch was disabled. To obtain the current state of the patch, you must call the `GetPatchInformation` function (page 12-46).

`kPatchCompatibilityMask`
Do not use this bit in the `installOptions` field of the patch description structure.

If this bit is set in the `patchOptions` field of the patch information structure, it means that this is a programmatic patch. If this bit is clear, it means that this is a data-driven patch.

`kPatchOptionalMask` The setting of this bit in the `patchOptions` field of the patch description structure, tells the Code Fragment Manager how to proceed if it cannot install a patch. If this bit is set, it means the Code Fragment Manager should launch your program anyway. If this bit is clear, the Code Fragment Manager should not launch your program.

Wildcard Enumeration

You use the wildcard enumeration to specify values for the `service` and `signature` fields of the ordered item name structure (page 12-34). You use the ordered item name structure to specify values for the `itemBefore` and `itemAfter` fields of the order requirements structure (page 12-35). The effect of using wildcard order specifiers is explained in “Specifying Order Requirements” on page 12-22.

```
enum {
    kMatchAnyOrderedItemService         =   (OrderedItemService)'****',
    kMatchAnyOrderedItemSignature       =   (OrderedItemSignature)'****',
    kDoNotMatchAnyOrderedItemService    =   (OrderedItemService)'----',
    kDoNotMatchAnyOrderedItemSignature  =   (OrderedItemSignature)'----'
};
```

`kMatchAnyOrderedItemService`
: In the `service` field of the `itemBefore` field, this value causes your patch to execute after all other patches whose signature matches that specified in the `signature` field of the `itemBefore` field.

  In the `service` field of the `itemAfter` field, this value causes your patch to execute before all other patches whose signature matches that specified in the `signature` field of the `itemAfter` field.

`kMatchAnyOrderedItemSignature`
: In the `signature` field of the `itemBefore` field, this value causes your patch to execute after all other patches whose service matches that specified in the `service` field of the `itemBefore` field.

  In the `signature` field of the `itemAfter` file, this value causes your patch to execute before all other patches whose `service` field matches that specified in the `service` field of the `itemAfter` field.

`kDoNotMatchAnyOrderedItemService`
: In the `service` field of the `itemBefore` field, this value means that you don't care if any patch executes before your patch.

  In the `signature` field of the `itemAfter` field, this value means that you don't care if any patch executes after your patch.

`kDoNotMatchAnyOrderedItemSignature`
: In the `service` field of the `itemBefore` field, this value means that you don't care if any patch executes before your patch.

  In the `signature` field of the `itemAfter` field, this value means that you don't care if any patch executes after your patch.

Ordered Item Enumeration

You use the ordered item enumeration to specify a value for the `options` field of the order requirements structure (page 12-35). This value determines whether your patch is executed immediately before or immediately after another patch.

Set this field to `kNilOptions`, to indicate that the patch does not have execute right before or right after another patch.

```
enum {
    kOrderedItemIsRightBefore   =   0x00000001,
    kOrderedItemIsRightAfter    =   0x00000002
};
```

`kOrderedItemIsRightBefore`
: The `itemBefore` field of the order requirements structure specifies the name of the patch that should execute before your patch. If you want that patch to execute immediately before your patch, specify this constant name for the `options` field of the order requirements structure.

`kOrderedItemIsRightAfter`
: The `itemAfter` field of the order requirements structure specifies the name of the patch that should execute after your patch. If you want that patch to execute immediately after your patch, specify this constant name for the `options` field of the order requirements structure.

Patch Header Flags Enumeration

The patch header flags enumeration is used by the Patch Manager to specify a value for the `flags` field of the patch header structure (page 12-32).

After the Code Fragment Manager has loaded your code fragment and patch fragment, and has installed your patches, it uses the `installResult` field of the patch description structure to let you know whether a particular patch was installed. At the same time, the Code Fragment Manager sets a bit in the `flags` field of the patch header structure to let you know whether *any* required or optional patch in a given patch chain set has failed. Thus, if you examine the `flags` field first and find either or both bits clear, you can save yourself the trouble of looking at the `installResult` field for each patch you have installed.

```
typedef OptionBits PatchHeaderOptions;
enum {
    kRequiredPatchErrorsMask    = 0x00000001,
    kOptionalPatchErrorsMask    = 0x00000002
};
```

Enumeration descriptions

`kRequiredPatchErrorsMask`	If this bit is set, the system has failed to install one or more required patches.
`kOptionalPatchErrorsMask`	If this bit is set, the system has failed to install one or more optional patches.

Patch Header Tag Enumeration

You use the patch header tag enumeration to specify values for the `tag` field and the `version` field of the patch header structure (page 12-32).

```
enum {
    kPatchHeaderTag     = 'Ptch'
    kPatchHeaderVersion = 1
};
```

Enumeration descriptions

`kPatchHeaderTag`	Value for the `tag` field of the patch header structure.
`kPatchHeaderVersion`	Value for the `version` field of the patch header structure.

Patch Header Structure

You use the patch header structure to specify the address and size of the array containing references to the patch descriptions for your patches.

The `PatchHeader` data type defines a patch header structure.

```
struct PatchHeader {
    OSType                  tag;
    UInt32                  version;
    PatchHeaderOptions      flags;
    ItemCount               count;
    PatchDescriptionPtr *   patches;
};
typedef struct PatchHeader PatchHeader;
```

Field descriptions

`tag`	The constant name `kPatchHeaderTag`.
`version`	The constant name `kPatchHeaderVersion`.
`flags`	This field is set by the Code Fragment Manager to one of the values defined by the patch header flags enumeration (page 12-31).
`count`	The number of entries in the patch reference array whose address is stored in the `patches` field, described next. This value should be the same as the dimension of the array, but the Patch Manager will gracefully deal with nulls anywhere in the array.
`patches`	A pointer to an array of pointers to patch description structures that specify the patches that you want to install.

Patch Description Structure

You use the patch description structure to specify a reference to your patch routine, a reference to the routine you want to patch, the name of your patch, and additional information about how you want the Patch Manager to execute your patch. The Patch Manager uses some fields in this structure to return information about the patch if it was successfully installed or to let you know about possible sources of conflict if it could not install the patch.

You must include one patch description structure for each patch you want to install. The `PatchDescription` data type defines a patch description structure.

```
struct PatchDescription {
    PatchableProcPtr        originalRoutine;
    PatchableProcPtr        patchRoutine;
    PatchName               thisPatchName;
    PatchOrderRequirements  thisPatchOrdering;
    PatchOptions            installOptions;
    OSStatus                installResult;
    PatchID                 thisPatchID;
    PatchableProcPtr        thisCallThroughProc;
    PatchID                 rejectingPatchID;
};
typedef struct PatchDescription PatchDescription;
```

Field descriptions

`originalRoutine`	A pointer to the routine you want to patch.
`patchRoutine`	A pointer to the patch routine.
`thisPatchName`	The name of your patch routine. You use the ordered item name structure (page 12-34) to specify the name of your patch.
`thisPatchOrdering`	The order in which you want your patch to execute. You use the order requirements structure (page 12-35) to specify whether your patch should execute before or after some other named patch. You can specify `NULL` if you do not care when the pach executes.
`installOptions`	A value given by the option bits mask enumeration (page 12-28) specifying whether the patch is initially enabled and whether the Code Fragment Manager should load your program if it is unable to install your patch.
`installResult`	The Patch Manager returns 0 in this field if the patch was successfully installed, or a nonzero result if it was not.
`thisPatchID`	The Patch Manager returns the patch ID in this field if the patch was successfully installed.
`thisCallThroughProc`	The Patch Manager sets this field to the address of the routine you use to transfer control to the next routine in the patch chain.
`rejectingPatchID`	The Patch Manager sets this field to the ID of a patch whose ordering requirements conflict with those you have specified for your patch. Disabling the conflicting patch does not solve ordering conflicts, but changing your ordering requirements might resolve the conflict. If there are no conflicts, the Patch Manager sets this field to the constant value `kInvalidID`.

Ordered Item Name Structure

You use the ordered item name structure to specify values for the `itemBefore` and `itemAfter` fields of the order requirements structure (page 12-35). A patch must be uniquely named within a patch chain.

The `OrderedItemName` data type defines an ordered item name structure.

```
struct OrderedItemName {
    OrderedItemService      service;
    OrderedItemSignature    signature;
};
typedef struct OrderedItemName OrderedItemName, *OrderedItemNamePtr;
```

`service`	A four-character literal that you define to identify a patch or one of the wildcard service enumerators (page 12-29).
`signature`	A four-character literal that specifies the program's creator or one of the wildcard signature enumerators (page 12-29).

Order Requirements Structure

You use the order requirements structure to specify a value for the field `thisPatchOrdering` of the patch description structure (page 12-33). This field defines the relative order in which you want your patch to execute.

The use of this structure is explained in "Specifying Order Requirements" on page 12-22. The `OrderRequirements` data type defines the order requirements structure.

```
struct OrderRequirements {
    OrderedItemOptions  options;
    OrderedItemName     itemBefore;
    OrderedItemName     itemAfter;
};
typedef struct OrderRequirements OrderRequirements, *OrderRequirementsPtr;
```

`options`	The value you specify for this field determines whether the item specified in the `itemBefore` field should come immediately before your patch or whether the item specified in the `itemAfter` field should come immediately after your patch. You specify one of these two values using the ordered item enumeration (page 12-30). If you do not need to specify either, use `kNilOptions`.
`itemBefore`	The ordered item name structure (page 12-34) for the patch that should execute before your patch.

`itemAfter`	The ordered item name structure (page 12-34) for the patch that should execute after your patch.

Patch Information Structure

The Patch Manager uses the patch information structure to return information to you about a patch specified by the `PatchID` parameter to the `GetPatchInformation` function (page 12-46).

The `PatchInformation` data type defines a patch information structure.

```
struct PatchInformation {
    PatchChainID             patchChain;
    PatchableProcPtr         patchingRoutine;
    PatchOptions             patchOptions;
    PatchName                patchName;
    PatchOrderRequirements  patchOrder;
};
typedef struct PatchInformation PatchInformation, *PatchInformationPtr;
```

Field descriptions

`patchChain`	The patch chain ID of the patch chain to which the patch belongs.
`patchingRoutine`	The address of the patch routine.
`PatchOptions`	A value given by the option bits enumeration (page 12-28) specifying whether the patch is currently enabled, whether it is a programmatic or data-driven patch, and whether the code fragment loader can launch a program even when it cannot install the patch.
`patchName`	The name of the patch. If this is a programmatic patch, the name is one assigned by the Patch Manager.
`patchOrder`	An order requirements structure (page 12-35) specifying the names of patches whose execution must precede or follow that of this patch.

Patch Chain Information Structure

You call the `GetPatchChainInformation` function (page 12-41) to determine the address of the routine that is being patched and the process ID of the process that is associated with the patch chain. The `GetPatchChainInformation` function returns this information in a patch chain information structure.

The `PatchChainInformation` data type defines a patch chain information structure.

```
struct PatchChainInformation {
    KernelProcessID     kernelProcess;
    PatchableProcPtr    chainRoot;
};
typedef struct PatchChainInformation PatchChainInformation,
                                    *PatchChainInformationPtr;
```

Field descriptions

`kernelProcess`	The ID of the process that is associated with the patch chain.
`chainRoot`	A reference to the routine being patched.

Functions

This section describes the function you use to enable, to disable patches and to obtain information about patches.

Enabling and Disabling Patches

When you define a patch using the patch description structure (page 12-33), you use the `installOptions` field to specify whether the patch should be enabled or disabled by default. You use the functions described in this section to change that default setting.

EnablePatch

Enables a patch.

```
OSStatus EnablePatch (PatchID thePatch);
```

`thePatch`	The ID of the patch being enabled.

DISCUSSION

Enabling a patch causes the Patch Manager to execute the patch when the routine being patched is called. The order in which the patch executes depends upon ordering constraints specified by the field `thisPatchOrdering` of the patch description structure for this patch.

Any client that can obtain a patch ID can enable or disable a patch.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

Use the `DisablePatch` function (page 12-38) to disable a patch.

Use the `GetPatchInformation` function (page 12-46) to determine whether a patch is enabled or disabled.

You use the patch description structure (page 12-33) to specify any ordering requirements for a patch.

DisablePatch

Disables a patch.

```
OSStatus DisablePatch (PatchID thePatch);
```

`thePatch`	The ID of the patch being disabled.

DISCUSSION

Disabling a patch causes the Patch Manager not to execute the patch when the routine being patched is called.

Disabling a patch is not the same as removing a patch from the patch chain. For example, if any ordering conflicts exist, they remain even though the patch causing the conflict is disabled.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

Use the `EnablePatch` function (page 12-37) to enable a patch.

Use the `GetPatchInformation` function (page 12-46) to determine whether a patch is enabled or disabled.

Obtaining Information About Patch Chains

You use the functions described in this section to obtain a list of patch chains associated with a process and to obtain information about a particular patch chain.

GetPatchChainsInKernelProcess

Returns a list of all patch chains associated with a process.

```
OSStatus GetPatchChainsInKernelProcess(KernelProcessID theKernelProcess,
                    ItemCount requestedPatchChains,
                    ItemCount *totalPatchChains,
                    PatchChainID *thePatchChains);
```

`theKernelProcess`
: The ID of the process with which the patch chains are associated. Specify `kCurrentKernelProcessID` to indicate the current process.

`requestedPatchChains`
: An integer specifying the size of the array in which this function stores patch chain information when it returns.

`totalPatchChains`
: A pointer to the total number of patch chains associated with the specified process.

`thePatchChains`
: A pointer to an array of patch chain IDs. On return, the `GetPatchChainsInKernelProcess` function stores the patch chain IDs in the specified process in this array.

DISCUSSION

The patch chain IDs in the array referenced by `thePatchChains` parameter are not listed in any particular order.

If you call this function and the value specified by the `requestedPatchChains` parameter is smaller than the value specified by the `totalPatchChains` parameter, this means that the array you have allocated for the patch chain IDs is too small. If you want to obtain more information, you must set the `requestedPatchChains` parameter to the desired value and then call the function again.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

Use the `GetPatchChainInformation` function (page 12-41) to get information about a specific patch chain.

You use the `GetPatchesInPatchChain` function (page 12-45) to obtain a list of all the patches in a patch chain.

GetPatchChainInformation

Returns information about a patch chain.

```
OSStatus GetPatchChainInformation (PatchChainID thePatchChain,
                    PBVersion version,
                    PatchChainInformation *patchChainInfo);
```

`thePatchChain`	The ID of the patch chain about which information is sought.
`version`	The constant `kPatchChainInformationVersion`.
`patchChainInfo`	A pointer to a patch chain information structure (page 12-36) that the `GetPatchChainInformation` function fills in when it returns.

DISCUSSION

For a given patch chain ID, the `GetPatchChainInformation` function returns a patch chain information structure that specifies the ID of the process with which the patch chain is associated and the address of the routine that is being patched.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

Use the `GetPatchChainsInKernelProcess` function (page 12-40) to find out what other patch chains are associated with the process to which this patch chain belongs.

Determining Whether a Routine is a Patch

You use the functions described in this section to determine whether a routine belongs to a patch chain and whether it is a patch or the routine being patched.

GetPatchChainFromProcPtr

Determines whether a routine belongs to a patch chain.

```
OSStatus GetPatchChainFromProcPtr (KernelProcessID theKernelProcess,
                        PatchableProcPtr theRoutine,
                        PatchChainID *thePatchChain);
```

`theKernelProcess`	The process ID of the process to which the routine belongs.
`theRoutine`	The address of the routine.
`thePatchChain`	A pointer to the ID of the patch chain to which the routine belongs either because it is a patch or because it is the routine being patched (the root of the chain).

DISCUSSION

If the parameter `theRoutine` specifies a routine that is either a patch in a patch chain or the root of a patch chain, the `GetPatchChainFromProcPtr` function returns the ID of the patch chain to which the routine belongs. Otherwise, the function returns an error.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To determine whether the routine is a patch or a root, use the `GetPatchFromProcPtr` function (page 12-44).

GetPatchFromProcPtr

Determines whether a routine is a patch and returns its ID if it is.

```
OSStatus GetPatchFromProcPtr(KernelProcessID theKernelProcess,
                    PatchableProcPtr theRoutine,
                    PatchID *thePatch);
```

`theKernelProcess`	The ID of the process to which the routine belongs.
`theRoutine`	The address of the routine.
`thePatch`	A pointer to the ID of the patch. If the routine is a patch, the `GetPatchFromProcPtr` function sets this field when it returns.

DISCUSSION

If the specified routine is a patch, this function references the ID of the patch in the parameter `thePatch`. If the routine is a root or cannot be found in a patch chain, the function returns an error.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

If the routine is a patch, you can obtain additional information about the patch by calling the `GetPatchInformation` function(page 12-46).

Obtaining Information About a Patch

You use the functions described in this section to obtain a list of all the patches in a patch chain and to obtain information about a particular patch in a chain.

GetPatchesInPatchChain

Returns a list of patches in a patch chain.

```
OSStatus GetPatchesInPatchChain (PatchChainID thePatchChain,
                        ItemCountrequestedPatches,
                        ItemCount * totalPatches,
                        PatchID * thePatches);
```

`thePatchChain`	The ID of the patch chain.
`requestedPatches`	An integer specifying the size of the array in which this function stores patch IDs when it returns.
`totalPatches`	A pointer to the total number of patch IDs in the patch chain.
`thePatches`	A pointer to an array of patch IDs. On return, the `GetPatchesInPatchChain` function stores the patch IDs in the specified patch chain in this array.

DISCUSSION

The patches in the array referenced by the parameter `thePatches` are not listed in any particular order. The function returns the names of all the patches in the patch chain, whether they are enabled or not.

If you call this function and the value specified by the `requestedPatches` parameter is smaller than the value returned in the `totalPatches` parameter, this means that the array you have allocated for the patch ID information is too small. If you want to obtain more information, you must set the `requestedPatchChains` parameter to the desired value and then call the function again.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

Use the `GetPatchInformation` function (page 12-46) to get information about a particular patch in a patch chain.

Use the `GetPatchChainInformation` function (page 12-41) to find out the name of the routine being patched and the ID of the process that contains the patched routine.

GetPatchInformation

Returns information about a patch.

```
OSStatus GetPatchInformation (PatchID thePatchID,
                        PBVersion version,
                        PatchInformation *patchInfo);
```

`thePatchID`	The ID of the patch.
`version`	The value `kPatchInformationVersion`.
`patchInfo`	A pointer to a patch information structure. On return, the `GetPatchInformation` function fills in the fields of this structure.

DISCUSSION

The `GetPatchInformation` function returns the following information about a patch: the ID of the patch chain to which the patch belongs, the address of the patch routine, the name of the patch, the ordering constraints specified for the patch, and any options for the patch. Once a patch has been installed by the Patch Manager, any client that knows the ID of the patch can obtain patch information by calling the `GetPatchInformation` function.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

Use the `GetPatchChainsInKernelProcess` function (page 12-40) to obtain the IDs of all patch chains associated with a process. Then you can use the `GetPatchesInChain` function (page 12-45) to obtain the IDs of all patches in a chain.

Glossary

data-driven patching A method of patching according to which you create data structures specifying the address of your patch and of the patched routine, the name of your patch, and any other information required to execute your patch. Using this model, the task of installing a patch and ensuring the order in which it executes relative to other patches is taken over by the Code Fragment Manager and the Patch Manager. This is the preferred patching method in Mac OS 8. See also **programmatic patching**.

head patch A patch that does some processing before it calls the patched routine.

local patch A patch that executes only within a program's process. You create a local patch by including a patch description structure for it in a special fragment associated with your program's fragment.

global patch A patch that is called by every application referencing the patched routine. To create a global patch, you create a shared library that calls the patch and a special shared library fragment that contains the patch description structure for that patch. The shared library must use per-process instantiation.

owning fragment A fragment that defines a patch by using a patch description fragment. When the owning fragment is prepared for execution, the system looks for and installs the patches owned by the fragment as part of the fragment's launch process.

patch A piece of code that intercepts the transaction between a client and a service that is implemented by a single routine. You can assume control in order to monitor the use of that service, to modify the service, or to replace the service. A patch is uniquely identified by a patch ID.

patch chain An ordered list of patches on the same instance of the same entry point. A patch chain is uniquely identified by a patch chain ID.

patch chain ID A unique identifier that specifies a patch chain. The Patch Manager uses this identifier to distinguish among patch chains belonging to the same patch chain set.

patch chain set The set of patch chains associated with a single process .

patch description structure A data structure you create to describe a patch. It specifies the address of the patch, the address of the patched routine, the name of the patch, and options related to the execution of the patch. You store patch description structures in a special fragment that is associated either with the application (for local patching) or a shared library (for global patching).

patch ID A unique identifier that specifies a patch in a patch chain. The Patch Manager returns this ID to you after it installs a patch.

programmatic patching A method of patching used with System 7 according to which you replace the address of an existing operating system routine in the trap dispatch table with the address of a patch routine. This method is supported in Mac OS 8 but not in any Mac OS release following that. Compare with **data-driven patching**.

replacement patch A patch that never calls the routine it is patching; it simply replaces it.

surround patch A patch that performs some processing, calls the patched routine, and then performs some additional processing. See also **head patch** and **tail patch.**

tail patch A patch that does some processing after calling the patched routine