INSIDE MACINTOSH

Adding Multitasking Capability to Applications Using Multiprocessing Services

For Multiprocessing Services 2.0

April 30, 1999
Technical Publications

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Introduction

Multiprocessing Services is a technology that allows your application to create tasks that run independently on one or more microprocessors. For example, you can have your application perform graphical calculations while writing data to a hard drive. Unlike the cooperative model (such as used by the Thread Manager or the Mac OS Process Manager), Multiprocessing Services automatically divides processor time among available tasks so no particular task can "hog" the system. On computers with multiple microprocessors, you can actually perform multiple tasks simultaneously. This feature allows you to divide up time-intensive calculations among several microprocessors.

This document is a complete guide to Multiprocessing Services 2.0. This technology is available with Mac OS 8.6 and later, although some functions may work with earlier system software versions.

You should read this document if you want to add multitasking capability to Mac OS applications. This document assumes you are familiar with programming Macintosh computers. For more information about how the Mac OS handles applications in memory, see the documents *Inside Macintosh: Processes* and *Mac OS Runtime Architectures*.

This document covers Multiprocessing Services in the following chapters:

- "About Multitasking on the Mac OS" (page 13) describes the basics of multitasking and multiprocessing, as well as information about how Multiprocessing Services implements these capabilities on the Mac OS.
- "Using Multiprocessing Services" (page 29) contains programming examples and other detailed information about adding Multiprocessing Services to your application.
- "Multiprocessing Services Reference" (page 55) contains a complete programming reference, documenting the functions, data types, and constants available with Multiprocessing Services.

- Appendix A, “Calculating the Intertask Signaling Time,” contains sample code you can use to determine the amount of time it takes to notify a task.
- Appendix B, “Changes From Previous Versions of Multiprocessing Services,”describes changes and additions to the Multiprocessing Services API between version 1.4 and 2.0.
- Appendix C, “Document Version History,” describes changes to this document.

For additional information about Multiprocessing Services, you should check the Apple Developer Web site:

<http://www.apple.com/developer/>

CHAPTER 2

About Multitasking on the Mac OS

Contents

This chapter describes the basic concepts underlying multitasking and how Multiprocessing Services uses them on Macintosh computers.

You should read this chapter if you are not familiar with multitasking or multiprocessing concepts. Note that this chapter covers mostly concepts rather than implementation or programming details. For information about actually using the Multiprocessing Services API in your application, see "Using Multiprocessing Services" (page 29)

The topics covered in this chapter include the following:

Multitasking Basics

Multitasking is essentially the ability to do many things concurrently. For example, you may be working on a project, eating lunch, and talking to a colleague at the same time. Not everything may be happening simultaneously, but you are jumping back and forth, devoting your attention to each task as necessary.

In programming, a **task** is simply an independent execution path. On a computer, the system software can handle multiple tasks, which may be applications or even smaller units of execution. For example, the system may execute multiple applications, and each application may have independently executing tasks within it. Each such task has its own stack and register set.

Multitasking may be either cooperative or preemptive. **Cooperative multitasking** requires that each task voluntarily give up control so that other tasks can execute. An example of cooperative multitasking is an unsupervised group of children wanting to look at a book. Each child can theoretically get a chance to look at the book. However, if a child is greedy, he or she may spend

an inordinate amount of time looking at the book or refuse to give it up altogether. In such cases, the other children are deprived.

Preemptive multitasking allows an external authority to delegate execution time to the available tasks. Preemptive multitasking would be the case where a teacher (or other supervisor) was in charge of letting the children look at the book. He or she would assign the book to each child in turn, making sure that each one got a chance to look at it. The teacher could vary the amount of time each child got to look at the book depending on various circumstances (for example, some children may read more slowly and therefore need more time).

The Mac OS 8 operating system implements cooperative multitasking between applications. The Process Manager can keep track of the actions of several applications. However, each application must voluntarily yield its processor time in order for another application to gain it. An application does so by calling `WaitNextEvent`, which cedes control of the processor until an event occurs that requires the application's attention.

Multiprocessing Services allows you to create preemptive tasks within an application (or process). For example, you can create tasks within an application to process information in the background (such as manipulating an image or retrieving data over the network) while still allowing the user to interact with the application using the user interface.

Note
The definition of *task* in this document is analogous to the use of the term *thread* in some other operating systems such as UNIX®. In older documentation, Apple has sometimes referred to separate units of execution as *threads*. For example, the Thread Manager allows you to create cooperatively scheduled threads within a task. You should not confuse these threads with the preemptively scheduled tasks created by Multiprocessing Services. ◆

Multitasking and Multiprocessing

Multitasking and multiprocessing are related concepts, but it is important to understand the distinctions between then. Multitasking is the ability to handle several different tasks at once. **Multiprocessing** is the ability of a computer to use more than one processor simultaneously. Typically, if multiple processors

are available in a multitasking environment, then tasks can be divided among the processors. In such cases, tasks can run simultaneously. For example, if you have a large image that you need to manipulate with a filter, you can break up the image into sections, and then assign a task to process each section. If the user's computer contains multiple processors, several tasks can be executed simultaneously, reducing the overall execution time. If only one processor exists, then the tasks are preempted in turn to give each access to the processor.

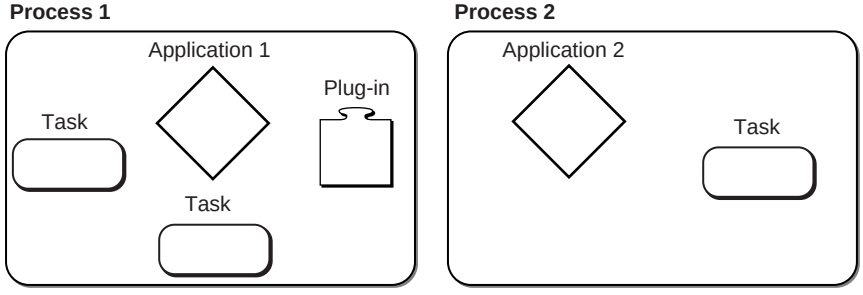
Note
Because multitasking allows an operating system to attend to several different operations at the same time, these operations may *appear* to occur simultaneously on even a single-processor system, due to the speed of the processor.
◆

Multiple processor support is transparent in Multiprocessing Services. If multiple processors exist, then Multiprocessing Services divides the tasks among all the available processors to maximize efficiency (a technique often called **symmetric multiprocessing**). If only one processor exists, then Multiprocessing Services simply schedules the available tasks with the processor to make sure that each task receives attention.

Tasks and Address Spaces

On the Mac OS, all applications are assigned a **process** or *application context* at runtime. The process contains all the resources required by the application, such as allocated memory, stack space, plug-ins, nonglobal data, and so on. Tasks created with Multiprocessing Services are automatically associated with the creating application's process, as shown in Figure 2-1.

Figure 2-1 Tasks within processes



All resources within a process occupy the same address space, so tasks created by the same application are free to share memory. For example, if you want to divide an image filtering operation among multiple identical tasks, you can allocate space for the entire image in memory, and then assign each task the address and length of the portion it should process.

IMPORTANT

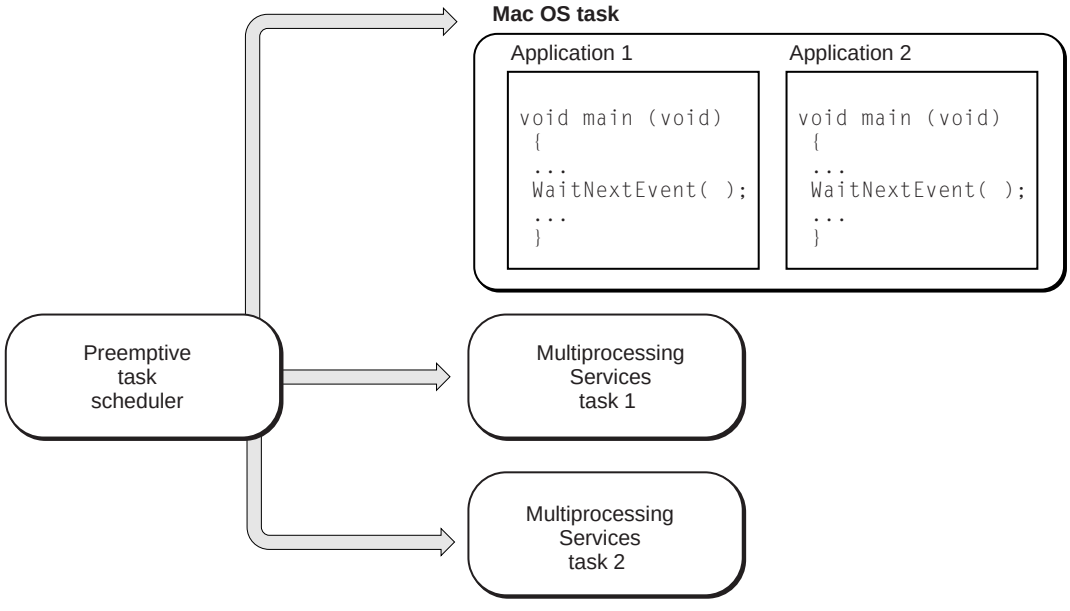
Although all processes share the same address space in Mac OS 8.6, you should not assume that this will remain the case; your application or task should not attempt to access data or code residing in another process. ▲

Task Scheduling

Multitasking environments require one or more **task schedulers**, which control how processor time (for one or more processors) is divided among available tasks. Each task you create is added to a task queue. The task scheduler then assigns processor time to each task in turn.

As seen by the Mac OS 8.6 task scheduler, all cooperatively multitasked programs (that is, all the applications that are scheduled by the Process Manager) occupy a single preemptive task called the **Mac OS task,** as shown in Figure 2-2.

Figure 2-2 The Mac OS task and other preemptive tasks



For example, if your cooperatively scheduled application creates a task, the task is preemptively scheduled. The application task (containing the main event loop) is not preemptively scheduled, but it resides within the Mac OS task, which *is* preemptively scheduled. Within the Mac OS task, the application must cooperatively share processor time with any other applications that are currently running.

A task executes until it completes, is blocked, or is preempted. A task is **blocked** when it is waiting for some event or data. For example, a task may need output from a task that has not yet completed, or it may need certain resources that are currently in use by another task.

A blocked task is removed from the task queue until it becomes eligible for execution. The task becomes eligible if either the event that it was waiting for occurs or the waiting period allowed for the event expires.

If the task does not complete or block within a certain amount of time (determined by the scheduler), the task scheduler preempts the task, placing it

at the end of the task queue, and gives processor time to the next task in the queue.

Note that if the main application task is blocked while waiting for a Multiprocessing Services event, the blocking application does not get back to its event loop until the event finally occurs. This delay may be unacceptable for long-running tasks. Therefore, in general your application should poll for Multiprocessing Services events from its event loop, rather than block while waiting for them. The task notification mechanisms described in "Shared Resources and Task Synchronization" (page 18) are ideal for this purpose.

Note
In the future, application tasks will run as individual preeemptive tasks, rather than within the Mac OS task. However, calls to nonreentrant Mac OS system software functions will cause the task to be blocked for the duration of the call, in a manner similar to a remote procedure call. See "Making Remote Procedure Calls" (page 47) for more information. ◆

Shared Resources and Task Synchronization

Although each created task may execute separately, it may need to share information or otherwise communicate with other tasks. For example, Task 1 may write information to memory that will be read by Task 2. In order for such operations to occur successfully, some synchronization method must exist to make sure that Task 2 does not attempt to read the memory until Task 1 has completed writing the data and until Task 2 knows that valid data actually exists in memory. The latter scenario can be an issue when using multiple processors, because the PowerPC™ architecture allows for writes to memory to be deferred. In addition, if multiple tasks are waiting for another task to complete, synchronization is necessary to ensure that only one task can respond at a time.

Multitasking environments offer several ways for tasks to coordinate and synchronize with each other. The sections that follow describe three **notification mechanisms** (or signaling mechanisms) that allow tasks to pass information between them, and one task sharing method.

Note that the time required to perform the work in a given request should be much more than the amount of time it takes to communicate the request and its results. Otherwise, delegating work to tasks may actually reduce overall performance. Typically the work performed should be greater than the intertask signalling time (20-50 microseconds).

Note that you should avoid creating your own synchronization or sharing methods, because they may work on some Mac OS implementations but not on others.

Semaphores

A **semaphore** is a single variable that can be incremented or decremented between zero and some specified maximum value. The value of the semaphore can communicate state information. A mail box flag is an example of a semaphore. You raise the flag to indicate that a letter is waiting in the mailbox. When the mailman picks up the letter, he lowers the flag again. You can use semaphores to keep track of how many occurrences of a particular thing are available for use.

Binary semaphores, which have a maximum value of one, are especially efficient mechanisms for indicating to some other task that something is ready. When a task or application has finished preparing data at some previously agreed to location, it raises the value of a binary semaphore that the target task waits on. The target task lowers the value of the semaphore, performs any necessary processing, and raises the value of a different binary semaphore to indicate that it is through with the data.

Semaphores are quicker and less memory intensive than other notification mechanisms, but due to their size they can convey only limited information.

Message Queues

A **message queue** is a collection of data (messages) that must be processed by tasks in a first-in, first-out order. Several tasks can wait on a single queue, but only one will obtain any particular message. Messages are useful for telling a task what work to do and where to look for information relevant to the request being made. They are also useful for indicating that a given request has been processed and, if necessary, what the results are.

Typically a task has two message queues, one for input and one for output. You can think of message queues as In boxes and Out boxes. For example, your In

box at work may contain a number of papers (messages) indicating work to do. After completing a job, you would place another message in the Out box. Note that if you have more than one person assigned to an In box/Out box pair, each person can work independently, allowing data to be processed in parallel.

In Multiprocessing Services, a message is 96-bits of data that can convey any desired information.

Message queues incur more overhead than the other two notification mechanisms. If you must synchronize frequently, you should try to use semaphores instead of message queues whenever possible.

Event Groups

An **event group** is essentially a group of binary semaphores. You can use event groups to indicate a number of simple events. For example, a task running on a server may need to be aware of multiple message queues. Instead of trying to poll each one in turn, the server task can wait on an event group. Whenever a message is posted on a queue, the poster can also set the bit corresponding to that queue in the event group. Doing so notifies the task, and it then knows which queue to access to extract the message. In Multiprocessing Services, an event group consists of thirty-two 1-bit flags, each of which may be set independently. When a task receives an event group, it receives all 32-bits at once (that is, it cannot poll individual bits), and all the bits in the event group are subsequently cleared.

Critical Regions

In addition to notification mechanisms, you can also specify **critical regions** in a multitasking environment. A critical region is a section of code that can be accessed by only one task at a time. For example, say part of a task's job is to search a data tree and modify it. If multiple tasks were allowed to search and try to modify the tree at the same time, the tree would quickly become corrupted. An easy way to avoid the problem is to form a critical region around the tree searching and modification code. When a task tries to enter the critical region it can do so only if no other task is currently in it, thus preserving the integrity of the tree.

Critical regions differ from semaphores in that critical regions can handle recursive entries and code that has multiple entry points. For example, if three functions `func1`, `func2`, and `func3` access some common resource (such as the

tree described above), but never call each other, then you can use a semaphore to synchronize access. However, suppose `func3` calls `func1` internally. In that case, `func3` would obtain the semaphore, but when it calls `func1`, it will deadlock. Synchronizing using a critical region instead allows the same task to enter multiple times, so `func1` can enter the region again when called from `func3`.

Because critical regions introduce forced linearity into task execution, improper use can create bottlenecks that reduce performance. For example, if the tree search described above constituted the bulk of a task's work, then the tasks would spend most of their time just trying to get permission to enter the critical region, at great cost to overall performance. A better approach in this case might be to use different critical regions to protect subtrees. You can then have one task search one part of the tree while others were simultaneously working on other parts of the tree.

Tasking Architectures

Determining how to divide work into tasks depends greatly on the type of work you need to do and how the individual tasks rely on each other.

For a computer running multiple processors, you should optimize your multitasking application to keep them as busy as possible. You can do so by creating a number of tasks and letting the task scheduler assign them to available processors, or you can query for the number of available processors and then create enough tasks to keep them all busy.

A simple method is to determine the number of processors available and create as many tasks as there are processors. The application can then split the work into that many pieces and have each processor work on a piece. The application can then poll the tasks from its event loop until all the work is completed.

IMPORTANT
Even if only one processor exists, you should create preemptive tasks to handle faceless computations (filtering, spellchecking, background updating, and so on). Doing so gives the task scheduler more flexibility in assigning processor time, and it will also scale transparently if multiple processors are available. The application should do all the work only if Multiprocessing Services is not available. ▲

The sections that follow describe several common tasking architectures you can use to divide work among multiple processors. You might want to combine these approaches to solve specific problems or come up with your own if none described here are appropriate.

Multiple Independent Tasks

In many cases, you can break down applications into different sections that do not necessarily depend on each other but would ideally run concurrently. For example, your application may have one code section to render images on the screen, another to do graphical computations in the background, and a third to download data from a server. Each such section is a natural candidate for preemptive tasking. Even if only one processor is available, it is generally advantageous to have such independent sections run as preemptive tasks. The application can notify the tasks (using any of the three notification mechanisms) and then poll for results within its event loop.

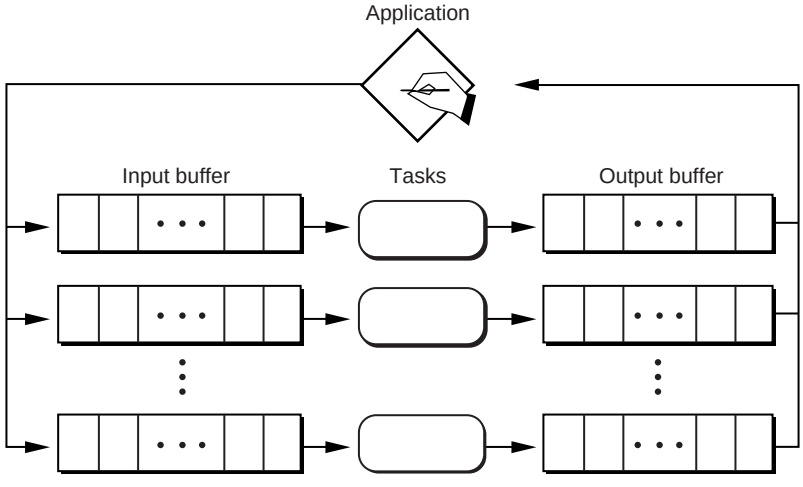
Parallel Tasks With Parallel I/O Buffers

If you can divide the computational work of your application into several similar portions, each of which takes about the same amount of time to complete, you can create as many tasks as the number of processors and divide the work evenly among the tasks (“divide and conquer”). An example would be a filtering task on a large image. You could divide the image into as many equal sections as there are processors and have each do a fraction of the total work.

This method for using Multiprocessing Services involves creating two buffers per task: one for receiving work requests and one for posting results. You can create these buffers using either message queues or semaphores.

As shown in Figure 2-3, the application splits the data evenly among the tasks and posts a work request, which defines the work a task is expected to perform, to each task's input buffer. Each task asks for a work request from its input buffer, and blocks if none is available. When a request arrives, the task performs the required work and then posts a message to its output buffer indicating that it has finished and providing the application with the results.

Figure 2-3 Parallel tasks with parallel I/O buffers



Parallel Tasks With a Single Set of I/O Buffers

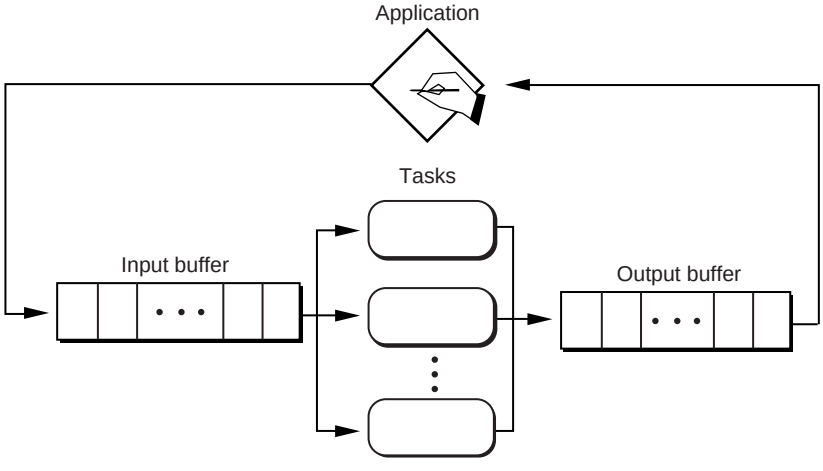
If you can divide the computational work of your application into portions that can be performed by identical tasks but you can't predict how long each computation will take, you can use a single input buffer for all your tasks. The application places each work request in the input buffer, and each free task asks for a work request. When a task finishes processing the request, it posts the result to a single output buffer shared by all the tasks and asks for a new request from the input buffer. This method is analogous to handling a queue of customers waiting in a bank line. There is no way to predict which task will process which request, and there is no way to predict the order in which results will be placed in the output buffer. For this reason, you might want to have the

task include the original work request with the result so the application can determine which result is which.

As in the "divide and conquer" architecture, the application can check events, control data flow, and perform some of the calculations while the tasks are running.

Figure 2-4 illustrates this "bank line" tasking architecture.

Figure 2-4 Parallel tasks with a single set of I/O buffers



Sequential Tasks

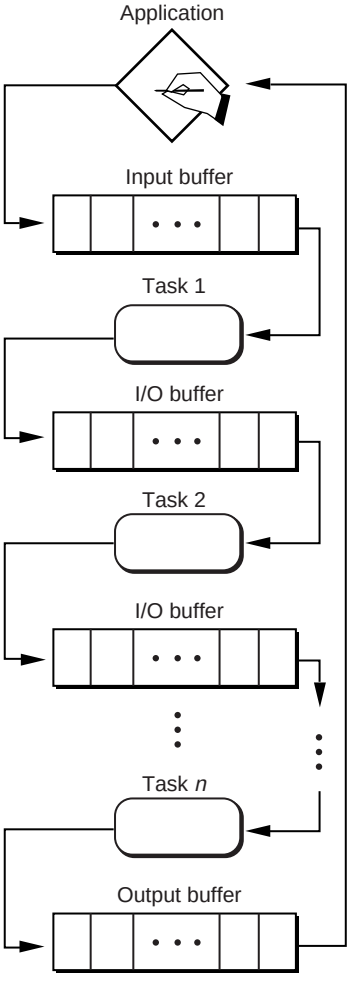
For some applications, you may want to create several different tasks, each of which performs a different function. Some of these tasks might be able to operate in parallel, in which case you can adapt the "divide and conquer" model to perform the work. However, sometimes the output of one task is needed as the input of another. For example a sound effects application may need to process several different effects in sequence (reverb, limiting, flanging, and so on). Each task can process a particular effect, and the output of one task would feed into the input of the next. In this case, a sequential or "pipeline" architecture is appropriate.

Note that a sequential task architecture benefits from multiple processors only if several of the tasks can operate at the same time; that is, new data must be

entering the pipeline while other tasks are processing data further down the line. It is harder with this architecture than with parallel task architectures to ensure that all processors are being used at all times, so you should add parallel tasks to individual stages of the pipeline whenever possible.

As shown in Figure 2-5, the sequential task architecture requires a single input buffer and a single output buffer for the pipeline, and intermediate buffers between sequential tasks.

Figure 2-5 Sequential tasks



CHAPTER 3

Using Multiprocessing Services

Contents

This chapter describes how to incorporate Multiprocessing Services into your Mac OS application. You should read this chapter if you are interested in adding preemptive tasks to your application.

The topics covered in this chapter include:

Note
This document describes version 2.0 of Multiprocessing Services. For a list of functions changed or added between versions 1.4 and 2.0, see Appendix B. Applications built using older versions of Multiprocessing Services can execute without modification under version 2.0. ◆

IMPORTANT
Preemptive tasks cannot execute 68K code. If you must call 68K code in a task, you must do so through a remote procedure call as described in "Making Remote Procedure Calls" (page 47). ▲

Multiprocessing Services in Mac OS 8 and Mac OS X

Multiprocessing Services 2.0 runs on system software Mac OS 8.6 and later. All PowerPC Macintosh computers are supported except for 6100/7100/8100 and 5200/6200 series computers.

Note
Multiprocessing Services 1.0 functions can run on System 7.5.2 and later if the Multiprocessing Services 1.x shared library is available. Pre-2.0 versions of the library were installed as part of system software for Mac OS 8 through Mac OS 8.5 but must be explicitly installed for earlier releases. The 1.x versions of the Multiprocessing Services library can run on all PowerPC Macintosh computers. ◆

Unlike earlier versions of Multiprocessing Services, you can create and execute preemptive tasks with virtual memory turned on.

Note that under Mac OS 8.6, Multiprocessing Services allows your application to create preemptive tasks within your application's process (or execution context). However, the individual applications are still cooperatively scheduled by the Process Manager. In Mac OS X, both applications and tasks created by applications will be preemptively scheduled. Multiprocessing Services is Carbon-compliant, so applications built following the Carbon specification can run transparently on both Mac OS 8 and Mac OS X systems.

Criteria for Creating Tasks

Although you can in theory designate almost any type of code as a task, in practice you should use the following guidelines to make best use of the available processors and to avoid unnecessary bottlenecks.

- Tasks should generally perform faceless, calculation-intensive processing.
- The work performed by a task should be substantially more than the time required to process the request and result notifications. If it takes much longer to notify a task and retrieve results than to execute the task itself, an

application's performance will be dramatically worse with multiprocessing. Assuming a typical intertask signaling time is 20-50 microseconds, your tasks should take at least 200-500 microseconds to execute. If you want to explicitly calculate the intertask signaling time, you can use the sample code provided in Appendix A.

- If your task needs to allocate memory, you must allocate the memory prior to signaling the task, or use the function `MPAllocateAligned` (page 94).
- Tasks should not call 68K code. The 68K emulator runs only cooperatively within the Mac OS task, and not within any preemptive task. If you must call 68K code, you can do so using a remote procedure call. See "Making Remote Procedure Calls" (page 47) for more information.
- Tasks should not call Mac OS system software functions except through remote procedure calls. See "Making Remote Procedure Calls" (page 47) for more information.
- Tasks should not access low-memory global data. A task may be executing at any time, including when applications that did not create them are running.
- Tasks should not call into unknown code. If you allow third parties to specify a callback function, you should never call that function from a task, since you cannot control what the callback will do. Calling back into nonreentrant code could easily corrupt data or cause a system crash.
- Avoid global variables. The main cause of nonreentrancy is the manipulation of global data. Tasks that manipulate global variables, global states, or buffers pointed to by global variables must use synchronization techniques to prevent other tasks from attempting to do so at the same time. Read-only global data are allowed.
- Do not call any Multiprocessing Services functions at interrupt time unless you are signaling a notification mechanism. See "Notifying Tasks at Interrupt Time" (page 44) for more information.

Checking for the Availability of Multiprocessing Services

You should always determine the availability of Multiprocessing Services before attempting to call any of its functions. If you are programming in C, you should do so by calling the Boolean macro `MPLibraryIsLoaded`. A return value of true indicates that the Multiprocessing Services library is present and available

for use. Listing 3-1 (page 33) in "Creating Tasks" shows an example of using the `MPLibraryIsLoaded` macro.

Note
For historical reasons, the Multiprocessing Services shared library may be prepared by the Code Fragment Manager and yet still be unusable; checking for resolved imported symbols is not enough to ensure that the functions are available. Therefore, you must always check for the presence of Multiprocessing Services by using the `MPLibraryIsLoaded` macro. ◆

You probably want your application to run even if Multiprocessing Services is not available, so you should specify a weak link to the Multiprocessing Services shared library. Doing so allows your application to run even if the shared library is not present.

Determining the Number of Processors

You may want to determine the number of processors available on the host computer before creating any tasks. Typically, you would create one task per processor; even if only one processor is present, it is generally more efficient to assign faceless work to a task and have the cooperatively scheduled main application handle only user interaction.

Multiprocessing Services uses two functions to determine the number of processors. The function `MPProcessors` (page 56) returns the number of physical processors available on the host computer. The function `MPProcessorsScheduled` (page 57) returns the number of active processors available (that is, the number that are currently available to execute tasks). The number of active processors may vary over time (due to changing priorities, power consumption issues, and so on).

Creating Tasks

After determining how many processors are available, you can go ahead and create tasks for your application.

Each task must be a function that takes one 32-bit parameter and returns a 32-bit result of type `OSStatus` when it finishes. The 32-bit input parameter can be any information that the task needs to perform its function. Some examples of input are

- a message queue ID that indicates which queue the task should go to for information
- a pointer to a structure containing data to process
- a pointer to a C++ object
- a pointer to a task-specific block of memory through which the application can communicate information for the life of the task

You create a task by calling the function `MPCreateTask` (page 58). The code in Listing 3-1 shows how you can create a number of identical tasks. Identical tasks can be useful when you want to divide up a large calculation (such as a image filtering operation) among several processors to improve performance.

Listing 3-1 Creating tasks

```
#define kMPStackSize 0 // use default stack size
#define kMPTaskOptions 0 // use no options

typedef struct {
    long firstThing;
    long totalThings;
    } sWorkParams, *sWorkParamsPtr;

typedef struct {
    MPTaskID taskID;
    MPQueueID requestQueue;
    MPQueueID resultQueue;
    sWorkParams params;
    } sTaskData, *sTaskDataPtr;

sTaskDataPtr myTaskData;
UInt32 numProcessors;
MPQueueID notificationQueue;

void CreateMPTasks( void ) {
```

```
    OSErr theErr;
    UInt32 i;

    theErr = noErr;

    /* Assume single processor mode */
    numProcessors = 1;

    /* Initialize remaining globals */
    myTaskData = NULL;
    notificationQueue = NULL;

    /* If the library is present then create the tasks */
    if( MPLibraryIsLoaded() ) {
        numProcessors = MPProcessorsScheduled();
        myTaskData = (sTaskDataPtr)NewPtrClear
                ( numProcessors * sizeof( sTaskData ) );
        theErr = MemError();
        if( theErr == noErr  )
                theErr = MPCreateQueue( &notificationQueue );
        for( i = 0; i < numProcessors && theErr == noErr; i++ ) {
            if( theErr == noErr )
                    theErr = MPCreateQueue(&myTaskData[i].requestQueue );
            if( theErr == noErr )
                    theErr = MPCreateQueue(&myTaskData[i].resultQueue );
            if( theErr == noErr )
                    theErr = MPCreateTask( MyTask, &myTaskData[i],
                        kMPStackSize, notificationQueue,
                        NULL, NULL, kMPTaskOptions,
                        &myTaskData[i].taskID );
            }
        }

/* If something went wrong, just go back to single processor mode */
    if( theErr != noErr ) {
        StopMPTasks();
        numProcessors = 1;
        }
    }
```

The sTaskData structure defines a number of values to be used with the task, such as the task ID, the IDs of the message queues used with the task, and a pointer to parameters to pass to the task. A pointer to a structure of this type is passed in the function MPCreateTask (page 58).

The variable notificationQueueID holds the ID of the notification queue to associate with the tasks. When a task terminates, it sends a message to this queue. After sending a termination request, the application typically polls this queue to determine when the task has actually terminated.

The CreateMPTasks function creates as many identical tasks as there are available processors (as stored in numProcessors). If for some reason the tasks cannot be created (for example, if Multiprocessing Services is not available), the variable numProcessors is set to 1 and the application should do the work of the tasks itself without making any Multiprocessing Services calls.

Before creating the tasks, CreateMPTasks calls the function MPCreateQueue (page 65) to create a notification queue to be used by all the tasks. It then calls the Memory Manager function NewPtrClear to allocate memory for all the myTaskData structures required (in the case of this example, one per task).

Next, CreateMPTasks iterates over the number of requested tasks. For each iteration, it does the following:

- Makes two calls to MPCreateQueue (page 65) to create a request queue and a result queue for each task. The IDs for these queues are stored in the task's myTaskData structure.
- Fills out the myTaskData structure for that task as necessary.
- Calls the function MPCreateTask (page 58). When calling this function, you must specify the following values:
  - the entry point of the task and its input parameters
  - the size of the stack to associate with the task
  - the notification queue to associate with the task (by passing the ID of the queue obtained in the function MPCreateQueue (page 65).)

Each task is assigned its own unique ID, which is passed back in the taskID field of the myTaskData task structure.

Although not a requirement, you can assign a relative weight to each task by calling the function MPSetTaskWeight (page 63). The task weight is a value that indicates the amount of processor attention to give this task relative to all other eligible tasks. If, as in this example, you create a number of identical tasks, each would probably be given equal weight.

The sample task in Listing 3-2 calls one of two different functions depending on the request is placed on its queue.

Listing 3-2 A sample task

```
#define kMyRequestOne                   1
#define kMyRequestTwo                   2

#define kMyResultException      -1

OSStatus MyTask( void *parameter ) {

    OSErr theErr;
    sTaskDataPtr p;
    Boolean finished;
    UInt32 message;

    theErr = noErr;

    /* Get a pointer to this task's unique data */
    p = (sTaskDataPtr)parameter ;

    /* Process each request handed to the task and return a result */
    finished = false;
    while( !finished ) {
        theErr = MPWaitOnQueue( p->requestQueue, (void **)&message,
                NULL, NULL, kDurationForever );
        if( theErr == noErr ) {
/* Pick a function to call and pass in the parameters.  */
/* The parameters should be set up prior to sending the */
/* message just received. Note that we could also just  */
/* pass in a pointer to the desired function instead of */
/* using a selector.
*/
            switch( message ) {
                case kMyRequestOne:
                    theErr = fMyTaskFunctionOne( &p->params );
                    break;
                case kMyRequestTwo:
                    theErr = fMyTaskFunctionTwo( &p->params );
```

```
                break;
            default:
                finished = true;
                theErr = kMyResultException;
            }
        MPNotifyQueue( p->resultQueue, (void *)theErr, NULL, NULL );
        }
    else
        finished = true;
    }

/* Task is finished now */
return( theErr );
}
```

This task takes one parameter, a pointer to its task data structure. This structure contains all the information that is needed for the life of the task, such as the request and result queues created for it, and any input necessary when processing a task request. The input parameters are passed along to the requested function.

After some initialization, the task sets the `finished` flag to `false` and then spends the rest of its time in a `while` loop processing message requests. The task calls the function `MPWaitOnQueue` (page 68), which waits indefinitely until a message appears on its request queue. In this case, the message indicates which function the task is to call. When a message is received, `MyTask` checks the request message to determine which function is desired and calls through to that function. Upon return, it posts a message on the result queue by calling `MPNotifyQueue` (page 66) and then calls `MPWaitOnQueue` again to wait for the next message.

Note that if you are creating tasks on-the-fly, you may want to have your task dispose of its task record (pointed to by `p`) upon completion of the task. For more information about allocating and disposing of memory in tasks, see "Allocating Memory in Tasks" (page 45).

Terminating Tasks

When you want to terminate a task, you should call the function `MPTerminateTask` (page 60). Doing so deletes the task, but you are still responsible for disposing of any memory you may have allocated for the task. In addition, because the tasks run asynchronously, the task may not actually terminate until sometime after the `MPTerminateTask` function returns. Therefore, you should not assume that the task has terminated until you have received a termination message from the notification queue you specified in the function `MPCreateTask` (page 58). Listing 3-3 shows how you might terminate the tasks created in Listing 3-1 (page 33).

Listing 3-3 Terminating tasks

```
void StopMPTasks(void)
{
    UInt32 i;

    if (myTaskData != NULL)
    {
        for (i = 0; i < numProcessors; i++)
        {
            if (myTaskData[i].TaskID != NULL)
            {
                MPTerminateTask(myTaskData[i].TaskID, noErr);
                MPWaitOnQueue(notificationQueue, NULL, NULL, NULL,
                    kDurationForever);
            }

            if (myTaskData[i].fRequestQueue != NULL)
                MPDeleteQueue(myTaskData[i].RequestQueue);
            if (myTaskData[i].fResultQueue != NULL)
                MPDeleteQueue (myTaskData[i].ResultQueue);
        }

        if (notificationQueue != NULL)
        {
```

```
            MPDeleteQueue (notificationQueue);
            notificationQueue = NULL;
        }

        DisposePtr((Ptr)myTaskData);
        myTaskData = NULL;
    }
}
```

The `StopMPTasks` function iterates through all the task data structures that were created in `CreateMPTasks` and checks for those with valid task IDs. It then calls the function `MPTerminateTask` (page 60) for each valid task ID.

After making the termination call, `StopMPTasks` then waits for a message to appear on the notification queue indicating that the task has in fact been terminated. It does so by waiting continuously on the notification queue until the termination message arrives. It then clears the task ID and disposes of the queues allocated for the task.

Note
If you call `MPWaitOnQueue` (page 68) from a cooperative task, you should specify only the `kDurationImmediate` wait time. You must use a `while` loop that continuously calls `MPWaitOnQueue` (page 68) until the termination message appears in the notification queue. Doing so also allows you to process events that may occur between calls. For additional information, see "Synchronizing and Notifying Tasks" (page 39). ◆

After terminating all the existing tasks, `StopMPTasks` then deletes the notification queue and disposes of the task data structures.

Synchronizing and Notifying Tasks

As described in "About Multitasking on the Mac OS" (page 13) tasks often need to coordinate with the main application or with other tasks to avoid data corruption or synchronization problems. To coordinate tasks, Multiprocessing Services provides three notification mechanisms (semaphores, event groups, and message queues) as well as critical regions.

Of the three notification mechanisms, message queues are the easiest to use, but they are the slowest. Typically a task has two message queues associated with it. It takes messages off an input queue, processes the information accordingly, and, when done, posts a message to an output queue.

IMPORTANT

You should never use only one instance of a notification mechanism to convey both input and output information, because doing so can easily cause confusion. For example, after posting a request, an application will at some point start waiting for results. If it waits on the same mechanism where the request was posted, the request itself may appear to be the result. The application may then clear the request in the mistaken belief that it was a result and no actual work gets done. ▲

Before notifying a task, your application should make sure that everything the task needs is in memory. That is, you should have created any necessary queues and allocated space for any data the task may require. For each task, your application establishes the parameters of the work that it wants the task to perform and then it must signal the task through either a queue or a semaphore to begin performing that work. The specific work that the task is to perform can be completely defined within a message, or possibly within a block of memory reserved for that task. You can also pass in a pointer to the function that the task should call to perform the work. Doing so allows one task to perform many different types of chores.

Listing 3-4 shows a function that divides up a large amount of data among multiple tasks, placing requests on each task's request queue and waiting for the results.

Listing 3-4 Assigning work to tasks

```
OSErr NotifyTasks( UInt32 realFirstThing, UInt32 realTotalThings ) {

    UInt32 i;
    OSErr theErr;
    UInt32 thingsPerTask;
    UInt32 message;
    sWorkParams appData;
```

```
    theErr = noErr;

    thingsPerTask = realTotalThings / numProcessors;

    /* Start each task working on a unique piece of the total data */
    for( i = 0; i < numProcessors; i++ ) {
        myTaskData[i].params.firstThing =
                                    realFirstThing + thingsPerTask * i;
        myTaskData[i].params.totalThings = thingsPerTask;
        message = kMyRequestOne;
        MPNotifyQueue( myTaskData[i].requestQueue, (void *)message,
            NULL, NULL );
        }

    /* Now wait for the tasks to finish */
    for( i = 0; i < numProcessors; i++ )
        MPWaitOnQueue( myTaskData[i].resultQueue, (void **)&message,
            NULL, NULL, kDurationForever );

    return( theErr );
}
```

For each task, it calls `MPNotifyQueue` (page 66) to place the pointer to the task's portion of the data on the task's request queue. It then calls `MPWaitOnQueue` (page 68) to wait for confirmation that the task has completed.

Note
A message queue message is passed to the queue as three 32-bit parameters. Because the message in Listing 3-4 is only 32-bits long, the remaining two parameters are set to `NULL`. ◆

If you want to use semaphores or event groups instead of message queues, you would call the following functions to set up, notify, and wait on them, in a manner similar to that shown in Listing 3-4:

- `MPCreateSemaphore` (page 71), `MPSignalSemaphore` (page 72), `MPWaitOnSemaphore` (page 73), and `MPDeleteSemaphore` (page 72) for semaphores
- `MPCreateEvent` (page 74), `MPSetEvent` (page 76), `MPWaitForEvent` (page 77), and `MPDeleteEvent` (page 75) for event groups

However, if you use the simpler notification mechanisms, you have to find another way to pass the function pointer to the task. One possibility is to assign the pointer to a field in the task's task data structure.

Note that the example in Listing 3-4 will wait forever (`kDurationForever`) for a message to appear on its result queue. While this method is fine if called from a preemptive task, it can cause problems if called from a cooperative task. If the task takes a significant amount of time to execute, the calling task "hangs" for that time, since it can't call `WaitNextEvent` to give other applications processor time. If you want to wait on a task from a cooperative task, your application should post the message and then return to its event loop. From within the event loop it can then poll the result queue using `kDurationImmediate` waits until a message appears.

If you specify `kDurationImmediate` for the waiting time for either `MPWaitOnQueue` (page 68), `MPWaitOnSemaphore` (page 73), `MPWaitForEvent` (page 77), or `MPEnterCriticalRegion` (page 79), the function always returns immediately. If the return value is `kMPTimeoutErr`, then the task generated no new results since the last time the application checked. That is, no message was available, the semaphore was zero, or the critical region was being executed by another processor. If the value is `noErr`, a result was present and obtained by the call.

Handling Periodic Actions

You can use notification mechanisms to do more than simply signal tasks. For example, Listing 3-5 shows a task that uses a semaphore to do periodic actions.

Listing 3-5 Using a semaphore to perform periodic actions

```
void MyTask(void) {

    MPSemaphoreID delay;

    MPCreateSemaphore(1, 0, &delay); // a binary semaphore
    while(true)
        {
            DoIt();    // do something interesting
            (void) MPWaitOnSemaphore(delay, 10 * kDurationMillisecond);
            }
    }
```

This example uses a semaphore solely to create a delay. After each call to the `DoIt` function, `MyTask` waits for a notification that never arrives and times out after 10ms.

You can combine the delaying and notification aspects of a semaphore to add more flexibility as shown in Listing 3-6.

Listing 3-6 Performing actions periodically and on demand

```
main(void) {

    MPSemaphoreID delay;
    …
    MPCreateSemaphore(2, 0, &delay);
    MPCreateTask(…);

    while(true) {
        // Event loop.

        if ( /* something important happened */ )
            {
                MPSignalSempahore(delay);
                }

            }
}

void MyTask(void) {

    while(true) {
        DoIt();    // Do interesting things.
        (void) MPWaitOnSemaphore(work, 100 * kDurationMillisecond);
        }
    }
```

In this example, the `MyTask` task runs essentially as before, except that the main application creates the semaphore. If no signal is sent to the semaphore, the `DoIt` function in `MyTask` executes every 100ms. However, in this example the application can signal the semaphore, which unblocks the task and allows the

DoIt function to execute. That is, the DoIt function executes whenever the application signals the semaphore, or every 100ms otherwise.

Notifying Tasks at Interrupt Time

If you want to send a notification to a task from a 68K-style interrupt handler, you can do so using the functions MPSignalSemaphore (page 72), MPSetEvent (page 76), or MPNotifyQueue (page 66). The MPSignalSemaphore and MPSetEvent functions are always interrupt-safe, while the MPNotifyQueue (page 66) function becomes interrupt-safe if you reserve notifications on the message queue. See the MPSetQueueReserve (page 69) function description for more information about reserving notifications.

▲ **WARNING**
Aside from these three notification functions, only MPCurrentTaskID (page 62) and MPYield (page 62) are interrupt-safe; attempting to call other Multiprocessing Services functions at interrupt time, or at a deferred-task time, may cause a system crash. ▲

Using Critical Regions

If your tasks need access to code that is nonreentrant, (that is, only one task can be executing the code at any particular time), you must designate that code as being a critical region. You do so by calling the function MPCreateCriticalRegion (page 78). Doing so returns a critical region ID that you use to identify the region when you want to enter or exit it later. To enter a critical region, the task must call MPEnterCriticalRegion (page 79) and specify the ID of the region to enter. This function acts much like the functions that wait on message queues and semaphores; if the critical region is not currently available, the task can wait for a specified time for it to become available (after which it will time out).

After the task has completed using the critical region, you must call MPExitCriticalRegion (page 81). Doing so "frees" the critical region so that another task that is waiting on it can enter. Note that a task can call MPEnterCriticalRegion multiple times during execution (as in a recursive call) as long as it balances each such call with MPExitCriticalRegion when it leaves the critical region.

Note that the area of code designated as a critical region is not "tagged" as such in any way. You must make sure that your code is synchronized to properly

isolate the critical region. For example, if you have a critical region that will be shared by two different tasks, you must create the critical region outside the tasks that will require it and pass the critical region ID to the tasks. This method ensures that, even if multiple instances of a task were created and running, only one could access a particular critical region at a time.

If a task contains more than one critical region, each critical region must have its own unique ID; otherwise, a task entering a critical region may block another task from entering a different critical region.

Allocating Memory in Tasks

If you need to allocate a block of memory for a task, you must call the function `MPAllocateAligned` (page 94). Doing so returns a pointer to allocated memory with the alignment you specify. You should always use the Multiprocessing Services memory allocation functions if your task needs to allocate, deallocate, or otherwise manipulate memory. For example, if your task deallocates its task data structure after it has finished processing, it must call `MPFree` (page 95). Note however, that since the memory is being deallocated by a preemptive task, you must have initially allocated the task record by calling `MPAllocateAligned`, even if this allocation didn't occur in a preemptive task.

Using Task-Specific Storage

Task-specific storage is useful for storing small pieces of data, such as pointers to task-specific information. For example, if you create several identical tasks, each of which requires some unique data, you can store that data as task-specific storage. Task-specific storage locations are cross-referenced by an index value and the task ID, so the same code can easily refer to "per-instance" variables. Each such storage location holds a 32-bit value.

Task-specific storage is automatically allocated when a task is created; the amount is fixed and cannot change for the life of the task. To access the task-specific storage, you call the function `MPAllocateTaskStorageIndex` (page 90). Doing so returns an index number which references a storage location in each available task in the process. Subsequent calls to `MPAllocateTaskStorageIndex` return new task index values to access more of the

task-specific storage. Note that, aside from the fact that each index value is unique, you should not assume anything about the actual values of the index. For example, you cannot assume that successive calls to `MPAllocateTaskStorageIndex` will monotonically increase the index value.

Since the amount of task-specific storage is fixed, you may use up the available storage (and corresponding index values) if you make many `MPAllocateTaskStorageIndex` calls. In such cases, further calls to `MPAllocateTaskStorageIndex` return an error indicating insufficient resources.

You call `MPSetTaskStorageValue` (page 91) and `MPGetTaskStorageValue` (page 92) to set and retrieve the storage data. After you are finished using the storage locations, you must call `MPDeallocateTaskStorageIndex` (page 91) to free the index.

Using Timers

On occasion you may want to use timers in your preemptive tasks. For example, say you want a task to send a message to a given queue every 20 milliseconds. To do so, you can set a timer to block your task for 20ms after sending the notification by calling the function `MPDelayUntil` (page 82).

Note
Note that in some cases you may want to use notification mechanisms to accomplish periodic actions, as described in "Handling Periodic Actions" (page 42). ◆

In addition, you can create timers that will signal a specified notification mechanism after the timer expires. For example, say you have a task that is prompting the user to enter a name and password. Once you bring up the input dialog box, you may have another task (or the application) create a timer object to expire after five minutes. If the user has not entered a password during those five minutes, the timer expires and sends a message to the task, signaling that it should terminate.

You create a timer using the function `MPCreateTimer` (page 82) and arm it by calling the function `MPArmTimer` (page 86). To specify the notification mechanisms to signal when the timer expires, you call the function `MPSetTimerNotify` (page 84). Note that you can signal one notification

mechanism of each type if desired. For example, the timer can send a message to a queue and also set a bit in an event group when it expires.

The timers in Multiprocessing Services use time units of type `AbsoluteTime`, which increases monotonically since the host computer was started up. You can obtain the time since startup by calling the function `UpTime` (page 88). Multiprocessing Services also provides the functions `DurationToAbsolute` (page 88) and `AbsoluteToDuration` (page 89)which let you convert time between units of `AbsoluteTime` and units of type `Duration`. Note that you should not make any assumptions about what the `AbsoluteTime` units are based upon.

Making Remote Procedure Calls

At times a preemptive task may need to call a system software function, and doing so may cause problems. For example, many calls to Mac OS system software manipulate global variables, so data could easily be corrupted if more than one task attempts to make similar calls. To work around this problem, Multiprocessing Services allows you to make **remote procedure calls** if you need to call system software from a preemptive task. A remote procedure call also allows your task to call 68K code.

IMPORTANT

With the exception of functions in Multiprocessing Services, you cannot safely call Mac OS system software functions directly from a preemptive task. Even if some system software function appears to work today when called from a preemptive task, unless explicitly stated otherwise there is no guarantee that subsequent versions of the same routine will continue to work in future versions of system software. In Mac OS 8 implementations of Multiprocessing Services, the only exceptions to this rule are the atomic memory operations (such as `AddAtomic`) exported in the `InterfaceLib` shared library. Even these functions may switch to 68K mode if the operands to them are not properly aligned. If you need to access system software from a preemptive task, you must do so using the `MPRemoteCall` (page 106) function. ▲

To make a remote procedure call, you must designate an application-defined function that will make the actual calls to system software. You then pass a pointer to this function as well as any required parameters in the `MPRemoteCall` (page 106) function.

Note
Since your application-defined function must be written in PowerPC code, you do not need to build a universal procedure pointer to pass to the `MPRemoteCall` function. ◆

When you call the function `MPRemoteCall` from a task, that task is blocked, and the application-defined function you designated then executes as a cooperatively scheduled task, which can make system software calls with no danger.

Note that when you call `MPRemoteCall`, you can designate which context (or process) you want your application-defined function to execute in. If you specify that the function should execute in the same context that owns the task, the function has access to data available to the main application (just as if the application had called the function). However, the function cannot execute until the owning context becomes active (and then not until the application calls `WaitNextEvent`). Alternatively, you can designate that the function execute in any available context. Doing so minimizes possible lag time, but the function cannot access any resources specific to the task's context.

IMPORTANT
In the future, individual application processes may not always share the same address space, so in general you should never attempt to access code or data in another process. ▲

After your application-defined function returns, the task is unblocked and execution proceeds normally.

Handling Exceptions and Debugging

Multiprocessing Services provides a number of functions you can use to handle exceptions and to aid in debugging.

By default, if you do not register an exception handler, and no debuggers are registered, a task terminates when it takes an exception. If debuggers or exception handlers exist, then the task is suspended when an exception occurs and a message is sent to the appropriate debugger or handler.

If desired, you can install an exception handler for a task by calling the function `MPSetExceptionHandler` (page 98). When an exception occurs, a message is sent to a designated queue, which your exception handler can wait upon.

In addition, you can register one or more debuggers with Multiprocessing Services by calling the function `MPRegisterDebugger` (page 103). When calling `MPRegisterDebugger`, you must specify the queue to which you want the exception message to be sent as well as a debugger level. The debugger level is simply an integer that indicates where to place this debugger in the hierarchy of registered debuggers. In addition, When an exception occurs, the order of notification for handlers is as follows:

- The debugger with the highest debugger level (for example, a debugger registered at level 3 will have precedence over one registered as level 2).
- The debugger with the next highest level (and so on, for all the registered debuggers).
- The task's exception handler.
- The task's termination function.

At each level, the handler can choose to do either of the following:

- Set or retrieve the task's register or stack information using `MPSetTaskState` (page 102) or `MPExtractTaskState` (page 101).
- Call `MPDisposeTaskException` (page 100), which allows you to do any of the following:
  - Resume the task.
  - Resume the task and enable single-stepping or branch-stepping.
  - Propagate the exception to the next lower level. For example, instead of handling the exception itself, a debugger can pass the exception message to the next debugger (or exception handler) in the hierarchy.

If you want to throw an exception to a task, you can use the `MPThrowException` (page 99) function.

CHAPTER 4

Multiprocessing Services Reference

Contents

This chapter describes the Multiprocessing Services application programming interface (API) introduced with Mac OS 8.6. This chapter contains the following sections:

Note
This document describes version 2.0 of Multiprocessing Services. For a list of functions changed or added between versions 1.4 and 2.0, see Appendix B. ◆

Functions

This section describes Multiprocessing Services functions in the following categories:

- "Debugger Support Functions" (page 103)
- "Remote Calling Function" (page 105)

Determining Processor Availability

Multiprocessing Services provides the following functions for determining the number of processors available on the host computer:

- `MPProcessors` (page 56) returns the number of processors on the host computer.
- `MPProcessorsScheduled` (page 57) returns the number of active processors available on the host computer.

MPProcessors

Returns the number of processors on the host computer.

```
ItemCount MPProcessors (void);
```

function result A value of type `ItemCount` that indicates the number of physical processors on the host computer.

VERSION NOTES

Introduced with Multiprocessing Services 1.0.

SEE ALSO

The function `MPProcessorsScheduled` (page 57).

MPProcessorsScheduled

Returns the number of active processors available on the host computer.

```
ItemCount MPProcessorsScheduled (void);
```

function result A value of type `ItemCount` that indicates the number of active processors available on the host computer.

DISCUSSION

The number of active processors is defined as the number of processors scheduled to run tasks. This number varies while the system is running. Advanced power management facilities may stop or start scheduling processors in the system to control power consumption or to maintain a proper operating temperature.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

SEE ALSO

The function `MPProcessors` (page 56).

Creating and Scheduling Tasks

Multiprocessing Services provides the following functions for creating and manipulating preemptive tasks:

- `MPCreateTask` (page 58) creates a preemptive task.
- `MPTerminateTask` (page 60) terminates an existing task.
- `MPExit` (page 61) allows a task to terminate itself.
- `MPCurrentTaskID` (page 62) obtains the task ID of the current preemptive task.
- `MPYield` (page 62) allows a task to yield the processor to another task.
- `MPSetTaskWeight` (page 63) assigns a relative weight to a task, indicating how much processor time it should receive compared to other available tasks.

- `MPTaskIsPreemptive` (page 64) determines whether a task is preemptively scheduled.

MPCreateTask

Creates a preemptive task.

```
OSStatus MPCreateTask (
                    TaskProc entryPoint,
                    void *parameter,
                    ByteCount stackSize,
                    MPQueueID notifyQueue,
                    void *terminationParameter1,
                    void *terminationParameter2,
                    MPTaskOptions options,
                    MPTaskID *task);
```

`entryPoint` A pointer of type `TaskProc` (page 109) that references the task function. The task function should take a single 32-bit parameter and return a value of type `OSStatus`.

`parameter` The parameter to pass to the task function.

`stackSize` A value of type `ByteCount` that specifies the size of the stack assigned to the task. Note that you should be careful not to exceed the bounds of the stack, since stack overflows may not be detected. Specifying zero for the size will result in a default stack size of 4KB.

`notifyQueue` A value of type `MPQueueID` (page 110) that specifies the ID of the message queue to which the system will send a message when the task terminates. You specify the first 64-bits of the message in the parameters `terminationParameter1` and `terminationParameter2` respectively. The last 32-bits contain the result code of the task function.

`terminationParameter1`
A 32-bit value that is sent to the message queue specified by the parameter `notifyQueue` when the task terminates.

`terminationParameter2`
: A 32-bit value that is sent to the message queue specified by the parameter `notifyQueue` when the task terminates.

`options`
: A value of type `MPTaskOptions` (page 110) that specifies optional attributes of the preemptive task. No options are currently defined; this value must be zero.

`task`
: A pointer to a variable of type `MPTaskID` (page 109). On return, the variable contains the ID of the newly created task.

function result
: A result code. See "Result Codes" (page 122) for a list of possible values. If `MPCreateTask` could not create the task because some critical resource was not available, the function returns `kMPInsufficientResourcesErr`. Usually this is due to lack of memory to allocate the internal data structures associated with the task or the stack. The function also returns `kMPInsufficientResourcesErr` if any reserved option bits are set.

DISCUSSION

Tasks are created in the unblocked state, ready for execution. A task can terminate in the following ways:

- By returning from its entry point
- By calling `MPExit` (page 61)
- When specified as the target of an `MPTerminateTask` (page 60) call
- If a hardware-detected exception or programming exception occurs and no exception handler is installed
- If the application calls `ExitToShell`

Task resources (its stack, active timers, internal structures related to the task, and so on) are reclaimed by the system when the task terminates. The task's address space is inherited from the process address space. All existing tasks are terminated when the owning process terminates.

VERSION NOTES

Introduced with Multiprocessing Services 1.0.

SEE ALSO

The function `MPTerminateTask` (page 60).

The function `MPSetTaskWeight` (page 63).

MPTerminateTask

Terminates an existing task.

```
OSStatus MPTerminateTask (
                    MPTaskID task,
                    OSStatus terminationStatus);
```

`task` A value of type `MPTaskID` (page 109) that specifies the ID of the task you wish to terminate.

`terminationStatus`
A 32-bit value of type `OSStatus` indicating termination status. This value is sent to the termination status message queue you specified in `MPCreateTask` (page 58) in place of the task function's result code.

function result A result code. See "Result Codes" (page 122) for a list of possible values. If the task to be terminated is already in the process of termination, `MPTerminateTask` returns `kMPInsufficientResourcesErr`. You do not need to take any additional action if this occurs.

DISCUSSION

You should not assume that the task has completed termination when this call returns; the proper way to synchronize with task termination is to wait on the termination queue (specified in `MPCreateTask` (page 58)) until a message appears. Because task termination is a multistage activity, it is possible for a preemptive task to attempt to terminate a task that is already undergoing termination. In such cases, `MPTerminateTask` returns `kMPInsufficientResourcesErr`.

Note that Multiprocessing Services resources (event groups, queues, semaphores, and critical regions) owned by a preemptive task are not released

when that task terminates. If a task has a critical region locked when it terminates, the critical region remains in the locked state. Multiprocessing Services resources no longer needed should be explicitly deleted by the task that handles the termination message. All Multiprocessing Services resources created by tasks are released when their owning process (that is, the host application) terminates.

VERSION NOTES

Introduced with Multiprocessing Services 1.0.

MPExit

Allows a task to terminate itself

```
void MPExit (OSStatus status);
```

`status` An application-defined value of type `OSStatus` that indicates termination status. This value is sent to the termination message queue in place of the task's result code.

DISCUSSION

When called from within a preemptive task, the task terminates, and the value indicated by the parameter `status` is sent to the termination message queue you specified in `MPCreateTask` (page 58). Note that you cannot call `MPExit` from outside a preemptive task.

VERSION NOTES

Introduced with Multiprocessing Services 1.0.

MPCurrentTaskID

Obtains the task ID of the currently-executing preemptive task

```
MPTaskID MPCurrentTaskID (void);
```

function result A value of type `MPTaskID` (page 109) that specifies the task ID of the current preemptive task.

DISCUSSION

Returns the ID of the current preemptive task. If called from a cooperative task, this function returns an ID which is different than the ID of any preemptive task. Nonpreemptive processes may or may not have different task IDs for each application; future implementations of this API may behave differently in this regard.

Note that you can call this function from an interrupt handler.

VERSION NOTES

Introduced with Multiprocessing Services 1.0.

MPYield

Allows a task to yield the processor to another task.

```
void MPYield (void);
```

DISCUSSION

This function indicates to the scheduler that another task can run. Other than possibly yielding the processor to another task or application, the call has no effect. Note that since tasks are preemptively scheduled, an implicit yield may occur at any point, whether or not this function is called.

In most cases you should not need to call this function. The most common use of `MPYield` is to release the processor when a task is in a loop in which further

progress is dependent on other tasks, and the task cannot be blocked by waiting on a Multiprocessing Services resource. You should avoid such busy waiting whenever possible.

Note that you can call this function from an interrupt handler.

VERSION NOTES

Introduced with Multiprocessing Services 1.0.

MPSetTaskWeight

Assigns a relative weight to a task, indicating how much processor time it should receive compared to other available tasks.

```
OSStatus MPSetTaskWeight (
                    MPTaskID task,
                    MPTaskWeight weight);
```

`task`	A value of type `MPTaskID` (page 109) that specifies the ID of the task to which you want to assign a weighting.
`weight`	A value of type `MPTaskWeight` (page 110) indicating the relative weight to assign. This value can range from 1 to 10,000, with the default value being 100.
function result	A result code. See "Result Codes" (page 122) for a list of possible values.

DISCUSSION

The approximate processor share is defined as

weight of the task/ total weight of available tasks

For a set of ready tasks, the amount of CPU time dedicated to the tasks will be determined by the dynamically computed share. Note that the processor share devoted to tasks may deviate from the suggested weighting if critical tasks require attention. For example, a real-time task (such as a QuickTime movie) may require more than its relative weight of processor time, and the scheduler will adjust proportions accordingly.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

MPTaskIsPreemptive

Determines whether a task is preemptively scheduled.

```
Boolean MPTaskIsPreemptive (MPTaskID taskID);
```

`taskID`	A value of type `MPTaskID` (page 109) that specifies the task you want to check. Pass `kInvalidID` if you want to specify the current task.
function result	A value of type `Boolean`. If true, the task is preemptively scheduled. If false, the task is cooperatively scheduled.

DISCUSSION

If you have code that may be called from either cooperative or preemptive tasks, that code can call `MPTaskIsPreemptive` if its actions should differ depending on its execution environment.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

Creating and Handling Message Queues

Multiprocessing Services provides the following functions for creating and handling message queues:

- `MPCreateQueue` (page 65) creates a message queue.
- `MPDeleteQueue` (page 66) deletes a message queue.
- `MPNotifyQueue` (page 66) sends a message to the specified message queue.
- `MPWaitOnQueue` (page 68) obtains a message from a specified message queue.

- `MPSetQueueReserve` (page 69) reserves space for messages on a specified message queue.

MPCreateQueue

Creates a message queue.

```
OSStatus MPCreateQueue (MPQueueID *queue);
```

`queue`	A pointer to a variable of type `MPQueueID` (page 110). On return, the variable contains the ID of the newly created message queue.
function result	A result code. See "Result Codes" (page 122) for a list of possible values. If a queue could not be created, `MPCreateQueue` returns `kMPInsufficientResourcesErr`.

DISCUSSION

This call creates a message queue, which can be used to notify (that is, send) and wait for (that is, receive) messages consisting of three 32-bit words in a preemptively safe manner.

Message queues are created from dynamically allocated internal resources. Other tasks may be competing for these resources so it is possible this function may not be able to create a queue.

VERSION NOTES

Introduced with Multiprocessing Services 1.0.

SEE ALSO

The function `MPDeleteQueue` (page 66).

The function `MPSetQueueReserve` (page 69).

MPDeleteQueue

Deletes a message queue.

```
OSStatus MPDeleteQueue (MPQueueID queue);
```

`queue`	A value of type `MPQueueID` (page 110) that specifies the ID of the message queue you want to delete.
function result	A result code. See "Result Codes" (page 122) for a list of possible values.

DISCUSSION

After calling `MPDeleteQueue`, the specified queue ID becomes invalid, and all internal resources associated with the queue (including queued messages) are reclaimed. Any tasks waiting on the queue are unblocked and their respective `MPWaitOnQueue` (page 68) calls will return with the result code `kMPDeletedErr`.

VERSION NOTES

Introduced with Multiprocessing Services 1.0.

SEE ALSO

The function `MPCreateQueue` (page 65).

MPNotifyQueue

Sends a message to the specified message queue.

```
OSStatus MPNotifyQueue (
                    MPQueueID queue,
                    void *param1,
                    void *param2,
                    void *param3);
```

`queue`	A value of type `MPQueueID` (page 110) that specifies the queue ID of the message queue you want to notify.
`param1`	The first 32-bits of the message to send.
`param2`	The second 32-bits of the message to send.
`param3`	The third 32-bits of the message to send.
function result	A result code. See "Result Codes" (page 122) for a list of possible values.

DISCUSSION

This function sends a message to the specified queue, which consist of the three parameters, `param1`, `param2`, and `param3`. The system does not interpret the three 32-bit words which comprise the text of the message. If tasks are waiting on the specified queue, the first waiting task is unblocked and the task's `MPWaitOnQueue` (page 68) function completes.

Depending on the queue mode, the system either allocates messages dynamically or assigns them to memory reserved for the queue. In either case, if no more memory is available for messages `MPNotifyQueue` returns `kMPInsufficientResourcesErr`.

You can call this function from an interrupt handler if messages are reserved on the queue. For more information about queueing modes and reserving messages, see `MPSetQueueReserve` (page 69).

VERSION NOTES

Introduced with Multiprocessing Services 1.0.

SEE ALSO

The function `MPWaitOnQueue` (page 68).

MPWaitOnQueue

Obtains a message from a specified message queue.

```
OSStatus MPWaitOnQueue (
                    MPQueueID queue,
                    void **param1,
                    void **param2,
                    void **param3,
                    Duration timeout);
```

`queue`	A value of type `MPQueueID` (page 110) that specifies the ID of the message queue from which to receive the notification.
`param1`	A pointer to a 32-bit variable. On return, this variable contains the first 32-bits of the notification message. Pass `NULL` if you do not need this portion of the message.
`param2`	A pointer to a 32-bit variable. On return, the variable contains the second 32-bits of the notification message. Pass `NULL` if you do not need this portion of the message.
`param3`	A pointer to a 32-bit variable. On return, the variable contains the third 32-bits of the notification message. Pass `NULL` if you do not need this portion of the message.
`timeout`	A value of type `Duration` specifying how long to wait for a notification before timing out. See "Timer Duration Constants" (page 116) for a list of constants you can use to specify the wait interval.
function result	A result code. See "Result Codes" (page 122) for a list of possible values.

DISCUSSION

This function receives a message from the specified message queue. If no messages are currently available, the timeout specifies how long the function should wait for one. Tasks waiting on the queue are handled in a first in, first out manner; that is, the first task to wait on the queue receives the message from the `MPNotifyQueue` (page 66) call.

After calling this function, when a message appears, it is removed from the queue and the three fields, `param1`, `param2`, and `param3` are set to the values specified by the message text. Note these parameters are pointers to variables to be set with the message text.

If you call this function from a cooperative task, you should specify only `kDurationImmediate` for the timeout length; other waits will cause the task to block.

VERSION NOTES

Introduced with Multiprocessing Services 1.0.

SEE ALSO

The function `MPNotifyQueue` (page 66).

MPSetQueueReserve

Reserves space for messages on a specified message queue.

```
OSStatus MPSetQueueReserve (
                    MPQueueID queue,
                    ItemCount count);
```

`queue`	A value of type `MPQueueID` (page 110) that specifies the ID of the queue whose messages you want to reserve.
`count`	A value of type `ItemCount` that specifies the number of messages to reserve.
function result	A result code. See "Result Codes" (page 122) for a list of possible values.

DISCUSSION

`MPNotifyQueue` (page 66) allocates spaces for messages dynamically; that is, memory to hold the message is allocated for the queue at the time of the call. In most cases this method is both speed and storage efficient. However, it is

possible that, due to lack of memory resources, space for the message may not be available at the time of the call; in such cases, `MPNotifyQueue` (page 66) will return `kInsufficientResourcesErr`.

If you must have guaranteed message delivery, or if you need to call `MPNotifyQueue` (page 66) from an interrupt handler, you should reserve space on the specified queue by calling `MPSetQueueReserve`. Because such allocated space is reserved for duration of the queue's existence, you should avoid straining internal system resources by reserving messages only when absolutely necessary. Note that if you have reserved messages on a queue, additional space *cannot* be added dynamically if the number of messages exceeds the number reserved for that queue.

The number of reserved messages is set to `count`, lowering or increasing the current number of reserved messages as required. If `count` is set to zero, no messages are reserved for the queue, and space for messages is allocated dynamically.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

Creating and Handling Semaphores

Multiprocessing Services provides the following functions for creating and handling semaphores:

- `MPCreateSemaphore` (page 71) creates a semaphore.
- `MPDeleteSemaphore` (page 72) removes a semaphore.
- `MPSignalSemaphore` (page 72) signals a semaphore.
- `MPWaitOnSemaphore` (page 73) waits on a semaphore.

MPCreateSemaphore

Creates a semaphore.

```
OSStatus MPCreateSemaphore (
                    MPSemaphoreCount maximumValue,
                    MPSemaphoreCount initialValue,
                    MPSemaphoreID *semaphore);
```

`maximumValue` A value of type `MPSemaphoreCount` (page 113) that specifies the maximum allowed value of the semaphore.

`initialValue` A value of type `MPSemaphoreCount` (page 113) that specifies the initial value of the semaphore.

`semaphore` A pointer to a variable of type `MPSemaphoreID` (page 111). On return, the variable contains the ID of the newly–created semaphore.

function result A result code. See "Result Codes" (page 122) for a list of possible values.

DISCUSSION

If you want to create a binary semaphore, you can call the macro `MPCreateBinarySemaphore (MPSemaphoreID *semaphore)` instead, which simply calls `MPCreateSemaphore` with both `maximumValue` and `initialValue` set to 1.

VERSION NOTES

Introduced with Multiprocessing Services 1.0.

SEE ALSO

The function `MPDeleteSemaphore` (page 72).

MPDeleteSemaphore

Removes a semaphore.

```
OSStatus MPDeleteSemaphore (MPSemaphoreID semaphore);
```

`semaphore` A value of type `MPSemaphoreID` (page 111) that specifies the ID of the semaphore you want to remove.

function result A result code. See "Result Codes" (page 122) for a list of possible values.

DISCUSSION

Calling this function unblocks all tasks waiting on the semaphore and the tasks' respective `MPWaitOnSemaphore` (page 73) calls will return with the result code `kMPDeletedErr`.

VERSION NOTES

Introduced with Multiprocessing Services 1.0.

SEE ALSO

The function `MPCreateSemaphore` (page 71).

MPSignalSemaphore

Signals a semaphore.

```
OSStatus MPSignalSemaphore (MPSemaphoreID semaphore);
```

`semaphore` A value of type `MPSemaphoreID` (page 111) that specifies the ID of the semaphore you want to signal.

function result A result code. See "Result Codes" (page 122) for a list of possible values. If the value of the semaphore was already at the maximum, `MPSignalSemaphore` returns `kInsufficientResourcesErr`.

DISCUSSION

If tasks are waiting on the semaphore, the oldest (first queued) task is unblocked so that the corresponding `MPWaitOnSemaphore` (page 73) call for that task completes. Otherwise, if the value of the semaphore is not already equal to its maximum value, it is incremented by one.

Note that you can call this function from an interrupt handler.

VERSION NOTES

Introduced with Multiprocessing Services 1.0.

SEE ALSO

The function `MPWaitOnSemaphore` (page 73).

MPWaitOnSemaphore

Waits on a semaphore

```
OSStatus MPWaitOnSemaphore (
                    MPSemaphoreID semaphore,
                    Duration timeout);
```

`semaphore` A value of type `MPSemaphoreID` (page 111) that specifies the ID of the semaphore you want to wait on.

`timeout` A value of type `Duration` that specifies the maximum time the function should wait before timing out. See "Timer Duration Constants" (page 116) for a list of constants you can use to specify the wait interval.

function result A result code. See "Result Codes" (page 122) for a list of possible values.

DISCUSSION

If the value of the semaphore is greater than zero, the value is decremented and the function returns with `noErr`. Otherwise, the task is blocked awaiting a signal until the specified timeout is exceeded.

If you call this function from a cooperative task, you should specify only `kDurationImmediate` for the timeout length; other waits will cause the task to block.

VERSION NOTES

Introduced with Multiprocessing Services 1.0.

SEE ALSO

The function `MPSignalSemaphore` (page 72).

Handling Event Groups

Multiprocessing Services provides the following functions for creating and handling event groups:

- `MPCreateEvent` (page 74) creates an event group.
- `MPDeleteEvent` (page 75) removes an event group.
- `MPSetEvent` (page 76) merges event flags into a specified event group.
- `MPWaitForEvent` (page 77) retrieves event flags from a specified event group.

MPCreateEvent

Creates an event group.

```
OSStatus MPCreateEvent (MPEventID *event);
```

`event` A pointer to a variable of type `MPEventID` (page 111). On return, the variable contains the ID of the newly created event group.

function result A result code. See "Result Codes" (page 122) for a list of possible values.

DISCUSSION

Event groups are created from dynamically allocated internal resources. Other tasks may be competing for these resources so it is possible that this function will not be able to create an event group.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

SEE ALSO

The function `MPDeleteEvent` (page 75).

MPDeleteEvent

Removes an event group.

```
OSStatus MPDeleteEvent (MPEventID event);
```

`event` A value of type `MPEventID` (page 111) that specifies the ID of the event group you want to remove.

function result A result code. See "Result Codes" (page 122) for a list of possible values.

DISCUSSION

After deletion, the event ID becomes invalid, and all internal resources associated with the event group are reclaimed. Calling this function unblocks all tasks waiting on the event group and their respective `MPWaitForEvent` (page 77) calls will return with the result code `kMPDeletedErr`.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

SEE ALSO

The function `MPCreateEvent` (page 74).

MPSetEvent

Merges event flags into a specified event group.

```
OSStatus MPSetEvent (
                    MPEventID event,
                    MPEventFlags flags);
```

`event` A value of type `MPEventID` (page 111) that specifies the ID of the event group you want to set.

`flags` A 32-bit value of type `MPEventFlags` (page 114) that contains the flags you want to merge into the event group.

function result A result code. See "Result Codes" (page 122) for a list of possible values.

DISCUSSION

The flags are logically ORed with the current flags in the event group. This procedure is an atomic operation to ensure that multiple updates do not get lost. If tasks are waiting on this event group, the first waiting task is unblocked.

Note that you can call this function from an interrupt handler.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

SEE ALSO

The function `MPWaitForEvent` (page 77).

MPWaitForEvent

Retrieves event flags from a specified event group.

```
OSStatus MPWaitForEvent (
                    MPEventID event,
                    MPEventFlags *flags,
                    Duration timeout);
```

`event`	A value of type `MPEventID` (page 111) that specifies the event group whose flags you want to retrieve.
`flags`	A pointer to a 32-bit variable of type `MPEventFlags` (page 114). On return, the variable contains the flags of the specified event group. Pass `NULL` if you do not need any flag information.
`timeout`	A value of type `Duration` that specifies the maximum time to wait for events before timing out. See "Timer Duration Constants" (page 116) for a list of constants you can use to specify the wait interval.
function result	A result code. See "Result Codes" (page 122) for a list of possible values.

DISCUSSION

This function obtains event flags from the specified event group. The timeout specifies how long to wait for events if none are present when the call is made. If any flags are set when this function is called, all the flags in the event group are moved to the `flag` field and the event group is cleared. This obtaining and clearing action is an atomic operation to ensure that no updates are lost. If multiple tasks are waiting on an event group, only one can obtain any particular set of flags.

If you call this function from a cooperative task, you should specify only `kDurationImmediate` for the timeout length; other waits will cause the task to block.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

SEE ALSO

The function `MPSetEvent` (page 76).

Handling Critical Regions

Multiprocessing Services provides the following functions for creating and handling critical regions:

- `MPCreateCriticalRegion` (page 78) creates a critical region object.
- `MPDeleteCriticalRegion` (page 79) removes the specified critical region object.
- `MPEnterCriticalRegion` (page 79) attempts to enter a critical region.
- `MPExitCriticalRegion` (page 81) exits a critical region.

MPCreateCriticalRegion

Creates a critical region object.

```
OSStatus MPCreateCriticalRegion (MPCriticalRegionID *criticalRegion);
```

`criticalRegion`
A pointer to a variable of type `MPCriticalRegionID` (page 112). On return, the variable contains the ID of the newly created critical region object.

function result A result code. See "Result Codes" (page 122) for a list of possible values.

VERSION NOTES

Introduced with Multiprocessing Services 1.0.

SEE ALSO

The function `MPDeleteCriticalRegion` (page 79).

MPDeleteCriticalRegion

Removes the specified critical region object.

```
OSStatus MPDeleteCriticalRegion (MPCriticalRegionID criticalRegion);
```

`criticalRegion`
A value of type `MPCriticalRegionID` (page 112) that specifies the critical region object you want to remove.

function result A result code. See "Result Codes" (page 122) for a list of possible values.

DISCUSSION

Calling this function unblocks all tasks waiting to enter the critical region and their respective `MPEnterCriticalRegion` (page 79) calls will return with the result code `kMPDeletedErr`.

VERSION NOTES

Introduced with Multiprocessing Services 1.0.

SEE ALSO

The function `MPCreateCriticalRegion` (page 78).

MPEnterCriticalRegion

Attempts to enter a critical region.

```
OSStatus MPEnterCriticalRegion (
                    MPCriticalRegionID criticalRegion,
                    Duration timeout);
```

`criticalRegion`
A value of type `MPCriticalRegionID` (page 112) that specifies the ID of the critical region you want to enter.

`timeout`	A value of type `Duration` that specifies the maximum time to wait for entry before timing out. See "Timer Duration Constants" (page 116) for a list of constants you can use to specify the wait interval.
function result	A result code. See "Result Codes" (page 122) for a list of possible values.

DISCUSSION

If another task currently occupies the critical region, the current task is blocked until the critical region is released or until the designated timeout expires. Otherwise the task enters the critical region and `MPEnterCriticalRegion` increments the region's use count.

Once a task enters a critical region it can make further calls to `MPEnterCriticalRegion` without blocking (its use count increments for each call). However, each call to `MPEnterCriticalRegion` must be balanced by a call to `MPExitCriticalRegion` (page 81); otherwise the region is not released for use by other tasks.

Note that you can enter a critical region from a cooperative task. Each cooperative task is treated as unique and different from any preemptive task. If you call this function from a cooperative task, you should specify only `kDurationImmediate` for the timeout length; other waits will cause the task to block.

VERSION NOTES

Introduced with Multiprocessing Services 1.0.

SEE ALSO

The function `MPExitCriticalRegion` (page 81).

MPExitCriticalRegion

Exits a critical region.

```
OSStatus MPExitCriticalRegion (MPCriticalRegionID criticalRegion);
```

`criticalRegion`
A value of type `MPCriticalRegionID` that specifies the ID of the critical region you want to exit.

function result A result code. See "Result Codes" (page 122) for a list of possible values. If the task does not own the critical region specified by `criticalRegion`, `MPExitCriticalRegion` returns `kMPInsufficientResourcesErr`.

DISCUSSION

This function decrements the use count of the critical region object. When the use count reaches zero, ownership of the critical region object is released (which allows another task to use the critical region).

VERSION NOTES

Introduced with Multiprocessing Services 1.0.

SEE ALSO

The function `MPEnterCriticalRegion` (page 79).

Timer Services Functions

Multiprocessing Services provides the following timer services functions:

- `MPDelayUntil` (page 82) blocks the calling task until a specified time.
- `MPCreateTimer` (page 82) creates a timer.
- `MPDeleteTimer` (page 83) removes a timer.
- `MPSetTimerNotify` (page 84) sets the notification information associated with a timer.

- `MPArmTimer` (page 86) arms the timer to expire at a given time.
- `MPCancelTimer` (page 87) cancels an armed timer.

MPDelayUntil

Blocks the calling task until a specified time.

```
OSStatus MPDelayUntil (AbsoluteTime *expirationTime);
```

`expirationTime`
A value of type `AbsoluteTime` specifying when to unblock the task.

function result A result code. See "Result Codes" (page 122) for a list of possible values.

DISCUSSION

You cannot call this function from a cooperative task.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

MPCreateTimer

Creates a timer.

```
OSStatus MPCreateTimer (MPTimerID *timerID);
```

`timerID` A pointer to a variable of type `MPTimerID` (page 112). On return, the variable contains the ID of the newly created timer.

function result A result code. See "Result Codes" (page 122) for a list of possible values.

DISCUSSION

You can use a timer to notify an event, queue, or semaphore after a specified amount of time has elapsed.

Timer objects are created from dynamically-allocated internal resources. Other tasks may be competing for these resources so it is possible this function may not be able to create one.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

SEE ALSO

The function `MPDeleteTimer` (page 83).

The function `MPArmTimer` (page 86).

The function `MPSetTimerNotify` (page 84).

MPDeleteTimer

Removes a timer.

```
OSStatus MPDeleteTimer (MPTimerID timerID);
```

`timerID`	A value of type `MPTimerID` (page 112) that specifies the ID of the timer you want to remove.
function result	A result code. See "Result Codes" (page 122) for a list of possible values.

DISCUSSION

After deletion, the timer ID becomes invalid, and all internal resources associated with the timer are reclaimed.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

SEE ALSO

The function `MPCreateTimer` (page 82).

MPSetTimerNotify

Sets the notification information associated with a timer.

```
OSStatus MPSetTimerNotify (
                    MPTimerID timerID,
                    MPOpaqueID notificationID,
                    void *notifyParam1,
                    void *notifyParam2,
                    void *notifyParam3);
```

`timerID` A value of type `MPTimerID` (page 112) that specifies the ID of the timer whose notification information you want to set.

`notificationID` A value specifying the ID of the notification mechanism to associate with the timer. This value should be the ID of an event group, a message queue, or a semaphore.

`notifyParam1` If `notificationID` specifies an event group, this parameter should contain the flags to set in the event group when the timer expires. If `notificationID` specifies a message queue, this parameter should contain the first 32-bits of the message to be sent to the message queue when the timer expires.

`notifyParam2` If `notificationID` specifies a message queue, this parameter should contain the second 32-bits of the message to be sent to the message queue when the timer expires. Pass `NULL` if you don't need this parameter.

`notifyParam3`	If `notificationID` specifies a message queue, this parameter should contain the third 32-bits of the message sent to the message queue when the timer expires. Pass `NULL` if you don't need this parameter.
function result	A result code. See "Result Codes" (page 122) for a list of possible values.

DISCUSSION

When the timer expires, Multiprocessing Services checks the notification ID, and if it is valid, notifies the related notification mechanisms (that is, event groups, queues, or semaphores) you had specified in your `MPSetTimerNotify` (page 84) calls.

You can specify multiple notification mechanisms by calling this function several times. For example, you can call `MPSetTimerNotify` to specify a message queue and then call it again to specify a semaphore. When the timer expires, a message is sent to the message queue and the appropriate semaphore is signaled. You cannot, however, specify more than one notification per notification mechanism (for example, if you call `MPSetTimerNotify` twice, specifying different messages or message queues in each call, the second call will overwrite the first). Note that if a call to `MPSetTimerNotify` returns an error, any previous calls specifying the same timer are still valid; previously set notifications will still be notified when the timer expires.

You can set the notification information at any time. If the timer is armed, it will modify the notification parameters dynamically. If the timer is disarmed, it will modify the notification parameters to be used for the next `MPArmTimer` (page 86) call.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

MPArmTimer

Arms the timer to expire at a given time.

```
OSStatus MPArmTimer (
                    MPTimerID timerID,
                    AbsoluteTime *expirationTime,
                    OptionBits options);
```

`timerID` A value of type `MPTimerID` (page 112) that specifies the ID of the timer you want to arm.

`expirationTime` A pointer to a value of type `AbsoluteTime` that specifies when you want the timer to expire. Note that if you arm the timer with a time that has already passed, the timer expires immediately.

`options` A value of type `OptionBits` specifying any optional action. See "Timer Option Masks" (page 116) for a list of possible values.

function result A result code. See "Result Codes" (page 122) for a list of possible values. If the timer has already expired, the reset does not take place and the function returns `kMPInsufficientResourcesErr`.

DISCUSSION

The expiration time is an absolute time, which you can generate by calling the `UpTime` (page 88) function. When the timer expires, a notification is sent to the notification mechanism specified in the last `MPSetTimerNotify` (page 84) call. If the specified notification ID has become invalid, no action is taken when the timer expires. The timer itself is deleted when it expires unless you specified the `kMPPreserveTimerID` option in the options parameter.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

SEE ALSO

The function `MPCancelTimer` (page 87).

The function `MPSetTimerNotify` (page 84).

MPCancelTimer

Cancels an armed timer.

```
OSStatus MPCancelTimer (
                    MPTimerID timerID,
                    AbsoluteTime *timeRemaining);
```

`timerID` A value of type `MPTimerID` that specifies the ID of the armed timer you want to cancel.

`timeRemaining`
A pointer to a variable of type `AbsoluteTime`. On return, the variable contains the time remaining before the timer would have expired.

function result A result code. See "Result Codes" (page 122) for a list of possible values. If the timer has already expired, this function returns `kMPInsufficientResourcesErr`.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

SEE ALSO

The function `MPArmTimer` (page 86).

Time Utility Functions

Multiprocessing Services provides the following time utility functions:

- `UpTime` (page 88) obtains the time elapsed since system startup.
- `DurationToAbsolute` (page 88) converts time from units of type `Duration` to units of type `AbsoluteTime`.

- `AbsoluteToDuration` (page 89) converts time from units of type `AbsoluteTime` to units of type `Duration`.

UpTime

Obtains the time elapsed since machine startup.

```
AbsoluteTime UpTime (void);
```

function result A value of type `AbsoluteTime` that indicates the time elapsed since the host computer was started up. Absolute time is a 64-bit monotonically increasing value. You should not make any assumptions about what units absolute time is based upon.

DISCUSSION

You can call this function from the task level, software interrupt level, or hardware interrupt level.

This function is identical to the `UpTime` function found in the Driver Services Library.

VERSION NOTES

Available in the Multiprocessing Services library with version 2.0.

DurationToAbsolute

Converts time from units of type `Duration` to units of type `AbsoluteTime`.

```
AbsoluteTime DurationToAbsolute (Duration duration);
```

`duration` A value of type `Duration` that you want to convert to type `AbsoluteTime`.

function result A value of type `AbsoluteTime` which is the converted time.

DISCUSSION

This function is identical to the `DurationToAbsolute` function found in the Driver Services Library.

VERSION NOTES

Available in the Multiprocessing Services library with version 2.0.

SEE ALSO

The function `AbsoluteToDuration` (page 89).

AbsoluteToDuration

Converts time from units of type `AbsoluteTime` to units of type `Duration`.

```
Duration AbsoluteToDuration (AbsoluteTime time);
```

`time`	A value of type `AbsoluteTime` that you want to convert to type `Duration`.
function result	A value of type `Duration` which is the converted time.

DISCUSSION

This function is identical to the `AbsoluteToDuration` function found in the Driver Services Library.

VERSION NOTES

Available in the Multiprocessing Services library with version 2.0.

SEE ALSO

The function `DurationToAbsolute` (page 88).

Accessing Per-Task Storage Variables

Multiprocessing Services provides the following functions for creating and manipulating per-task storage variables:

- `MPAllocateTaskStorageIndex` (page 90) returns an index number to access per-task storage.
- `MPDeallocateTaskStorageIndex` (page 91) frees an index number used to access per-task storage.
- `MPSetTaskStorageValue` (page 91) sets the storage value for a given index number.
- `MPGetTaskStorageValue` (page 92) gets the storage value stored at a specified index number.

MPAllocateTaskStorageIndex

Returns an index number to access per-task storage.

```
OSStatus MPAllocateTaskStorageIndex (TaskStorageIndex *index);
```

`index` A pointer to a variable of type `TaskStorageIndex` (page 113). On return, the variable contains an index number you can use to store task data.

function result A result code. See "Result Codes" (page 122) for a list of possible values.

DISCUSSION

A call to the function `MPAllocateTaskStorageIndex` returns an index number that is common across all tasks in the current process. You can use this index number in calls to `MPSetTaskStorageValue` (page 91) and `MPGetTaskStorageValue` (page 92) to set a different value for each task using the same index.

You can think of the task storage area as a two dimensional array cross-referenced by the task storage index number and the task ID. Note that since the amount of per-task storage is determined when the task is created, the number of possible index values associated with a task is limited.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

SEE ALSO

The function `MPDeallocateTaskStorageIndex` (page 91).

MPDeallocateTaskStorageIndex

Frees an index number used to access per-task storage

```
OSStatus MPDeallocateTaskStorageIndex (TaskStorageIndex index);
```

`index`	A value of type `TaskStorageIndex` (page 113) that specifies the index number you want to deallocate.
function result	A result code. See "Result Codes" (page 122) for a list of possible values.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

SEE ALSO

The function `MPAllocateTaskStorageIndex` (page 90).

MPSetTaskStorageValue

Sets the storage value for a given index number.

```
OSStatus MPSetTaskStorageValue (
                    TaskStorageIndex index,
                    TaskStorageValue value);
```

`index`	A 32-bit value of type `TaskStorageIndex` (page 113) specifying the index number whose storage value you want to set.
`value`	A value of type `TaskStorageValue` (page 113) that specifies the value you want to set.
function result	A result code. See "Result Codes" (page 122) for a list of possible values.

DISCUSSION

Typically you use `MPSetTaskStorageValue` to store pointers to task-specific structures or data.

Calling this function from within a task effectively assigns a value in a two-dimensional array cross-referenced by task storage index value and the task ID.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

SEE ALSO

The function `MPGetTaskStorageValue` (page 92).

MPGetTaskStorageValue

Gets the storage value stored at a specified index number.

```
TaskStorageValue MPGetTaskStorageValue (TaskStorageIndex index);
```

`index`	A 32-bit value of type `TaskStorageIndex` (page 113) specifying the index number of the storage value you want to obtain.
function result	A value of type `TaskStorageValue` (page 113) that is the value stored at the specified index number.

DISCUSSION

Calling this function from within a task effectively reads a value in a two-dimensional array cross-referenced by task storage index value and the task ID.

Note that since this function does not return any status information, it may not be immediately obvious whether the returned storage value is valid.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

SEE ALSO

The function `MPSetTaskStorageValue` (page 91).

Memory Allocation Functions

Multiprocessing Services provides the following functions for allocating and manipulating blocks of memory from within preemptive tasks:

- `MPAllocateAligned` (page 94) allocates a nonrelocatable memory block.
- `MPFree` (page 95) frees memory allocated by `MPAllocateAligned` (page 94).
- `MPGetAllocatedBlockSize` (page 95) returns the size of a memory block.
- `MPBlockCopy` (page 96) copies a block of memory.
- `MPBlockClear` (page 97) clears a block of memory.
- `MPDataToCode` (page 97) designates the specified block of memory as executable code.

MPAllocateAligned

Allocates a nonrelocatable memory block.

```
LogicalAddress MPAllocateAligned (
                    ByteCount size,
                    UInt8 alignment,
                    OptionBits options);
```

`size` A value of type `ByteCount` specifying the size, in bytes, of the memory block to allocate.

`alignment` An integer specifying the desired alignment of the allocated memory block. See "Memory Allocation Alignment Constants" (page 117) for a list of possible values to pass. Note that there will be a minimum alignment regardless of the requested alignment. If the requested memory block is 4 bytes or smaller, the block will be at least 4-byte aligned. If the requested block is greater than 4 bytes, the block will be at least 8-byte aligned.

`options` A value of type `OptionBits` that specifies any optional information to use with this call. See "Memory Allocation Option Constants" (page 119) for a list of possible values to pass.

function result A pointer to the allocated memory. If the function cannot allocate the requested memory or the requested alignment, the returned address is `NULL`.

DISCUSSION

The memory referenced by the returned address is guaranteed to be accessible by the application's cooperative task and any preemptive tasks that it creates, but not by other applications or their preemptive tasks. Any existing nonglobal heap blocks are freed when the application terminates. As with all shared memory, you must explicitly synchronize access to allocated heap blocks using a notification mechanism.

You can replicate the effect of the older `MPAllocate` function by calling `MPAllocateAligned` with 32-byte alignment and no options.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

SEE ALSO

The function `MPFree` (page 95).

MPFree

Frees memory allocated by `MPAllocateAligned` (page 94).

```
void MPFree (LogicalAddress object);
```

`object` A pointer of type `LogicalAddress` to the memory you want to release.

VERSION NOTES

Introduced with Multiprocessing Services 1.0.

SEE ALSO

The function `MPAllocateAligned` (page 94).

MPGetAllocatedBlockSize

Returns the size of a memory block.

```
ByteCount MPGetAllocatedBlockSize (LogicalAddress object);
```

`object` A value of type `LogicalAddress` that specifies the address of the memory block whose size you want to determine.

function result A value of type `ByteCount` that is the size of the allocated memory block, in bytes.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

MPBlockCopy

Copies a block of memory.

```
void MPBlockCopy (
                    LogicalAddress source,
                    LogicalAddress destination,
                    ByteCount size);
```

`source`	A pointer of type `LogicalAddress` that specifies the starting address of the memory block you want to copy.
`destination`	A pointer of type `LogicalAddress` that specifies the location to which you want to copy the memory block.
`size`	A value of type `ByteCount` that specifies the number of bytes to copy.

DISCUSSION

As with all shared memory, your application must synchronize access to the memory blocks to avoid data corruption. `MPBlockCopy` ensures the copying stays within the bounds of the area specified by `size`, but the calling task can be preempted during the copying process.

VERSION NOTES

Introduced with Multiprocessing Services 1.0.

MPBlockClear

Clears a block of memory.

```
void MPBlockClear (
                    LogicalAddress address,
                    ByteCount size);
```

`address` A pointer of type `LogicalAddress` that specifies the starting address of the memory block you want to clear.

`size` A value of type `ByteCount` that specifies the number of bytes you want to clear.

DISCUSSION

As with all shared memory, your application must synchronize access to the memory blocks to avoid data corruption. `MPBlockClear` ensures the clearing stays within the bounds of the area specified by `size`, but the calling task can be preempted during the copying process.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

MPDataToCode

Designates the specified block of memory as executable code.

```
void MPDataToCode (
                    LogicalAddress address,
                    ByteCount size);
```

`address` A pointer of type `LogicalAddress` that specifies the starting address of the memory block you want to designate as code.

`size` A value of type `ByteCount` that specifies the size of the memory block.

DISCUSSION

Since PowerPC processors need to differentiate between code and data in memory, you should call this function to tag any executable code that your tasks may generate.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

Exception Handling Functions

Multiprocessing Services provides the following functions for handling exceptions and examining the state of a suspended task:

- `MPSetExceptionHandler` (page 98) sets an exception handler for a task.
- `MPThrowException` (page 99) throws an exception to a specified task.
- `MPDisposeTaskException` (page 100) removes a task exception.
- `MPExtractTaskState` (page 101) extracts state information from a suspended task.
- `MPSetTaskState` (page 102) sets state information for a suspended task.

MPSetExceptionHandler

Sets an exception handler for a task.

```
OSStatus MPSetExceptionHandler (
                    MPTaskID task,
                    MPQueueID exceptionQ);
```

`task`	A value of type `MPTaskID` (page 109) that specifies the task to associate with the exception handler.
`exceptionQ`	A value of type `MPQueueID` (page 110) that specifies the message queue to which an exception message will be sent.
function result	A result code. See "Result Codes" (page 122) for a list of possible values.

DISCUSSION

When an exception handler is set and an exception occurs, the task is suspended and a message is sent to the message queue specified by `exceptionQ`. The message contains the following information:

- The first 32-bits contain the ID of the task in which the exception occurred.
- The second 32-bits contain the type of exception that occurred. See the header file `MachineExceptions.h` for a listing of exception types.
- The last 32-bits are set to `NULL` (reserved for future use).

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

MPThrowException

Throws an exception to a specified task.

```
OSStatus MPThrowException (
                    MPTaskID task,
                    MPExceptionKind kind);
```

`task`	A value of type `MPTaskID` (page 109) that specifies the task to which the exception should be thrown.
`kind`	A value of type `ExceptionKind` that specifies the type of exception to give to the task.
function result	A result code. See "Result Codes" (page 122) for a list of possible values. If the task is already suspended or if the task is not defined to take thrown exceptions, the function returns `kMPInsufficientResourcesErr`.

DISCUSSION

The exception is treated in the same manner as any other exception taken by a task. However, since it is asynchronous, it may not be presented immediately.

By convention, you should set the exception kind to `kMPTaskStoppedErr` if you want to suspend a task. In general, you should do so only if you are debugging and wish to examine the state of the task. Otherwise you should block the task using one of the traditional notification mechanisms (such as a message queue).

An exception can be thrown at any time, whether that task is running, eligible to be run (that is, ready), or blocked. The task is suspended and an exception message may be generated the next time the task is about to run. Note that this may never occur— for example, if the task is deadlocked or the resource it is waiting on is never released. If the task is currently blocked when this function is executed, `kMPTaskBlockedErr` is returned. If the task was suspended immediately at the conclusion of this function call the return value is `kMPTaskStoppedErr`.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

MPDisposeTaskException

Removes a task exception.

```
OSStatus MPDisposeTaskException (
                    MPTaskID task,
                    OptionBits action);
```

`task` A value of type `MPTaskID` (page 109) that specifies the task whose exception you want to remove.

`action` A value of type `OptionBits` that specifies actions to perform on the task. For example, you can enable single-stepping when the task resumes, or you can pass the exception on to another handler. See "Task Exception Disposal Constants" (page 121) for a listing of possible values.

function result A result code. See "Result Codes" (page 122) for a list of possible values. If the specified action is invalid or unsupported, or if the specified task is not suspended, this function returns `kMPInsufficientResourcesErr`.

DISCUSSION

This function removes the task exception and allows the task to resume operation. If desired, you can enable single-stepping or branch-stepping, or propagate the exception instead.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

MPExtractTaskState

Extracts state information from a suspended task.

```
OSStatus MPExtractTaskState (
                    MPTaskID task,
                    MPTaskStateKind kind,
                    void *info);
```

`task` A value of type `MPTaskID` that specifies the task whose state information you want to obtain.

`kind` A value of type `MPTaskStateKind` that specifies the kind of state information you want to obtain. See "Task State Constants" (page 120) for a listing of possible values.

`info` A pointer to a data structure to hold the state information. On return, the data structure holds the desired state information. The format of the data structure varies depending on the state information you want to retrieve. See the header file `MachineExceptions.h` for the formats of the various state information structures.

function result A result code. See "Result Codes" (page 122) for a list of possible values. If you attempt to extract state information for a running task, this function returns `kMPInsufficientResourcesErr`.

DISCUSSION

You can use this function to obtain register contents or exception information about a particular task.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

SEE ALSO

The function `MPSetTaskState` (page 102).

MPSetTaskState

Sets state information for a suspended task.

```
OSStatus MPSetTaskState (
                    MPTaskID task,
                    MPTaskStateKind kind,
                    void *info);
```

`task` A value of type `MPTaskID` (page 109) that specifies the task whose state information you want to set.

`kind` A value of type `MPTaskStateKind` that specifies the kind of state information you want to set. See "Task State Constants" (page 120) for a listing of possible values. Note that some state information is read-only and cannot be changed using this function.

`info` A pointer to a data structure holding the state information you want to set. The format of the data structure varies depending on the state information you want to set. See the header file `MachineExceptions.h` for the formats of the various state information structures.

function result A result code. See "Result Codes" (page 122) for a list of possible values. If you specify `kMPTaskState32BitMemoryException` for the state information, this function returns

kMPInsufficientResourcesErr, since the exception state information is read-only. Attempting to set state information for a running task will also return kMPInsufficientResourcesErr.

DISCUSSION

You can use this function to set register contents or exception information for a particular task. However, some state information, such as the exception information (as specified by kMPTaskState32BitMemoryException) as well as the MSR, ExceptKind, DSISR, and DAR machine registers (specified under kMPTaskStateMachine) are read-only. Attempting to set the read-only machine registers will do nothing, while attempting to set the exception information will return an error.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

SEE ALSO

The function MPExtractTaskState (page 101).

Debugger Support Functions

Multiprocessing Services supplies two functions for use with third-party debuggers:

- MPRegisterDebugger (page 103) registers a debugger.
- MPUnregisterDebugger (page 105) unregisters a debugger.

MPRegisterDebugger

Registers a debugger.

```
OSStatus MPRegisterDebugger (
                    MPQueueID queue,
                    MPDebuggerLevel level);
```

`queue`	A value of type `MPQueueID` (page 110) that specifies the ID of the queue to which you want exception messages and other information to be sent.
`level`	A value of type `MPDebuggerLevel` (page 114) that specifies the level of this debugger with respect to other debuggers. Exceptions and informational messages are sent first to the debugger with the highest level. If more than one debugger attempts to register at a particular level, only the first debugger is registered. Other attempts return an error.
function result	A result code. See "Result Codes" (page 122) for a list of possible values. If the number of registered debuggers exceeds the system limit, the function returns `kMPInsufficientResourcesErr`.

DISCUSSION

Exception messages are sent when tasks are suspended. When a task exception occurs, Multiprocessing Services notifies registered debuggers and other handlers in the following order:

- the registered debugger queue of the highest level
- lower level registered debugger queues according to level
- the local exception handler
- task termination

The notification moves to the next lower level when a debugger (or eventually the task's local exception handler) calls the `MPDisposeTaskException` (page 100) function with the indication that the exception be propagated. The exception messages sent to the debugger's queue are in the same format as those described in `MPSetExceptionHandler` (page 98).

The system may implement a limited number of debugger slots. It is possible no debugger support is provided.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

SEE ALSO

The function `MPUnregisterDebugger` (page 105).

MPUnregisterDebugger

Unregisters a debugger.

```
OSStatus MPUnregisterDebugger (MPQueueID queue);
```

`queue`	A value of type `MPQueueID` (page 110) that specifies the ID of the queue whose debugger you want to unregister.
function result	A result code. See "Result Codes" (page 122) for a list of possible values.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

SEE ALSO

The function `MPRegisterDebugger` (page 103).

Remote Calling Function

If you want to call a nonreentrant function (such as a system software function) from a preemptive task, you must call it remotely using the `MPRemoteCall` (page 106) function.

MPRemoteCall

Calls a nonreentrant function and blocks the current task.

```
void *MPRemoteCall (
                    MPRemoteProcedure remoteProc,
                    void *parameter,
                    MPRemoteContext context);
```

`remoteProc` A pointer of type `MPRemoteProcedure` (page 115) that references the application-defined function you want to call. See `MyRemoteProcedure` (page 107) for more information about the form of this function.

`parameter` A pointer to a parameter to pass to the application-defined function. For example, this value could point to a data structure or a memory location.

`context` A value of type `MPRemoteContext` that specifies which contexts (that is processes) are allowed to execute the function. See "Remote Call Context Option Constants" (page 122) for a list of possible values.

DISCUSSION

You use this function primarily to indirectly execute Mac OS system software functions. The task making the remote call is blocked until the call completes. The amount of time taken to schedule the remote procedure depends on the choice of the designated operating context. Specifying `kMPAnyRemoteContext` offers the lowest latency, but the called procedure may not have access to process-specific resources such as some low-memory values. Specifying `kMPOwningProcessRemoteContext` has higher latency because the remote procedure is deferred until the owning process becomes active. However, the remote procedure is guaranteed to execute within the owning process.

Note that with the exception of functions in Multiprocessing Services, you cannot safely call any system software functions directly from a preemptive task. Even if some system software function appears to work today when called from a preemptive task, unless explicitly stated otherwise there is no guarantee that subsequent versions of the same function will continue to work in future versions of system software. In Mac OS 8 implementations of Multiprocessing Services, the only exceptions to this rule are the atomic memory operations

(such as `AddAtomic`) exported in the `InterfaceLib` shared library. Even these functions may switch to 68K mode if the operands to them are not aligned. If you need to access system software functions from a preemptive task, you must do so using the `MPRemoteCall` function.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

Application-Defined Function

Multiprocessing Services allows you to create an application-defined function such as `MyRemoteProcedure` (page 107) to call nonreentrant functions from your preemptive task.

MyRemoteProcedure

When calling `MPRemoteCall` (page 106), you must designate an application-defined function to handle any calls to nonreentrant functions (such as Mac OS system software calls). For example, this is how you would declare the application-defined function if you were to name the function `MyRemoteProcedure`:

```
void* MyRemoteProcedure (void *parameter);
```

`parameter` A pointer to any information you want to pass to `MyRemoteProcedure`. For example, `parameter` might point to a parameter list that `MyRemoteProcedure` could then pass to a Mac OS system software function.

DISCUSSION

Note that your application-defined function must be PowerPC native code, since Multiprocessing Services tasks cannot call 68K code.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

Data Types

This section describes the data types used with Multiprocessing Services:

MPProcessID

Multiprocessing Services manipulates processes by passing a process ID, which is defined by the `MPProcessID` type:

```
typedef struct OpaqueMPProcessID* MPProcessID;
```

Note that this process ID is identical to the process ID (or context ID) handled by the Code Fragment Manager.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

MPTaskID

Multiprocessing Services manipulates tasks by passing a task ID, which is defined by the `MPTaskID` type:

```
typedef struct OpaqueMPTaskID* MPTaskID;
```

You obtain a task ID by calling the function `MPCreateTask` (page 58).

VERSION NOTES

Introduced with Multiprocessing Services 1.0.

TaskProc

When calling the `MPCreateTask` (page 58) function, you specify the task to create by passing a pointer of type `TaskProc` which references the code to be executed as the task:

```
typedef OSStatus (*TaskProc) (void *parameter);
```

VERSION NOTES

Introduced with Multiprocessing Services 1.0.

MPTaskOptions

When calling the `MPCreateTask` (page 58) function, you can specify any optional attributes for the task created by passing a value of type `MPTaskOptions`, which has the following type definition:

```
typedef OptionBits MPTaskOptions;
```

No options are currently defined.

VERSION NOTES

Introduced with Multiprocessing Services 1.0.

MPTaskWeight

The Multiprocessing Services function `MPSetTaskWeight` (page 63) handles the weight of a task using the `MPTaskWeight` type, which has the following definition.

```
typedef UInt32 MPTaskWeight;
```

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

MPQueueID

Multiprocessing Services manipulates message queues by passing a queue ID, which is defined by the `MPQueueID` type:

```
typedef struct OpaqueMPQueueID* MPQueueID;
```

You obtain a queue ID by calling the function `MPCreateQueue` (page 65).

VERSION NOTES

Introduced with Multiprocessing Services 1.0.

MPSemaphoreID

Multiprocessing Services manipulates semaphores by passing a semaphore ID, which is defined by the `MPSemaphoreID` type:

```
typedef struct OpaqueMPSemaphoreID* MPSemaphoreID;
```

You obtain a semaphore ID by calling the function `MPCreateSemaphore` (page 71).

VERSION NOTES

Introduced with Multiprocessing Services 1.0.

MPEventID

Multiprocessing Services manipulates event groups by passing an event group ID, which is defined by the `MPEventID` type:

```
typedef struct OpaqueMPEventID* MPEventID;
```

You obtain an event group ID by calling the function `MPCreateEvent` (page 74).

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

MPCriticalRegionID

Multiprocessing Services manipulates critical region objects by passing a critical region ID, which is defined by the `MPCriticalRegionID` type:

```
typedef struct OpaqueMPCriticalRegionID*  MPCriticalRegionID;
```

You obtain a critical region ID by calling the function `MPCreateCriticalRegion` (page 78).

VERSION NOTES

Introduced with Multiprocessing Services 1.0.

MPTimerID

Multiprocessing Services manipulates timers by passing a timer ID, which is defined by the `MPTimerID` type:

```
typedef struct OpaqueMPTimerID* MPTimerID;
```

You obtain a timer ID by calling the function `MPCreateTimer` (page 82).

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

MPOpaqueID

The Multiprocessing Services function handles a generic notification ID (that is, an ID that could be a queue ID, event ID, or semaphore ID) by passing an opaque ID, which is defined by the `MPOpaqueID` type:

```
typedef struct OpaqueMPOpaqueID* MPOpaqueID;
```

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

TaskStorageIndex

The Multiprocessing Services functions described in "Accessing Per-Task Storage Variables" (page 90) manipulate task storage index values using the `TaskStorageIndex` type, which has the following definition:

```
typedef UInt32 TaskStorageIndex;
```

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

TaskStorageValue

The Multiprocessing Services functions described in "Accessing Per-Task Storage Variables" (page 90) manipulate task storage values using the `TaskStorageValue` type, which has the following definition:

```
typedef UInt32 TaskStorageValue;
```

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

MPSemaphoreCount

The Multiprocessing Services function `MPCreateSemaphore` (page 71) handles the minimum and maximum semaphore values using the `MPSemaphoreCount` type, which has the following definition.

```
typedef ItemCount MPSemaphoreCount;
```

VERSION NOTES

Introduced with Multiprocessing Services 1.0.

MPEventFlags

The Multiprocessing Services functions described in "Handling Event Groups" (page 74) handle event information using the `MPEventFlags` type, which has the following definition:

```
typedef UInt32 MPEventFlags;
```

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

MPExceptionKind

The Multiprocessing Services function `MPThrowException` (page 99) indicates the kind of exception thrown using the `MPExceptionKind` type, which has the following definition:

```
typedef UInt32 MPExceptionKind;
```

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

MPDebuggerLevel

The Multiprocessing Services function `MPRegisterDebugger` (page 103) indicates the debugger level using the `MPDebuggerLevel` type, which has the following definition:

```
typedef UInt32 MPDebuggerLevel;
```

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

MPRemoteProcedure

When calling the `MPRemoteCall` (page 106) function the remote procedure call (that is, the application-defined function) you want to designate must have the following type definition:

```
typedef void* (*MPRemoteProcedure)(void *parameter);
```

See `MyRemoteProcedure` (page 107) for more information about how to implement the application-defined function.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

Constants

This section describes the constants available with Multiprocessing Services.

- "Timer Duration Constants" (page 116)
- "Timer Option Masks" (page 116)
- "Memory Allocation Alignment Constants" (page 117)
- "Memory Allocation Option Constants" (page 119)
- "Task State Constants" (page 120)
- "Task Exception Disposal Constants" (page 121)
- "Remote Call Context Option Constants" (page 122)

Timer Duration Constants

Several Multiprocessing Services functions take a parameter of type `Duration`, which specifies the maximum time a task should wait for an event to occur. Multiprocessing Services recognizes four constants which you can use when specifying a duration. Note that you can use these constants in conjunction with other values to indicate specific wait intervals. For example, to wait 1 second, you can pass `kDurationMillisecond * 1000`.

```
enum {
    kDurationImmediate      = 0L,
    kDurationForever        = 0x7FFFFFFF,
    kDurationMillisecond    = 1,
    kDurationMicrosecond    = -1
    };
```

Constant Descriptions

`kDurationImmediate`
: The task times out immediately, whether or not the event has occurred. If the event occurred, the return status is `noErr`. If the event did not occur, the return status is `kMPTimeoutErr` (assuming no other errors occurred).

`kDurationForever`
: The task waits forever. The blocking call waits until either the event occurs, or until the object being waited upon (such as a message queue) is deleted.

`kDurationMillisecond`
: The task waits one millisecond before timing out.

`kDurationMicrosecond`
: The task waits one microsecond before timing out.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

Timer Option Masks

When calling `MPArmTimer` (page 86), you can pass values of type `OptionBits` to specify any optional actions.

```
enum {
    kMPPreserveTimerIDMask      = 1L << 0,
    kMPTimeIsDeltaMask          = 1L << 1,
    kMPTimeIsDurationMask       = 1L << 2
    };
```

Constant Descriptions

`kMPPreserveTimerID`

Specifying this mask prevents the timer from being deleted when it expires.

`kMPTimeIsDeltaMask`

Specifying this mask indicates that the specified time should be added to the previous expiration time to form the new expiration time. You can use this mask to compensate for timing drift caused by the finite amount of time required to arm the timer, receive the notification, and so on.

`kMPTimeIsDurationMask`

Specifying this mask indicates that the specified expiration time is of type `Duration`. You can use this mask to avoid having to call time conversion routines when specifying an expiration time.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

Memory Allocation Alignment Constants

When calling the `MPAllocateAligned` (page 94) function, you must specify the alignment of the desired memory block by passing a constant of type `UInt8` in the `alignment` parameter.

```
enum {
    /* Values for the alignment parameter to MPAllocateAligned.*/
    kMPAllocateDefaultAligned   = 0,
    kMPAllocate8ByteAligned     = 3,
    kMPAllocate16ByteAligned    = 4,
    kMPAllocate32ByteAligned    = 5,
```

```
    kMPAllocate1024ByteAligned  = 10,
    kMPAllocate4096ByteAligned  = 12,
    kMPAllocateMaxAlignment     = 16,
    kMPAllocateAltiVecAligned   = kMPAllocate16ByteAligned,
    kMPAllocateVMXAligned       = kMPAllocateAltiVecAligned
    kMPAllocateVMPageAligned    = 254,
    kMPAllocateInterlockAligned = 255
    };
```

Constant Descriptions

`kMPAllocateDefaultAligned`
Use the default alignment.

`kMPAllocate8ByteAligned`
Use 8-byte alignment.

`kMPAllocate16ByteAligned`
Use 16-byte alignment.

`kMPAllocate32ByteAligned`
Use 32-byte alignment.

`kMPAllocate1024ByteAligned`
Use 1024-byte alignment.

`kMPAllocate4096ByteAligned`
Use 4096-byte alignment.

`kMPAllocateMaxAlignment`
Use the maximum alignment (65536 byte).

`kMPAllocateAltiVecAligned`
Use AltiVec alignment.

`kMPAllocateVMXAligned`
Use VMX (now called AltiVec) alignment.

`kMPAllocateVMPageAligned`
Use virtual memory page alignment. This alignment is set at runtime.

`kMPAllocateInterlockAligned`
Use interlock alignment, which is the alignment needed to allow the use of CPU interlock instructions (that is, `lwarx` and `stwcx`.) on the returned memory address. This alignment is set at runtime. In most cases you would never need to use this alignment.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

Memory Allocation Option Constants

When calling the `MPAllocateAligned` (page 94) function, you can specify optional actions by passing a constant of type `OptionBits` in the `options` parameter.

```
enum {
    /* Values for the options parameter to MPAllocateAligned.*/
    kMPAllocateClearMask    = 0x0001,
    kMPAllocateGloballyMask = 0x0002,
    kMPAllocateResidentMask = 0x0004,
    kMPAllocateNoGrowthMask = 0x0010
    };
```

Constant Descriptions

`kMPAllocateClearMask`
: Zero out the allocated memory block.

`kMPAllocateGloballyMask`
: Allocate memory from in memory space that is visible to all processes. Note that such globally-allocated space is not automatically reclaimed when the allocating process terminates. By default, `MPAllocateAligned` (page 94) allocates memory from process-specific (that is, not global) memory.

`kMPAllocateResidentMask`
: Allocate memory from resident memory only (that is, the allocated memory is not pageable).

`kMPAllocateNoGrowthMask`
: Do not attempt to grow the pool of available memory. Specifying this option is useful, as attempting to grow memory may cause your task to block until such memory becomes available.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

Task State Constants

When calling the `MPExtractTaskState` (page 101) or `MPSetTaskState` (page 102) functions you must specify what states you want to obtain or set by passing a constant of type `TaskStateKind` in the `kind` parameter.

```
enum {
    kMPTaskStateRegisters   = 0,
    kMPTaskStateFPU         = 1,
    kMPTaskStateVectors     = 2,
    kMPTaskStateMachine     = 3,
    kMPTaskState32BitMemoryException= 4
    };
typedef UInt32 MPTaskStateKind;
```

Constant Descriptions

`kMPTaskStateRegisters`
: The task's general-purpose (GP) registers. The `RegisterInformationPowerPC` structure in `MachineExceptions.h` defines the format of this information.

`kMPTaskStateFPU`
: The task's floating point registers. The `FPUInformationPowerPC` structure in `MachineExceptions.h` defines the format of this information.

`kMPTaskStateVectors`
: The task's vector registers. The `VectorInformationPowerPC` structure in `MachineExceptions.h` defines the format of this information.

`kMPTaskStateMachine`
: The task's machine registers. The `MachineInformationPowerPC` structure in `MachineExceptions.h` defines the format of this information. Note that the MSR, ExceptKind, DSISR, and DAR registers are read-only.

`kMPTaskState32BitMemoryException`
: The task's exception information for older 32-bit memory

exceptions (that is, memory exceptions on 32-bit CPUs). The `MemoryExceptionInformation` structure in `MachineExceptions.h` defines the format of this information. This exception information is read-only.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

Task Exception Disposal Constants

When calling the `MPDisposeTaskException` (page 100) function, you must pass a constant of type `OptionBits` in the `action` parameter specifying what actions to take.

```
enum {
    /* Option bits for MPDisposeTaskException.*/
    kMPTaskResumeMask       = 0x0000,
    kMPTaskPropagateMask    = 1 << kMPTaskPropagate,
    kMPTaskResumeStepMask   = 1 << kMPTaskResumeStep,
    kMPTaskResumeBranchMask = 1 << kMPTaskResumeBranch
    };
```

Constant Descriptions

`kMPTaskResumeMask` Resume the task.

`kMPTaskPropagateMask`
Propagate the exception to the next debugger level.

`kMPTaskResumeStepMask`
Resume the task and enable single stepping.

`kMPTaskResumeBranchMask`
Resume the task and enable branch stepping.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

Remote Call Context Option Constants

When making a remote call to an application-defined function using `MPRemoteCall` (page 106), you must pass a constant of type `MPRemoteContext` in the `context` parameter specifying which contexts are allowed to execute the function.

```
enum {
    kMPAnyRemoteContext             = 0,
    kMPOwningProcessRemoteContext   = 1
};
typedef UInt8 MPRemoteContext;
```

Constant Descriptions

`kMPAnyRemoteContext`
Any cooperative context can execute the function. Note that the called function may not have access to any of the owning context's process-specific low-memory values.

`kMPOwningProcessRemoteContext`
Only the context that owns the task can execute the function.

VERSION NOTES

Introduced with Multiprocessing Services 2.0.

Result Codes

Many Multiprocessing Services functions return result codes. In addition, Multiprocessing Services functions may also return File Manager, Code Fragment Manager, and Process Manager result codes, which are described in *Inside Macintosh.*

The most common result codes returned by Multiprocessing Services are listed below

`noErr`	No error
`paramErr`	Invalid parameter in function call
`memFullErr`	Out of memory
`kMPDeletedErr`	The desired notification the function was waiting upon was deleted.
`kMPInsufficientResourcesErr`	Could not complete task due to unavailable Multiprocessing Services resources. Note that many functions return this value as a general error when the desired action could not be performed.
`kMPInvalidID`	Invalid ID value (for example, an invalid message queue ID was passed to `MPNotifyQueue` (page 66).
`kMPTaskBlockedErr`	The desired task is blocked.
`kMPTaskStoppedErr`	The desired task is stopped.
`kMPTimeoutErr`	The designated timeout interval passed before the function could take action.

Calculating the Intertask Signaling Time

When using Multiprocessing Services tasks, the amount of time used by the task should be much greater than the time taken to pass notifications to the task. This intertask signaling time is generally between 20 and 50 microseconds. If you want to explicitly calculate the signaling time, you can use the code in Listing A-1 to do so.

Listing A-1 Calculating the intertask signaling time

```
#include <Multiprocessing.h>

#include <Types.h>
#include <stdio.h>
#include <stdlib.h>
#include <sioux.h>
#include <math64.h>
#include <DriverServices.h>


enum  {
    aQueue = 0,
    aSemaphore,
    anEvent
} reflectOP;

MPOpaqueID waiterID, postID;


static OSStatus Reflector ( )
{
    void*p1,*p2,*p3;
    MPEventFlags events;
```

```
    while (true)
    {
        switch (reflectOP)
        {
            case aQueue:
                MPWaitOnQueue((MPQueueID) waiterID, &p1, &p2, &p3, kDurationForever);
                break;

            case aSemaphore:
                MPWaitOnSemaphore((MPSemaphoreID) waiterID, kDurationForever);
                break;

            case anEvent:
                MPWaitForEvent((MPEventID) waiterID, &events, kDurationForever);
                break;

            default:
                return -123;
        }

        switch (reflectOP)
        {
            case aQueue:
                MPNotifyQueue((MPQueueID) postID, &p1, &p2, &p3);
                break;

            case aSemaphore:
                MPSignalSemaphore((MPSemaphoreID) postID);
                break;

            case anEvent:
                MPSetEvent((MPEventID) postID, 0x01010101);
                break;

            default:
                return -123;
        }
    }

    return -123;
}
```

```
static float HowLong(
    AbsoluteTime endTime,
    AbsoluteTime bgnTime
    )
{
    AbsoluteTime absTime;
    Nanoseconds  nanosec;

    absTime = SubAbsoluteFromAbsolute(endTime, bgnTime);
    nanosec = AbsoluteToNanoseconds(absTime);
    return (float) UnsignedWideToUInt64( nanosec ) / 1000.0;
}


void main ( void )
{
    OSStatus    err;
    MPTaskID    task;
    UInt32          i, count;
    void            *p1,*p2,*p3;
    MPEventFlags events;
    AbsoluteTime nowTime, bgnTime;
    float           uSec;
    char            buff[10];


    /* Set the console window defaults */
    /* (this is a Metrowerks CodeWarrior thing). */
    SIOUXSettings.autocloseonquit = true;
    SIOUXSettings.asktosaveonclose = false;
    SIOUXSettings.showstatusline = false;
    SIOUXSettings.columns = 100;
    SIOUXSettings.rows = 20;
    SIOUXSettings.fontsize = 10;
//  SIOUXSettings.fontid = monaco;
    SIOUXSettings.standalone = true;


//  DebugStr ( "\pStarting" );
```

```
/*  Can't get very far without this one. */
if  (!MPLibraryIsLoaded())
{
    printf("The MP library did not load.\n");
    return;
}


/*  Find the overhead up UpTime. Perform a bunch of calls to average out */
/*   cache effects. */
printf("\n");

bgnTime = UpTime();
for (i=0; i<16; i++)
{
    nowTime = UpTime();
}

uSec  = HowLong(nowTime, bgnTime);
uSec /=  16.0;
printf(" UpTime overhead: %.3f usec \n", uSec);

/*  Time intertask communication. */
printf("\n Queues\n");

reflectOP = aQueue;
MPCreateQueue((MPQueueID*) &waiterID);
MPCreateQueue((MPQueueID*) &postID);

bgnTime = UpTime();
err = MPCreateTask( Reflector,
              NULL,
              0,
              NULL,
              0,
              0,
              kNilOptions,
              &task );
nowTime = UpTime();
uSec  = HowLong(nowTime, bgnTime);
```

```
printf(" MPCreateTask overhead: %.3f usec (may vary significantly) \n", uSec);
if (err != noErr)
{
    printf(" Task not created!\n");
    return;
}

count = 100000;
bgnTime = UpTime();
for (i=0; i<count; i++)
{
    MPNotifyQueue((MPQueueID) waiterID, 0, 0, 0);
    while (true)
    {
        err = MPWaitOnQueue((MPQueueID) postID, &p1, &p2, &p3, kDurationImmediate);
        if (err != kMPTimeoutErr) break;
    }
}
nowTime = UpTime();
uSec  = HowLong(nowTime, bgnTime);
uSec  /= ((float) count / 2.0); // Two trips.
printf(" Intertask signalling using queues overhead: %.3f usec \n", uSec);

/*  Time intertask communication.
*/
MPTerminateTask(task, 123);
printf("\n Semaphores\n");

reflectOP = aSemaphore;

MPCreateSemaphore(1, 0, (MPSemaphoreID*) &waiterID);
MPCreateSemaphore(1, 0, (MPSemaphoreID*) &postID);

bgnTime = UpTime();
err = MPCreateTask( Reflector,
              NULL,
              0,
              NULL,
              0,
              0,
              kNilOptions,
```

```
                &task );
nowTime = UpTime();
uSec  = HowLong(nowTime, bgnTime);
printf(" MPCreateTask overhead: %.3f usec (may vary significantly) \n", uSec);
if (err != noErr)
{
    printf(" Task not created!\n");
    return;
}

count = 100000;
bgnTime = UpTime();
for (i=0; i<count; i++)
{
    MPSignalSemaphore((MPSemaphoreID) waiterID);
    while (true)
    {
        err = MPWaitOnSemaphore((MPSemaphoreID) postID, kDurationImmediate);
        if (err != kMPTimeoutErr) break;
    }
}
nowTime = UpTime();
uSec  = HowLong(nowTime, bgnTime);
uSec  /= ((float) count / 2.0);// Two trips.
printf(" Intertask signalling using sempahores overhead: %.3f usec \n", uSec);


/*  Time intertask communication. */
MPTerminateTask(task, 123);
printf("\n Event Groups\n");

reflectOP = anEvent;
MPCreateEvent((MPEventID*) &waiterID);
MPCreateEvent((MPEventID*) &postID);

bgnTime = UpTime();
err = MPCreateTask( Reflector,
                NULL,
                0,
                NULL,
                0,
```

```
                0,
                kNilOptions,
                &task );
    nowTime = UpTime();
    uSec  = HowLong(nowTime, bgnTime);
    printf(" MPCreateTask overhead: %.3f usec (may vary significantly) \n", uSec);
    if (err != noErr)
    {
        printf(" Task not created!\n");
        return;
    }

    count = 100000;
    bgnTime = UpTime();
    for (i=0; i<count; i++)
    {
        MPSetEvent((MPEventID) waiterID, 0x01);
        while (true)
        {
            err = MPWaitForEvent((MPEventID) postID, &events, kDurationImmediate);
            if (err != kMPTimeoutErr) break;
        }
    }
    nowTime = UpTime();
    uSec  = HowLong(nowTime, bgnTime);
    uSec  /= ((float) count / 2.0);// Two trips.
    printf(" Intertask signalling using events overhead: %.3f usec \n", uSec);

    gets(buff);

}
```

Changes From Previous Versions of Multiprocessing Services

Multiprocessing Services 2.0 supports some, but not all, functions available in earlier releases. Table B-1 lists the functions that were introduced in version 1.0 that are still supported in version 2.0.

Table B-1 Older functions supported in version 2.0

Name	Comments
`MPProcessors` (page 56)	
`MPCreateTask` (page 58)	
`MPTerminateTask` (page 60)	
`MPCurrentTaskID` (page 62)	
`MPYield` (page 62)	
`MPExit` (page 61)	
`MPCreateQueue` (page 65)	
`MPDeleteQueue` (page 66)	
`MPNotifyQueue` (page 66)	
`MPWaitOnQueue` (page 68)	
`MPCreateSemaphore` (page 71)	
`MPCreateBinarySemaphore`	In C, a macro that calls `MPCreateSemaphore` (page 71).
`MPDeleteSemaphore` (page 72)	
`MPSignalSemaphore` (page 72)	
`MPWaitOnSemaphore` (page 73)	
`MPCreateCriticalRegion` (page 78)	

Table B-1 Older functions supported in version 2.0 (continued)

Name	Comments
`MPDeleteCriticalRegion` (page 79)	
`MPEnterCriticalRegion` (page 79)	
`MPExitCriticalRegion` (page 81)	
`MPAllocate`	Deprecated. Use `MPAllocateAligned` instead.
`MPFree` (page 95)	
`MPBlockCopy` (page 96)	
`MPLibraryIsLoaded`	In C, a macro that checks to see if the `MPProcessors` symbol is resolved.

Table B-2 lists Multiprocessing Services functions that are new in version 2.0.

Table B-2 New functions introduced with version 2.0

Name	Comments
`MPProcessorsScheduled` (page 57)	
`MPSetTaskWeight` (page 63)	
`MPTaskIsPreemptive` (page 64)	
`MPAllocateTaskStorageIndex` (page 90)	
`MPDeallocateTaskStorageIndex` (page 91)	
`MPSetTaskStorageValue` (page 91)	
`MPGetTaskStorageValue` (page 92)	
`MPSetQueueReserve` (page 69)	
`MPCreateEvent` (page 74)	
`MPDeleteEvent` (page 75)	
`MPSetEvent` (page 76)	

Table B-2 New functions introduced with version 2.0 (continued)

Name	Comments
`MPWaitForEvent` (page 77)	
`UpTime` (page 88)	
`DurationToAbsolute` (page 88)	
`AbsoluteToDuration` (page 89)	
`MPDelayUntil` (page 82)	
`MPCreateTimer` (page 82)	
`MPDeleteTimer` (page 83)	
`MPSetTimerNotify` (page 84)	
`MPArmTimer` (page 86)	
`MPCancelTimer` (page 87)	
`MPSetExceptionHandler` (page 98)	
`MPThrowException` (page 99)	
`MPDisposeTaskException` (page 100)	
`MPExtractTaskState` (page 101)	
`MPSetTaskState` (page 102)	
`MPRegisterDebugger` (page 103)	
`MPRegisterDebugger` (page 103)	
`MPAllocateAligned` (page 94)	Preferred over MPAllocate.
`MPGetAllocatedBlockSize` (page 95)	
`MPBlockClear` (page 97)	
`MPDataToCode` (page 97)	
`MPRemoteCall` (page 106)	Preferred over `_MPRPC`

Table B-3 shows unofficial functions included in earlier header files that remain supported in version 2.0. Note, however, that future versions may not support these functions.

Table B-3 Unofficial functions still supported in version 2.0

Name	Comments
`_MPRPC`	Deprecated. Use `MPRemoteCall` (page 106) instead.
`_MPAllocateSys`	Deprecated. Use `MPAllocateAligned` (page 94) instead.
`_MPTaskIsToolboxSafe`	
`_MPLibraryVersion`	
`_MPLibraryIsCompatible`	

Table B-4 shows functions used for debugging that are no longer supported in version 2.0. You can access these functions for older builds if you `#define MPIncludeDefunctServices` to be nonzero.

Table B-4 Debugging functions unsupported in version 2.0

Name	Comments
`_MPInitializePrintf`	
`_MPPrintf`	
`_MPDebugStr`	
`_MPStatusPString`	
`_MPStatusCString`	

Document Version History

This document has had the following releases:

Table C-1 Multiprocessing Services documentation revision history

Version	Notes
April 30, 1999	Initial public release. The following changes were made from the previous (seed draft) version: Added Chapter 1, "Introduction," Chapter 2, "About Multitasking on the Mac OS," and Chapter 3, "Using Multiprocessing Services," which include introductory information, conceptual information, programming discussions, and sample code. Added versioning information to functions, data types, and constants in Chapter 4, "Multiprocessing Services Reference." Added discussion and parameter information to `MPTaskIsPreemptive` (page 64) indicating how and why you can specify `kInvalidID` to determine the preemptiveness of the current task. Also added Version Notes section. Correction in `MPWaitOnQueue` (page 68), `MPWaitOnSemaphore` (page 73), `MPWaitForEvent` (page 77), and `MPEnterCriticalRegion` (page 79): When calling from a cooperative task, you should specify only `kDurationImmediate` waits; others are allowable, but they will cause the task to block. Added information stating that setting event bits in `MPSetEvent` (page 76) and obtaining and clearing and event group in `MPWaitForEvent` (page 77) are atomic operations. For example, bits cannot be set between when a task obtains an event group and when the event group is cleared, so no data can be lost. Changed wording in `MPDelayUntil` (page 82) to clarify that you must indicate a specific time to unblock the task, not a duration. Changed wording for `MPAllocateTaskStorageIndex` (page 90) and `MPDeallocateTaskStorageIndex` (page 91) to indicate that these functions do not actually allocate or deallocate memory.

Table C-1 Multiprocessing Services documentation revision history (continued)

Version	Notes
	Added disclaimer to `MPThrowException` (page 99) indicating that you should throw an exception to a task to stop it only if you are debugging and plan to examine the state of the task. Otherwise, you should block the task using a traditional notification method (such as a message queue).
	Modified discussion of `MPSetExceptionHandler` (page 98) to indicate the format of the message sent to the exception handler.
	Discussion of informative messages in `MPRegisterDebugger` (page 103) removed to reflect status as of version 2.0.
	Added information to `MPExtractTaskState` (page 101) and `MPSetTaskState` (page 102) indicating that attempting to set or read state information for a nonsuspended task returns the error `kMPInsufficientResourcesErr`.
	Added information to `MPSetTaskState` (page 102) and "Task State Constants" (page 120): the exception state information and some machine registers (MRS, ExceptKind, DSISR, and DAR) are read-only. Attempting to set the exception state information will return an error. Attempts to change the MRS, ExceptKind, DSISR, and DAR registers will simply have no effect.
	Discussion in `MPRemoteCall` (page 106) modifed to reflect this clarification: If you specify that the function should execute in the same context that owns the task, the function has access to data available to the main application (just as if the application had called the function). However, the function cannot execute until the owning context becomes active (and then not until the application calls `WaitNextEvent`).
	Atomic memory operations mentioned in `MPRemoteCall` (page 106) and "Making Remote Procedure Calls" (page 47) are now located in `InterfaceLib`, not the Driver Services Library.
	Data types `MPAddressSpaceID` and `MPCpuID` removed to reflect status as of version 2.0.
	Clarified that you can use the constants in "Timer Duration Constants" (page 116) to specify any number of waiting times by adding multipliers.
	Correction in "Memory Allocation Alignment Constants" (page 117): CPU interlock instruction `swarx` should be `stwcx`.

Table C-1 Multiprocessing Services documentation revision history (continued)

Version	Notes
	Specified the `MachineExceptions.h` structures that correspond to the state information constants in "Task State Constants" (page 120)
	Removed nonbitmask values (`kMPTaskPropagate`, `kMPTaskResumeStep`, and `kMPTaskResumeBranch`) and descriptions from "Task Exception Disposal Constants" (page 121).
Feb. 26, 1999	First seed draft release

Glossary

atomic operation A n action that executes as one indivisible sequence. For example, obtaining and clearing an event group occurs as an atomic operation. It is not possible for another task to set an event bit *after* the event group is obtained but *before* it is cleared.

cooperative multitasking A multitasking model that requires a task to voluntarily suspend its execution in order to give processor time to other tasks.

blocked The state where a task is not executing because it is waiting for data or a resource to become available. Blocked tasks are not assigned processor time by the scheduler.

context The execution environment for an application. Each running application has its own context which is scheduled (cooperatively or preemptively) by system software. For example, in Mac OS 8, contexts are cooperatively-scheduled by the Process Manager. Also called a *process* or an *execution context*.

critical region A section of executable code that can only be accessed by one task at a time.

event group A set of flags which you can use to signal events for a task.

latency A significant time delay between when your code requests an action and when the action actually occurs. For example, there is typically a latency between when you request that a preemptive task be terminated and when it actually terminates.

Mac OS task The preemptive task that contains all the cooperatively scheduled programs handled by the Process Manager.

message A 96-bit block of data that passes information to a task. Messages are passed to message queues which then can be read by a task. *See also* message queue.

message queue A storage location for messages. Tasks can place or retrieve messages from a queue. *See also* message.

multiprocessing The ability of a computer and operating system to use more than one microprocessor at a time.

Multiprocessing Services Apple Computer's preemptive multitasking technology for Macintosh computers.

multitasking The ability to handle several tasks at a time.

notification mechanism An indicator that passes information to a task. Some notification mcchanisms include messages, semaphores, and event groups.

nonreentrant function A function that cannot be called simultaneously or sequentially by different preemptive tasks. Typically functions that manipulate global data are nonreentrant.

preemptive multitasking The ability of an operatiing system to divide processor time among many tasks, allowing them to execute in a simultaneous or near-simultaneous manner.

preemptive scheduling A multitasking model that allows a scheduler to determine the amount of processor time to assign to tasks. The scheduler uses well-defined rules to determine when to stop execution of one task and resume another.

process *See* **context**.

remote procedure call A call by a preemptive task that must execute in a nonpreemptive environment. Typically you use remote procedure calls to call nonreentrant functions.

scheduler An authority that assigns processor time to individual tasks. Also called a *task scheduler*.

semaphore A counter that can be incremented or decremented to signal an event to a task.

task An individual unit of program execution. Each task has its own stack and register set.

symmetric multiprocessing A multiprocessing technique where tasks are divided equally among all available processsors. This method is more efficient than *asymmetric multiprocessing* where one master processor assigns work to a number of slave processors.

thread As defined for the Mac OS operating system, a task that is cooperatively-scheduled by the Thread Manager. Compare **task.**

Index

M

N

P

Q

R

S

T

U

V

W

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Line art was created using Adobe™ Illustrator and Adobe Photoshop.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Adobe Letter Gothic.

WRITER
Jun Suzuki

ILLUSTRATOR
Ruth Anderson

DEVELOPMENTAL EDITOR
Laurel Rezeau

PRODUCTION EDITOR
Gerri Gray

Special thanks to René Vega, Alan Lillich, Jim Murphy, Roger Pantos, and George Warner.