

---

# Multiprocessing Services Programming Guide

[Carbon](#) > [Process Management](#)



2007-10-31



Apple Inc.  
© 2007 Apple Inc.  
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

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

Apple, the Apple logo, Carbon, Mac, Mac OS, Macintosh, MPW, and Power Mac are trademarks of Apple Inc., registered in the United States and other countries.

PowerPC and the PowerPC logo are trademarks of International Business Machines Corporation, used under license therefrom.

UNIX is a registered trademark of The Open Group

Simultaneously published in the United States and Canada.

**Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY,**

**MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.**

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

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

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

# Contents

---

**Introduction      Introduction to Multiprocessing Services Programming Guide   7**

---

Organization of This Document   7  
See Also   8

---

**Chapter 1      About Multitasking on the Mac OS   9**

---

Multitasking Basics   9  
Multitasking and Multiprocessing   10  
Tasks and Address Spaces   10  
Task Scheduling   11  
Shared Resources and Task Synchronization   12  
    Semaphores   12  
    Message Queues   13  
    Event Groups   13  
    Kernel Notifications   13  
    Critical Regions   14  
Tasking Architectures   14  
    Multiple Independent Tasks   15  
    Parallel Tasks With Parallel I/O Buffers   15  
    Parallel Tasks With a Single Set of I/O Buffers   16  
    Sequential Tasks   17

---

**Chapter 2      Using Multiprocessing Services   19**

---

Multiprocessing Services in Mac OS 9 and Mac OS X   19  
Compatibility with Older System Software   19  
Criteria for Creating Tasks   20  
Checking for the Availability of Multiprocessing Services   20  
Determining the Number of Processors   21  
Creating Tasks   21  
Terminating Tasks   25  
Synchronizing and Notifying Tasks   26  
    Handling Periodic Actions   28  
    Notifying Tasks at Interrupt Time   29  
    Using Critical Regions   29  
Allocating Memory in Tasks   30  
Using Task-Specific Storage   30  
Using Timers   30  
Making Remote Procedure Calls   31  
Handling Exceptions and Debugging   32

**Appendix A**      **Preemptive Task–Safe Mac OS System Software Functions** 35

---

**Appendix B**      **Calculating the Intertask Signaling Time** 43

---

**Appendix C**      **Changes From Previous Versions of Multiprocessing Services** 49

---

**Document Revision History** 53

---

**Index** 57

---

# Figures, Tables, and Listings

## Chapter 1 About Multitasking on the Mac OS 9

---

Figure 1-1	Tasks within processes	10
Figure 1-2	The Mac OS task and other preemptive tasks	11
Figure 1-3	Parallel tasks with parallel I/O buffers	16
Figure 1-4	Parallel tasks with a single set of I/O buffers	16
Figure 1-5	Sequential Tasks	17

## Chapter 2 Using Multiprocessing Services 19

---

Listing 2-1	Creating tasks	21
Listing 2-2	A sample task	23
Listing 2-3	Terminating tasks	25
Listing 2-4	Assigning work to tasks	26
Listing 2-5	Using a semaphore to perform periodic actions	28
Listing 2-6	Performing actions periodically and on demand	28

## Appendix A Preemptive Task–Safe Mac OS System Software Functions 35

---

Table A-1	Preemptive task–safe Device Manager functions	35
Table A-2	Preemptive task–safe HFS Plus parameter block functions	36
Table A-3	Preemptive task–safe HFS Plus file system reference functions	37
Table A-4	Preemptive task–safe HFS parameter block functions	38
Table A-5	Preemptive task–safe high-level HFS functions	41
Table A-6	Preemptive task–safe file system specification functions	42
Table A-7	Preemptive task–safe functions shared by the Device Manager and File Manager	42
Table A-8	Miscellaneous preemptive task–safe functions	42

## Appendix B Calculating the Intertask Signaling Time 43

---

Listing B-1	Calculating the intertask signaling time	43
-------------	--	----

## Appendix C Changes From Previous Versions of Multiprocessing Services 49

---

Table C-1	New functions introduced with version 2.1	49
Table C-2	Functions introduced with version 2.0	49
Table C-3	Older functions supported in version 2.0	50
Table C-4	Unofficial functions still supported in version 2.0	51
Table C-5	Debugging functions unsupported in version 2.0	52



# Introduction to Multiprocessing Services Programming Guide

---

**Note:** This book was previously titled *Adding Multitasking Capability to Applications using Multiprocessing Services*.

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 2.1 . This technology is available with Mac OS 9.0 and later (including Mac OS X), 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*.

## Organization of This Document

This document covers Multiprocessing Services in the following chapters:

- [“About Multitasking on the Mac OS”](#) (page 9) 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 19) contains programming examples and other detailed information about adding Multiprocessing Services to your application.
- [“Preemptive Task–Safe Mac OS System Software Functions”](#) (page 35) lists Mac OS system software functions that you can call from preemptive tasks.
- [“Calculating the Intertask Signaling Time”](#) (page 43) contains sample code you can use to determine the amount of time it takes to notify a task.
- [“Changes From Previous Versions of Multiprocessing Services”](#) (page 49) describes changes and additions to the Multiprocessing Services API between version 1.4 and 2.1.

## See Also

For more information about multithreading on Mac OS X, see *Threading Programming Guide* in Carbon Process Management documentation.

For additional information about Multiprocessing Services, you should check the Apple Developer Web site: <http://developer.apple.com>



# About Multitasking on the Mac OS

---

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 19).

## 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<sup>®</sup>. 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 them. 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.

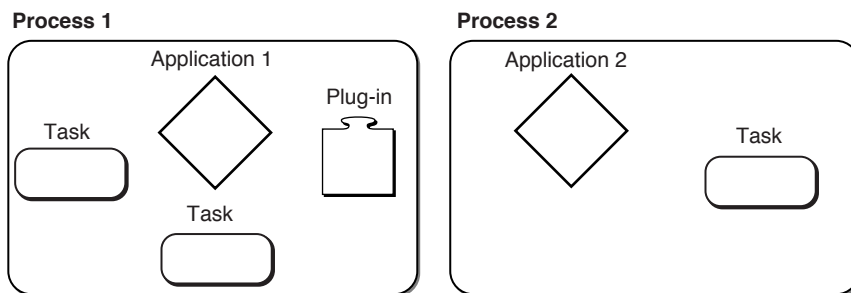
**Note:** Because multitasking allows an operating system to attend to several different operations at the same time, these operations may appear to occur simultaneously on even a single-processor system, due to the speed of the processor.

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, non-global data, and so on. Tasks created with Multiprocessing Services are automatically associated with the creating application's process, as shown in Figure 1-1.

**Figure 1-1** 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.

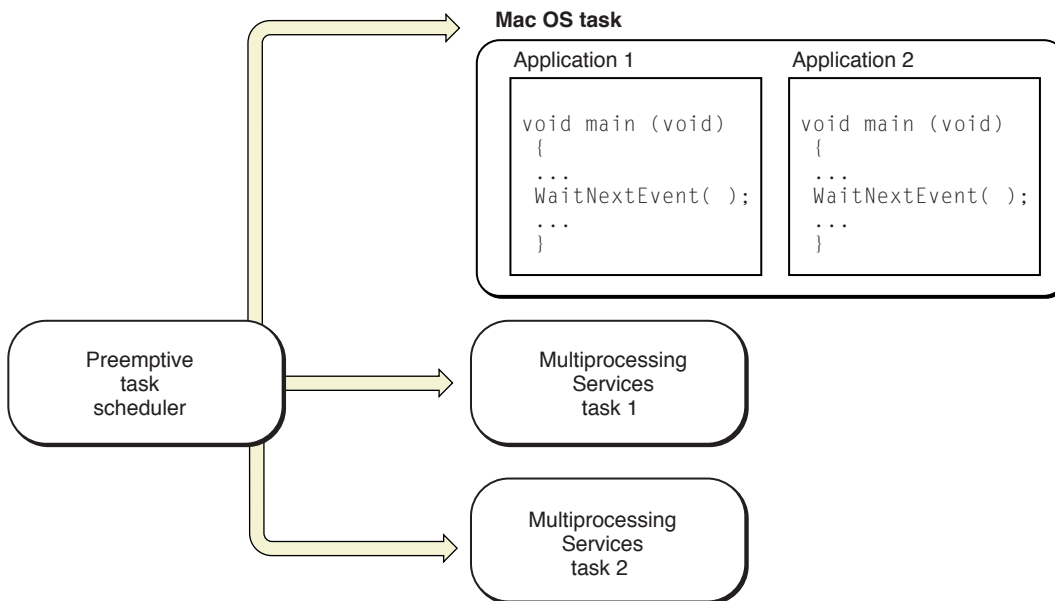
**Important:** Although all processes share the same address space in Mac OS 9.0, you should not assume that this will remain the case; your application or task should not attempt to access data or code residing in another process.

## 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 9.0 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 1-2.

**Figure 1-2** 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 12) are ideal for this purpose.

**Note:** In the future, application tasks will run as individual preemptive tasks, rather than within the Mac OS task. However, calls to non-reentrant 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 31) 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 four notification mechanisms (or signaling mechanisms), which 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 signaling 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.

## Kernel Notifications

---

A kernel notification is a set of simple notification mechanisms (semaphores, message queues, and event groups) which can be notified with only one signal. For example, a kernel notification might contain both a semaphore and a message queue. When you signal the kernel notification, the semaphore is signaled and a message is sent to the specified queue. A kernel notification can contain one of each type of simple notification mechanism.

You use kernel notifications to hide complexity from the signaler. For example, say a server has three queues to process, ranked according to priority (queue 1 holds the most important tasks, queue 2 holds lesser priority tasks, and queue 3 holds low priority tasks). The server can wait on an event group, which indicates when a task is posted to a queue.

If you do not use a kernel notification, then when a client has a message to deliver, it must both send the message to the appropriate queue and signal the event group with the correct priority value. Doing so requires the client to keep track of which queue to send to as well as the proper event group bit to set.

A simpler method is to set up three kernel notifications, one for each priority. To signal a high priority message, the client simply loads the message and signals the high priority kernel notification.

## 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.

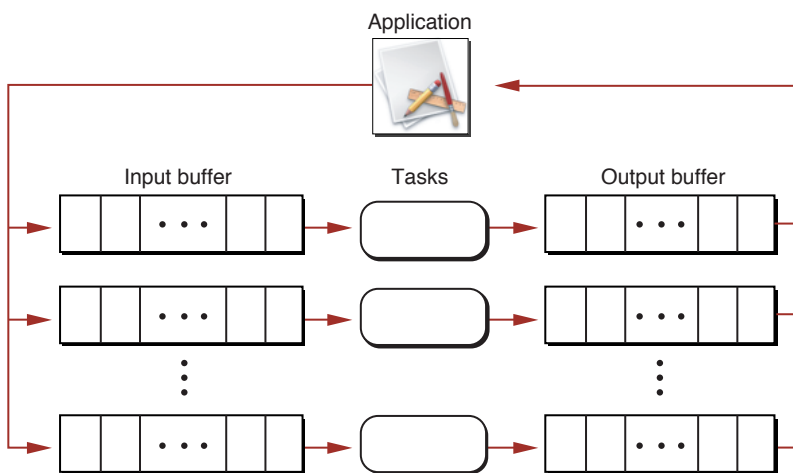
## Parallel Tasks With Parallel I/O Buffers

---

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 1-3, 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.

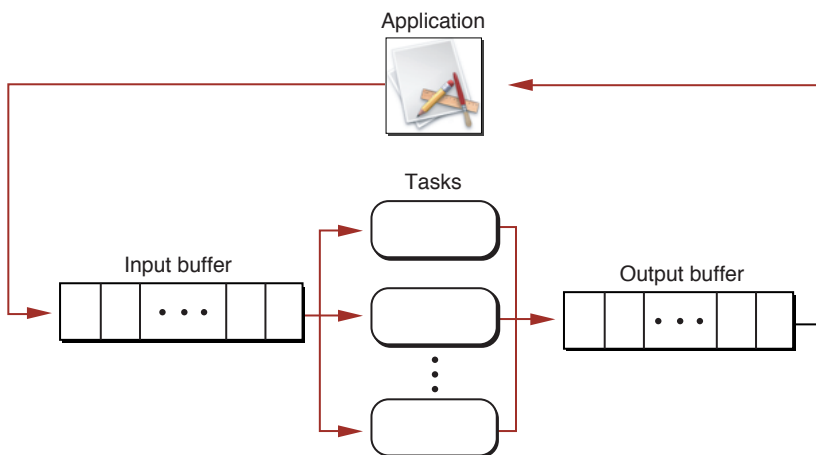
**Figure 1-3** Parallel tasks with parallel I/O buffers

## 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 1-4 illustrates this “bank line” tasking architecture.

**Figure 1-4** Parallel tasks with a single set of I/O buffers



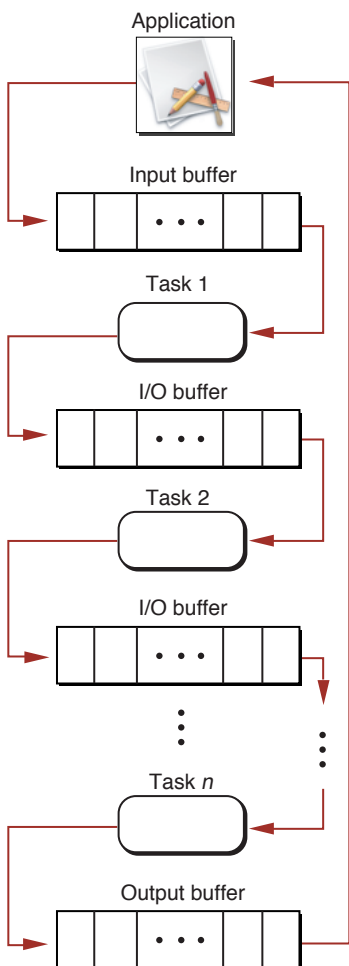
## Sequential Tasks

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 1-5, the sequential task architecture requires a single input buffer and a single output buffer for the pipeline, and intermediate buffers between sequential tasks.

**Figure 1-5** Sequential Tasks





# Using Multiprocessing Services

---

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.

**Note:** This document describes version 2.1 of Multiprocessing Services. For a list of functions changed or added between versions 1.4 and 2.1, see [“Preemptive Task–Safe Mac OS System Software Functions”](#) (page 35). Applications built using older versions of Multiprocessing Services can execute without modification under version 2.1.

**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 31).

## Multiprocessing Services in Mac OS 9 and Mac OS X

Multiprocessing Services 2.1 runs on system software Mac OS 9 and later. All PowerPC Macintosh computers are supported.

Multiprocessing Services 2.1 is packaged as part of the Mac OS System file, so you cannot install version 2.1 on older versions of system software.

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&9 and Mac OS X systems.

## Compatibility with Older System Software

Multiprocessing Services 2.0 was introduced with Mac OS 8.6. Unlike earlier versions of Multiprocessing Services, you can create and execute preemptive tasks with virtual memory turned on. Multiprocessing Services 2.0 runs on all Power Macintosh computers except for 6100/7100/8100 and 5200/6200 series computers.

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.

For a listing of functions introduced with versions 1.0, 2.0, and 2.1 of Multiprocessing Services, see [“Preemptive Task–Safe Mac OS System Software Functions”](#) (page 35).

## 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 processing, such as calculation-intensive work or I/O operations.
- 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 ["Preemptive Task-Safe Mac OS System Software Functions"](#) (page 35).
- If your task needs to allocate memory in Mac OS 9 and earlier, you must allocate the memory prior to signaling the task, or use the function `MPAllocateAligned`.
- 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 31) for more information.
- You can call only preemptive task-safe Mac OS system software functions directly from a task. See Multiprocessing Gestalt Constants and ["Preemptive Task-Safe Mac OS System Software Functions"](#) (page 35) to determine which functions are preemptive task-safe. Other (unsafe) system software functions must be called indirectly through remote procedure calls. See ["Making Remote Procedure Calls"](#) (page 31) 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 non-reentrant code could easily corrupt data or cause a system crash.
- Avoid global variables. The main cause of non-reentrancy 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 29) 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 2-1](#) (page 21) in ["Creating Tasks"](#) (page 21) 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` returns the number of physical processors available on the host computer. The function `MPProcessorsScheduled` 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 pointer-sized parameter and returns a result of type `OSStatus` when it finishes. The 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`. The code in Listing 2-1 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 2-1 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`.

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` 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` 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`. 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`.)

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 `MPSetTaskWeigh`. 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 2-2 calls one of two different functions depending on the request is placed on its queue.

**Listing 2-2** 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`, 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` 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 30).



## Terminating Tasks

In general, you should avoid terminating a task directly. Instead, you should let the task exit normally, either because it has finished its assigned work, or because you posted a quit notification. Doing so allows the task to dispose of any resources or structures it may have allocated. In addition, in Mac OS X, MP tasks may use additional system resources that do not expect to have the task abruptly terminated.

If you must terminate a task, you should call the function `MPTerminateTask`, ideally when the task is blocked waiting on an MP synchronization construct (queue, event, semaphore, or critical region). 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`. See the discussion of `MPTerminateTask` in *Multiprocessing Services Reference* in Carbon Process Management documentation for additional considerations.

Listing 2-3 shows how you might terminate the tasks created in [Listing 2-1](#) (page 21).

### Listing 2-3 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` 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` from a cooperative task, you should specify only the `kDurationImmediate` wait time. You must use a `while` loop that continuously calls `MPWaitOnQueue` 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”.

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 9) 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 simple notification mechanisms (semaphores, event groups, and message queues), one complex one (kernel notifications), and critical regions.

Of the three simple 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 2-4 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 2-4 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` to place the pointer to the task's portion of the data on the task's request queue. It then calls `MPWaitOnQueue` to wait for confirmation that the task has completed.

**Note:** A message queue message is passed to the queue as three pointer-sized parameters. Because the message in Listing 2-4 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 2-4](#) (page 26):

- `MPCreateSemaphore`, `MPSignalSemaphore`, `MPWaitOnSemaphore`, and `MPDeleteSemaphore` for semaphores
- `MPCreateEvent`, `MPSetEvent`, `MPWaitForEvent`, and `MPDeleteEvent` 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.

To use kernel notifications, you should call the following functions:

- `MPCreateNotification`, `MPCauseNotification`, `MPModifyNotification`, and `MPDeleteNotification`

There is no function for waiting on a kernel notification, as the task would wait on the appropriate subcomponents of the kernel notification (for example, a semaphore and a message queue).

Note that the example in [Listing 2-4](#) (page 26) 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`, `MPWaitOnSemaphore`, `MPWaitForEvent`, or `MPEnterCriticalRegion`, 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 2-5 shows a task that uses a semaphore to do periodic actions.

**Listing 2-5** Using a semaphore to perform periodic actions

```
void MyTask(void) {
    MPSemaphoreID delay;

    MPMCreateSemaphore(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 2-6.

**Listing 2-6** Performing actions periodically and on demand

```
main(void) {
    MPSemaphoreID delay;
    ...
    MPMCreateSemaphore(2, 0, &delay);
    MPMCreateTask(...);

    while(true) {
        // Event loop.

        if ( /* something important happened */ )
        {
            MPSignalSemaphore(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`, `MPSetEvent`, or `MPNotifyQueue`. The `MPSignalSemaphore` and `MPSetEvent` functions are always interrupt-safe, while the `MPNotifyQueue` function becomes interrupt-safe if you reserve notifications on the message queue. See the `MPSetQueueReserve` function description for more information about reserving notifications.

 **Warning:** Aside from these three notification functions, only `MPCurrentTaskID`, `MPBlockClear`, `MPBlockCopy`, `MPDataToCode`, `MPTaskIsPreemptive`, and `MPYield` 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 non-reentrant, (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`. 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` 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`. 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

In Mac OS X, if you need to allocate memory for a task, you can use `malloc` or the usual Core Foundation allocator functions, and free it using `free` or `CFRelease` respectively.

However, in Mac OS 9 and earlier, you must call the function `MPAllocateAligned`. Doing so returns a pointer to allocated memory with the alignment you specify. Prior to Mac OS X, 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`. 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 pointer-sized 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`. 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` and `MPGetTaskStorageValue` to set and retrieve the storage data. After you are finished using the storage locations, you must call `MPDeallocateTaskStorageIndex` 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`.

**Note:** Note that in some cases you may want to use notification mechanisms to accomplish periodic actions, as described in “[Handling Periodic Actions](#)” (page 28).

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` and arm it by calling the function `MPArmTimer`. To specify the notification mechanisms to signal when the timer expires, you call the function `MPSetTimerNotify`. 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`. Multiprocessing Services also provides the functions `DurationToAbsolute` and `AbsoluteToDuration` 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:** A subset of system software calls, as described in “[Preemptive Task–Safe Mac OS System Software Functions](#)” (page 35), are preemptive task–safe in Mac OS 9. However, with the exception of the functions in Multiprocessing Services, you should assume that all other system calls are unsafe. 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 the Mac OS 8 and Mac OS 9 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.

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` 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`. 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`. 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` or `MPExtractTaskState`.
- Call `MPDisposeTaskException`, 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 `MPTrowException` function.



# Preemptive Task–Safe Mac OS System Software Functions

This appendix lists Mac OS 9 system software calls that you can call directly from a Multiprocessing Services task without using a remote procedure call. Other system software functions are not preemptive task–safe, so you must call them through a callback as described in “[Making Remote Procedure Calls](#)” (page 31).

These preemptive task–safe functions cause the calling task to block until the call is completed. However, unlike a remote procedure call, the called function does not have to wait until the calling process receives processor time; it executes asynchronously as if it was an interrupt call.

**Important:** Before calling any system software function from a task, you should call `Gestalt` with the appropriate selector to determine whether it is safe to do so.

This appendix contains the following tables of functions::

- Device Manager functions: Table A-1
- File Manager:
  - Table A-2, “Preemptive task–safe HFS Plus parameter block functions”
  - Table A-3, “Preemptive task–safe HFS Plus file system reference functions”
  - Table A-4, “Preemptive task–safe HFS parameter block functions”
  - Table A-5, “Preemptive task–safe high-level HFS functions”
  - Table A-6, “Preemptive task–safe file system specification functions”
- Device Manager and File Manager shared functions: Table A-7
- Miscellaneous functions: Table A-8

**Table A-1** Preemptive task–safe Device Manager functions

Name	Comments
PBOpenSync	Not supported in Carbon, so you should not use this function for File Manager open requests. For file open requests, you should call <code>PBHOpenDFSync</code> , <code>PBHOpenDF</code> , <code>PBOpenForkSync</code> , or <code>FSOpenFork</code> instead. May be handled by the File Manager in Mac OS 9.
PBControlSync	
PBStatusSync	
PBKillIOSync	

**Table A-2** Preemptive task-safe HFS Plus parameter block functions

Name	Comments
PBMakeFSRefSync	
PBMakeFSRefUnicodeSync	
PBCompareFSRefsSync	
PBCreateFileUnicodeSync	
PBCreateDirectoryUnicodeSync	
PBDeleteObjectSync	
PBMoveObjectSync	
PBExchangeObjectsSync	
PBRenameUnicodeSync	
PBGetCatalogInfoSync	
PBSetCatalogInfoSync	
PBOpenIteratorSync	
PBCloseIteratorSync	
PBGetCatalogInfoBulkSync	
PBCatalogSearchSync	
PBCreateForkSync	
PBDeleteForkSync	
PBIterateForksSync	
PBOpenForkSync	
PBReadForkSync	
PBWriteForkSync	
PBGetForkPositionSync	
PBSetForkPositionSync	
PBGetForkSizeSync	
PBSetForkSizeSync	
PBAllocateForkSync	

## Preemptive Task-Safe Mac OS System Software Functions

Name	Comments
PBFlushForkSync	
PBCloseForkSync	
PBGetForkCBInfoSync	
PBGetVolumeInfoSync	
PBSetVolumeInfoSync	

**Table A-3** Preemptive task-safe HFS Plus file system reference functions

Name	Comments
FSpMakeFSRef	
FMakeFSRefUnicode	
FSCompareFSRefs	
FSCreateFileUnicode	
FSCreateDirectoryUnicode	
FSDeleteObject	
FSMoveObject	
FSExchangeObjects	
FSRenameUnicode	
FSGetCatalogInfo	
FSSetCatalogInfo	
FSOpenIterator	
FSCloseIterator	
FSGetCatalogInfoBulk	
FSCatalogSearch	
FSCreateFork	
FSDeleteFork	
FSIterateForks	
FSOpenFork	
FSReadFork	

## Preemptive Task-Safe Mac OS System Software Functions

Name	Comments
FSWriteFork	
FSGetForkPosition	
FSSetForkPosition	
FSGetForkSize	
FSSetForkSize	
FSAllocateFork	
FSFlushFork	
FSCloseFork	
FSGetForkCBInfo	
FSGetVolumeInfo	
FSSetVolumeInfo	
FSGetDataForkName	
FSGetResourceForkName	

**Table A-4** Preemptive task-safe HFS parameter block functions

Name	Comments
PBXGetVolInfoSync	
PBFlushVolSync	
PBAllocateSync	
PBGetEOFSync	
PBSetEOFSync	
PBGetFPosSync	
PBSetFPosSync	
PBFlushFileSync	
PBCatSearchSync	
PBHSetVolSync	
PBHGetVolSync	
PBCatMoveSync	

## Preemptive Task-Safe Mac OS System Software Functions

Name	Comments
PBDirCreateSync	
PBGetFCBInfoSync	
PBGetCatInfoSync	
PBSetCatInfoSync	
PBAllocContigSync	
PBLockRangeSync	
PBUnlockRangeSync	
PBSetVInfoSync	
PBHGetVInfoSync	
PBHOpenSync	
PBHOpenRFSync	
PBHOpenDFSync	
PBHCreateSync	
PBHDeleteSync	
PBHRenameSync	
PBHRstFLockSync	
PBHSetFLockSync	
PBHGetFInfoSync	
PBHSetFInfoSync	
PBMakeFSSpecSync	
PBHGetVolParmsSync	
PBHGetLogInInfoSync	
PBHGetDirAccessSync	
PBHSetDirAccessSync	
PBHMapIDSync	
PBHMapNameSync	
PBHCopyFileSync	

## Preemptive Task-Safe Mac OS System Software Functions

Name	Comments
PBHMoveRenameSync	
PBHOpenDenySync	
PBHOpenRFDenySync	
PBGetXCatInfoSync	
PBExchangeFilesSync	
PBCreateFileIDRefSync	
PBResolveFileIDRefSync	
PBDeleteFileIDRefSync	
PBGetForeignPrivsSync	
PBSetForeignPrivsSync	
PBDTAddIconSync	
PBDTGetIconSync	
PBDTGetIconInfoSync	
PBDTAddAPPLSync	
PBDTRemoveAPPLSync	
PBDTGetAPPLSync	
PBDTSetCommentSync	
PBDTRemoveCommentSync	
PBDTGetCommentSync	
PBDTFlushSync	
PBDTResetSync	
PBDTGetInfoSync	
PBDTDeleteSync	
PBShareSync	
PBUnshareSync	
PBGetUGEntrySync	



**Table A-5** Preemptive task-safe high-level HFS functions

Name	Comments
FSClose	
FSRead	
FSWrite	
FlushVol	
Allocate	
GetEOF	
SetEOF	
GetFPos	
SetFPos	
HGetVol	
HSetVol	
HOpen	
HOpenDF	
HOpenRF	
AllocContig	
HCreate	
DirCreate	
HDelete	
HGetFInfo	
HSetFInfo	
HSetFLock	
HRstFLock	
HRename	
CatMove	

**Table A-6** Preemptive task-safe file system specification functions

Name	Comments
FMakeFSSpec	
FSpOpenDF	
FSpOpenRF	
FSpCreate	
FSpDirCreate	
FSpDelete	
FSpGetFInfo	
FSpSetFInfo	
FSpSetFLock	
FSpRstFLock	
FSpRename	
FSpCatMove	
FSpExchangeFiles	

**Table A-7** Preemptive task-safe functions shared by the Device Manager and File Manager

Name	Comments
PBCloseSync	
PBReadSync	
PBWriteSync	

**Table A-8** Miscellaneous preemptive task-safe functions

Name	Comments
DTInstall	Deferred Task Manager function
WakeUpProcess	Process Manager function

# 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 B-1 to do so.

## Listing B-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:
```

## Calculating the Intertask Signaling Time

```

        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;

```

## Calculating the Intertask Signaling Time

```

    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();

```

## Calculating the Intertask Signaling Time

```

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 signaling 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.

```

## Calculating the Intertask Signaling Time

```

printf(" Intertask signaling using semaphores 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 signaling using events overhead: %.3f usec \n", uSec);

gets(buff);
}

```





# Changes From Previous Versions of Multiprocessing Services

---

Multiprocessing Services 2.1 supports all the functions available with version 2.0. For compatibility between version 2.0 and older versions, see Table C-3, Table C-4, and Table C-5.

Table C-1 lists Multiprocessing Services functions introduced with version 2.1:

**Table C-1** New functions introduced with version 2.1

Name	Comments
MPGetNextCpuID	
MPGetNextTaskID	
MPCreateNotification	For manipulating kernel notifications. See <a href="#">“Kernel Notifications”</a> (page 13) for more information about this notification mechanism.
MPDeleteNotification	
MPSModifyNotification	
MPCauseNotification	

Table C-2 lists Multiprocessing Services functions that were introduced in version 2.0.

**Table C-2** Functions introduced with version 2.0

Name	Comments
MPProcessorsScheduled	
MPsetTaskWeight	
MPTaskIsPreemptive	
MPAllocateTaskStorageIndex	
MPDeallocateTaskStorageIndex	
MPSetTaskStorageValue	
MPGetTaskStorageValue	
MPSetQueueReserve	
MPCreateEvent	
MPDeleteEvent	

## Changes From Previous Versions of Multiprocessing Services

Name	Comments
MPSetEvent	
MPWaitForEvent	
UpTime	
DurationToAbsolute	
AbsoluteToDuration	
MPDelayUntil	
MPCreateTimer	
MPDeleteTimer	
MPSetTimerNotify	
MPArmTimer	
MPCancelTimer	
MPSetExceptionHandler	
MPTThrowException	
MPDisposeTaskException	
MPExtractTaskState	
MPSetTaskState	
MPRegisterDebugger	
MPRegisterDebugger	
MPAllocateAligned	Preferred over MPAllocate.
<b>MPGetAllocatedBlockSize</b>	
MPBlockClear	
MPDataToCode	
MPRemoteCall	Preferred over _MPRPC

Table C-3 lists the functions that were introduced in version 1.0 that are still supported in version 2.0.

**Table C-3** Older functions supported in version 2.0

Name	Comments
MPProcessors	

## Changes From Previous Versions of Multiprocessing Services

Name	Comments
MPCreateTask	
MPTerminateTask	
MPCurrentTaskID	
MPYield	
MPExit	
MPCreateQueue	
MPDeleteQueue	
MPNotifyQueue	
MPWaitOnQueue	
MPCreateSemaphore	
MPCreateBinarySemaphore	In C, a macro that calls MPCreateSemaphore.
MPDeleteSemaphore	
MPSignalSemaphore	
MPWaitOnSemaphore	
MPCreateCriticalRegion	
MPDeleteCriticalRegion	
MPEnterCriticalRegion	
MPExitCriticalRegion	
MPAllocate	Deprecated. Use MPAllocateAligned instead.
MPFree	
MPBlockCopy	
MPLibraryIsLoaded	In C, a macro that checks to see if the MPProcessors symbol is resolved.

Table C-4 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 C-4** Unofficial functions still supported in version 2.0

Name	Comments
_MPRPC	Deprecated. Use MPRemoteCall instead.

## Changes From Previous Versions of Multiprocessing Services

Name	Comments
<code>_MPAllocateSys</code>	Deprecated. Use <code>MPAllocateAligned</code> instead.
<code>_MPTaskIsToolboxSafe</code>	
<code>_MPLibraryVersion</code>	
<code>_MPLibraryIsCompatible</code>	

Table C-5 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 C-5** Debugging functions unsupported in version 2.0

Name	Comments
<code>_MPInitializePrintf</code>	
<code>_MPPrintf</code>	
<code>_MPDebugStr</code>	
<code>_MPStatusPString</code>	
<code>_MPStatusCString</code>	

# Document Revision History

This table describes the changes to *Multiprocessing Services Programming Guide*.

Date	Notes
2007-10-31	Made minor editorial and technical corrections.
2005-07-07	Changed title from "Adding Multitasking Capability to Applications Using Multiprocessing Services." Indicated that <code>MPAllocateAligned</code> is not needed in Mac OS X. Emphasized avoiding calling <code>MPTerminateTask</code> if possible. Added link to "Multithreading" programming topic.
	Fixed bugs that caused broken links in HTML.
1999-10-15	Revised for Multiprocessing Services 2.1. The following changes were made from the 2.0 version:
	Added "Gestalt Constants", which you can use to determine the availability of preemptively safe Mac OS system software functions, and added a list of these functions in " <a href="#">Preemptive Task-Safe Mac OS System Software Functions</a> " (page 35). Changed text in " <a href="#">Criteria for Creating Tasks</a> " (page 20) and " <a href="#">Making Remote Procedure Calls</a> " (page 31) to reflect these additions.
	Added new functions: <code>MPGetNextCpuID</code> and <code>MPGetNextTaskID</code> .
	Added new kernel notification functions: <code>MPCreateNotification</code> , <code>MPCreateNotification</code> , <code>MPModifyNotification</code> , and <code>MPCauseNotification</code> . Added " <a href="#">Kernel Notifications</a> " (page 13) section.
	Added new options for <code>MPCreateTask</code> . See "Task Creation Options" for details.
	Added new task information structure, <code>MPTaskInfo</code> , for use with <code>MPExtractTaskState</code> .
	Added new task run state constants to be used with the new <code>MPTaskInfo</code> structure. See "Task Run State Constants" for details.
	Added glossary entries for coherence groups and kernel notifications.
1999-04-30	Initial public release. The following changes were made from the previous (seed draft) version: Added " <a href="#">Introduction to Multiprocessing Services Programming Guide</a> " (page 7) " <a href="#">About Multitasking on the Mac OS</a> " (page 9) and " <a href="#">Using Multiprocessing Services</a> " (page 19) which include introductory information, conceptual information, programming discussions, and sample code.
	Added versioning information to functions, data types, and constants in "Multiprocessing Services Reference."

Date	Notes
	Added discussion and parameter information to <code>MPTaskIsPreemptive</code> indicating how and why you can specify <code>kInvalidID</code> to determine the preemptiveness of the current task. Also added Version Notes section.
	Correction in <code>MPWaitOnQueue</code> , <code>MPWaitOnSemaphore</code> , <code>MPWaitForEvent</code> , and <code>MPEnterCriticalRegion</code> : When calling from a cooperative task, you should specify only <code>kDurationImmediate</code> waits; others are allowable, but they will cause the task to block.
	Added information stating that setting event bits in <code>MPSetEvent</code> and obtaining and clearing and event group in <code>MPWaitForEvent</code> 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 <code>MPDelayUntil</code> to clarify that you must indicate a specific time to unblock the task, not a duration.
	Changed wording for <code>MPAllocateTaskStorageIndex</code> and <code>MPDeallocateTaskStorageIndex</code> to indicate that these functions do not actually allocate or deallocate memory.
	Added disclaimer to <code>MPTThrowException</code> 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 <code>MPSetExceptionHandler</code> to indicate the format of the message sent to the exception handler. Discussion of informative messages in <code>MPRegisterDebugger</code> removed to reflect status as of version 2.0.
	Added information to <code>MPExtractTaskState</code> and <code>MPSetTaskState</code> indicating that attempting to set or read state information for a non-suspended task returns the error <code>kMPIInsufficientResourcesErr</code> . Added information to <code>MPSetTaskState</code> and "Task State Constants": 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 <code>MPRemoteCall</code> modified 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 <code>WaitNextEvent</code> ). Atomic memory operations mentioned in <code>MPRemoteCall</code> and "Making Remote Procedure Calls" (page 31) are now located in <code>InterfaceLib</code> , not the Driver Services Library.
	Data types <code>MPAddressSpaceID</code> and <code>MPCpuID</code> removed to reflect status as of version 2.0.

## REVISION HISTORY

### Document Revision History

Date	Notes
	Clarified that you can use the constants in "Timer Duration Constants" to specify any number of waiting times by adding multipliers.
	Correction in "Memory Allocation Alignment Constants": CPU interlock instruction <code>swarx</code> should be <code>stwcx</code> .
	Specified the <code>MachineExceptions.h</code> structures that correspond to the state information constants in "Task State Constants". Removed non-bitmask values ( <code>kMPTaskPropagate</code> , <code>kMPTaskResumeStep</code> , and <code>kMPTaskResumeBranch</code> ) and descriptions from "Task Exception Disposal Constants".
1999-02-26	First seed draft release

**REVISION HISTORY**

Document Revision History



# Index

---

## Symbols

---

"bank line" tasking architecture [16](#)  
"divide and conquer" tasking architecture [15](#)  
"pipeline" tasking architecture [17](#)

## Numerals

---

68K code in preemptive tasks [19](#)

## A

---

absolute time, defined [31](#)  
address spaces [11](#), [32](#)  
allocating memory in tasks [30](#)

## B

---

binary semaphores [12](#)  
blocked task, defined [11](#)

## C

---

calling 68K code [31](#)  
calling nonreentrant functions [31](#)  
changes from previous versions of Multiprocessing Services [49](#)  
checking for the availability of Multiprocessing Services [20](#)  
critical regions  
  creating [29](#)  
  defined [14](#)

## D

---

debuggers [32](#)  
determining the number of processors [21](#)

## E

---

event groups, defined [13](#)  
exception handlers [32](#)

## H

---

hardware requirements [19](#)

## I

---

interrupt handlers [29](#)  
intertask signaling time, calculating [43](#)

## K

---

kernel notifications, defined [13](#)

## M

---

Mac OS task, defined [11](#)  
message queues  
  defined [13](#)  
  used when creating tasks [23](#)  
microprocessors *See* processors  
Multiprocessing Services on Mac OS 8 versus Mac OS X [19](#)  
multiprocessing, defined [10](#)  
multitasking

cooperative versus preemptive 9  
 defined 9

## N

---

new functions added with version 2.0 49  
 notification methods 12  
 notification queue 23

## P

---

preemptive tasks *See* tasks  
 preemptive task–safe system software functions 20, 31, 35  
 processors  
   determining the number of 21  
   versus number of tasks 14

## Q

---

queues *See* message queues

## R

---

recursion in critical regions 14  
 remote procedure calls 31

## S

---

schedulers 11  
 semaphores  
   defined 12  
   used for periodic actions 28  
 shared resources 12  
 synchronization of tasks 12, 26  
 system requirements 19  
 system software, preemptive task–safe functions in 20, 31, 35

## T

---

task schedulers 11  
 task weight 23  
 task-specific storage 30

tasking architectures  
   multiple independent tasks 15  
   parallel tasks with a single set of I/O buffers 16  
   parallel tasks with parallel I/O buffers 15  
   sequential tasks 17

### tasks

  blocked 11  
   compared to threads 10  
   creating 21  
   defined 9  
   intertask signaling time 43  
   notifying at interrupt time 29  
   synchronization of 12, 26  
   terminating 25  
   versus number of processors 14

### tasks

  setting weight of 23  
 terminating tasks 25  
 termination queue 23  
 Thread Manager, compared to Multiprocessing Services 10  
 timers 30

## V

---

version 2.0, new functions added for 49  
 virtual memory 19

## W

---

waiting on a queue 24, 27