Multiprocessing Services Reference

Carbon > Process Management

2008-02-08

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

Simultaneously published in the United States and Canada.

Contents

Multiprocessing Services Reference

Framework:	CoreServices/CoreServices.h
Companion guide	Multiprocessing Services Programming Guide
Declared in	Multiprocessing.h MultiprocessingInfo.h

Overview

Multiprocessing Services is an API that lets you create preemptive tasks in your application that can run on one or more microprocessors. Unlike the cooperative threads created by the Thread Manager, Multiprocessing Services automatically divides processor time among the available tasks, so that no particular task can monopolize the system. This document is relevant to you if you want to add multitasking capability to your Mac OS applications.

In Mac OS X, Carbon supports Multiprocessing Services with the following restrictions:

- Debugging functions are not implemented. Use the mach APIs provided by the system to implement debugging services.
- Opaque notification IDs are local to your process; they are not globally addressable across processes.
- Global memory allocation is not supported.

Functions by Task

Determining Multiprocessing Services And Processor Availability

`_MPIsFullyInitialized` (page 48)
Indicates whether Multiprocessing Services is available for use.

`MPGetNextCpuID` (page 29)
Obtains the next CPU ID in the list of physical processors of the specified memory coherence group.

`MPProcessors` (page 33)
Returns the number of processors on the host computer.

`MPProcessorsScheduled` (page 33)
Returns the number of active processors available on the host computer.

Creating and Handling Message Queues

`MPCreateQueue` (page 18)
Creates a message queue.

`MPDeleteQueue` (page 24)
Deletes a message queue.

`MPNotifyQueue` (page 32)
Sends a message to the specified message queue.

`MPSetQueueReserve` (page 37)
Reserves space for messages on a specified message queue.

`MPWaitOnQueue` (page 46)
Obtains a message from a specified message queue.

Creating and Handling Semaphores

`MPCreateSemaphore` (page 18)
Creates a semaphore.

`MPDeleteSemaphore` (page 25)
Removes a semaphore.

`MPSignalSemaphore` (page 42)
Signals a semaphore.

`MPWaitOnSemaphore` (page 47)
Waits on a semaphore

Creating and Scheduling Tasks

`MPCreateTask` (page 19)
Creates a preemptive task.

`MPCurrentTaskID` (page 21)
Obtains the task ID of the currently-executing preemptive task

`MPSetTaskType` (page 39)
Sets the type of the task.

`MPExit` (page 27)
Allows a task to terminate itself

`MPGetNextTaskID` (page 30)
Obtains the next task ID in the list of available tasks.

`MPSetTaskWeight` (page 40)
Assigns a relative weight to a task, indicating how much processor time it should receive compared to other available tasks.

`MPTaskIsPreemptive` (page 42)
Determines whether a task is preemptively scheduled.

`MPTerminateTask` (page 43)
Terminates an existing task.

`MPYield` (page 47)
Allows a task to yield the processor to another task.

Handling Critical Regions

`MPCreateCriticalRegion` (page 16)
Creates a critical region object.

`MPDeleteCriticalRegion` (page 23)
Removes the specified critical region object.

`MPEnterCriticalRegion` (page 26)
Attempts to enter a critical region.

`MPExitCriticalRegion` (page 27)
Exits a critical region.

Handling Event Groups

`MPCreateEvent` (page 17)
Creates an event group.

`MPDeleteEvent` (page 23)
Removes an event group.

`MPSetEvent` (page 36)
Merges event flags into a specified event group.

`MPWaitForEvent` (page 45)
Retrieves event flags from a specified event group.

Handling Kernel Notifications

`MPCauseNotification` (page 16)
Signals a kernel notification.

`MPCreateNotification` (page 17)
Creates a kernel notification

`MPDeleteNotification` (page 24)
Removes a kernel notification.

`MPModifyNotification` (page 31)
Adds a simple notification to a kernel notification.

`MPModifyNotificationParameters` (page 32)

Accessing Per-Task Storage Variables

`MPAllocateTaskStorageIndex` (page 12)
Returns an index number to access per-task storage.

`MPDeallocateTaskStorageIndex` (page 22)
Frees an index number used to access per-task storage

`MPGetTaskStorageValue` (page 30)
Gets the storage value stored at a specified index number.

`MPSetTaskStorageValue` (page 39)
Sets the storage value for a given index number.

Memory Allocation Functions

`MPAllocate` (page 11)
Allocates a nonrelocatable memory block. (Deprecated. Use `MPAllocateAligned` instead.)

`MPAllocateAligned` (page 12)
Allocates a nonrelocatable memory block.

`MPBlockClear` (page 14)
Clears a block of memory.

`MPBlockCopy` (page 14)
Copies a block of memory.

`MPDataToCode` (page 21)
Designates the specified block of memory as executable code.

`MPFree` (page 28)
Frees memory allocated by `MPAllocateAligned`.

`MPGetAllocatedBlockSize` (page 29)
Returns the size of a memory block.

Remote Calling Functions

`MPRemoteCall` (page 34)
Calls a non-reentrant function and blocks the current task.

`MPRemoteCallCFM` (page 35)
Calls a non-reentrant function and blocks the current task.

Timer Services Functions

`MPArmTimer` (page 13)
Arms the timer to expire at a given time.

`MPCancelTimer` (page 15)
Cancels an armed timer.

`MPCreateTimer` (page 20)
Creates a timer.

`MPDelayUntil` (page 22)
Blocks the calling task until a specified time.

`MPDeleteTimer` (page 25)
Removes a timer.

`MPSetTimerNotify` (page 40)
Sets the notification information associated with a timer.

Exception Handling Functions

`MPDisposeTaskException` (page 26)
Removes a task exception.

`MPExtractTaskState` (page 28)
Extracts state information from a suspended task.

`MPSetExceptionHandler` (page 36)
Sets an exception handler for a task.

`MPSetTaskState` (page 38)
Sets state information for a suspended task.

`MPThrowException` (page 44)
Throws an exception to a specified task.

Debugger Support Functions

`MPRegisterDebugger` (page 34)
Registers a debugger.

`MPUnregisterDebugger` (page 44)
Unregisters a debugger.

Functions

MPAllocate

Allocates a nonrelocatable memory block. (**Deprecated.** Use `MPAllocateAligned` instead.)

```
LogicalAddress MPAllocate (
    ByteCount size
);
```

Parameters

size
The size, in bytes, of the memory block to allocate.

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

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPAllocateAligned

Allocates a nonrelocatable memory block.

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

Parameters

size

The size, in bytes, of the memory block to allocate.

alignment

The desired alignment of the allocated memory block. See "Memory Allocation Alignment Constants" (page 63) for a list of possible values to pass. Note that there will be a minimum alignment regardless of the requested alignment. If the requested memory block is 4 bytes or smaller, the block will be at least 4-byte aligned. If the requested block is greater than 4 bytes, the block will be at least 8-byte aligned.

options

Any optional information to use with this call. See "Memory Allocation Option Constants" (page 65) for a list of possible values to pass.

Return Value

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

Discussion

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

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

Also see the function `MPFree` (page 28).

Special Considerations

Mac OS X does not support allocation of global (cross address space) or resident memory with this function. In addition, passing the `kMPAllocateNoGrowthMask` constant in the `options` parameter has no effect in Mac OS X, since memory allocation is done with sparse heaps.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`Multiprocessing.h`

MPAllocateTaskStorageIndex

Returns an index number to access per-task storage.

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

Parameters

index

On return, `index` contains an index number you can use to store task data.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73).

Discussion
A call to the function `MPAllocateTaskStorageIndex` returns an index number that is common across all tasks in the current process. You can use this index number in calls to `MPSetTaskStorageValue` (page 39) and `MPGetTaskStorageValue` (page 30) to set a different value for each task using the same index.

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

Also see the function `MPDeallocateTaskStorageIndex` (page 22).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPArmTimer

Arms the timer to expire at a given time.

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

Parameters

timerID

The ID of the timer you want to arm.

expirationTime

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

options

Any optional actions. See "Timer Option Masks" (page 72) for a list of possible values.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73). If the timer has already expired, the reset does not take place and the function returns `kMPInsufficientResourcesErr`.

Discussion
The expiration time is an absolute time, which you can generate by calling the Driver Services Library function `UpTime`. When the timer expires, a notification is sent to the notification mechanism specified in the last `MPSetTimerNotify` (page 40) call. If the specified notification ID has become invalid, no action is taken when the timer expires. The timer itself is deleted when it expires unless you specified the `kMPPreserveTimerID` option in the options parameter.

Also see the function `MPCancelTimer` (page 15).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPBlockClear

Clears a block of memory.

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

Parameters

address
The starting address of the memory block you want to clear.

size
The number of bytes you want to clear.

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

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

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPBlockCopy

Copies a block of memory.

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

Parameters

source

The starting address of the memory block you want to copy.

destination

The location to which you want to copy the memory block.

size

The number of bytes to copy.

Discussion

This function simply calls through to the Driver Services Library function `BlockMoveData`. Note that you should not make any assumptions about the state of the destination memory while this function is executing. In the intermediate state, values may be present that are neither the original nor the final ones. For example, this function may use the `'dcbz'` instruction. If the underlying memory is not cacheable, if the memory is write-through instead of copy-back, or if the cache block is flushed for some reason, the `'dcbz'` instruction will write zeros to the destination. You can avoid the use of the `'dcbz'` instruction by calling `BlockMoveDataUncached`, but even that function makes no other guarantees about the memory block's intermediate state.

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

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

Availability

Available in Mac OS X v10.0 and later.

Declared In

`Multiprocessing.h`

MPCancelTimer

Cancels an armed timer.

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

Parameters

timerID

The ID of the armed timer you want to cancel.

timeRemaining

On return, the `timeRemaining` contains the time remaining before the timer would have expired.

Return Value

A result code. See "Multiprocessing Services Result Codes" (page 73). If the timer has already expired, this function returns `kMPInsufficientResourcesErr`.

Discussion
Also see the function `MPArmTimer` (page 13).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPCauseNotification

Signals a kernel notification.

```
OSStatus MPCauseNotification (
   MPNotificationID notificationID
);
```

Parameters

notificationID
The ID of the kernel notification you want to signal.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73).

Discussion
You call this function to signal a kernel notification much as you would signal any simple notification (for example, `MPNotifyQueue` (page 32)).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPCreateCriticalRegion

Creates a critical region object.

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

Parameters

criticalRegion
On return, the `criticalRegion` contains the ID of the newly created critical region object.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73).

Discussion
Also see the function `MPDeleteCriticalRegion` (page 23).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPCreateEvent

Creates an event group.

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

Parameters

event

On return, `event` contains the ID of the newly created event group.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73).

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

Also see the function `MPDeleteEvent` (page 23).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPCreateNotification

Creates a kernel notification

```
OSStatus MPCreateNotification (
   MPNotificationID *notificationID
);
```

Parameters

notificationID

On return, `notificationID` points to the newly created kernel notification.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73).

Discussion
After creating the kernel notification object, you can add simple notifications by calling the function `MPModifyNotification` (page 31).

Also see the function `MPDeleteNotification` (page 24).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPCreateQueue

Creates a message queue.

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

Parameters

queue
On return, the variable contains the ID of the newly created message queue.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73). If a queue could not be created, `MPCreateQueue` returns `kMPInsufficientResourcesErr`.

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

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

See also the functions `MPDeleteQueue` (page 24) and `MPSetQueueReserve` (page 37).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPCreateSemaphore

Creates a semaphore.

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

Parameters

maximumValue
The maximum allowed value of the semaphore.

initialValue
The initial value of the semaphore.

semaphore
On return, `semaphore` contains the ID of the newly–created semaphore.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73).

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

Also see the function `MPDeleteSemaphore` (page 25).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPCreateTask

Creates a preemptive task.

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

Parameters

entryPoint

A pointer to the task function. The task function should take a single pointer-sized parameter and return a value of type `OSStatus`.

parameter

The parameter to pass to the task function.

stackSize

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

Note that in Mac OS X prior to version 10.1, this parameter is ignored, and all stacks have the default size of 512 KB. Versions 10.1 and later do not have this limitation.

notifyQueue

The ID of the message queue to which the system will send a message when the task terminates. You specify the first two values of the message in the parameters `terminationParameter1` and `terminationParameter2` respectively. The last message value contains the result code of the task function.

terminationParameter1

A pointer-sized value that is sent to the message queue specified by the parameter `notifyQueue` when the task terminates.

terminationParameter2

A pointer-sized value that is sent to the message queue specified by the parameter `notifyQueue` when the task terminates.

options

Optional attributes of the preemptive task. See "Task Creation Options" (page 68) for a list of possible values.

task

On return, `task` points to the ID of the newly created task.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73). If `MPCreateTask` could not create the task because some critical resource was not available, the function returns `kMPInsufficientResourcesErr`. Usually this is due to lack of memory to allocate the internal data structures associated with the task or the stack. The function also returns `kMPInsufficientResourcesErr` if any reserved option bits are set.

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

- By returning from its entry point
- By calling `MPExit` (page 27)
- When specified as the target of an `MPTerminateTask` (page 43) call
- If a hardware-detected exception or programming exception occurs and no exception handler is installed
- If the application calls `ExitToShell`

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

To set the relative processor weight to be assigned to a task, use the function `MPSetTaskWeight` (page 40).

See also the function `MPTerminateTask` (page 43).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPCreateTimer

Creates a timer.

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

Parameters

timerID

On return, the `timerID` contains the ID of the newly created timer.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73).

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

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

To specify the notification mechanism to signal, use the function `MPSetTimerNotify` (page 40).

Also see the functions `MPDeleteTimer` (page 25) and `MPArmTimer` (page 13).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPCurrentTaskID

Obtains the task ID of the currently-executing preemptive task

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

Return Value
The task ID of the current preemptive task. See the description of the `MPTaskID` data type.

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

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

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPDataToCode

Designates the specified block of memory as executable code.

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

Parameters

address
The starting address of the memory block you want to designate as code.

size
The size of the memory block.

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

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

Availability
Available in Mac OS X v10.0 and later.
Not available to 64-bit applications.

Declared In
`Multiprocessing.h`

MPDeallocateTaskStorageIndex

Frees an index number used to access per-task storage

```
OSStatus MPDeallocateTaskStorageIndex (
   TaskStorageIndex taskIndex
);
```

Parameters
index
The index number you want to deallocate.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73).

Discussion
Also see the function `MPAllocateTaskStorageIndex` (page 12).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPDelayUntil

Blocks the calling task until a specified time.

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

Parameters
expirationTime
The time to unblock the task.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73).

Discussion
You cannot call this function from a cooperative task.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPDeleteCriticalRegion

Removes the specified critical region object.

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

Parameters

criticalRegion
The critical region object you want to remove.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73).

Discussion
Calling this function unblocks all tasks waiting to enter the critical region and their respective `MPEnterCriticalRegion` (page 26) calls will return with the result code `kMPDeletedErr`.

Also see the function `MPCreateCriticalRegion` (page 16).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPDeleteEvent

Removes an event group.

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

Parameters

event
The ID of the event group you want to remove.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73).

Discussion
After deletion, the event ID becomes invalid, and all internal resources associated with the event group are reclaimed. Calling this function unblocks all tasks waiting on the event group and their respective `MPWaitForEvent` (page 45) calls will return with the result code `kMPDeletedErr`.

Also see the function `MPCreateEvent` (page 17).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPDeleteNotification

Removes a kernel notification.

```
OSStatus MPDeleteNotification (
   MPNotificationID notificationID
);
```

Parameters

notificationID
The ID of the notification you want to remove.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73).

Discussion
Also see the function `MPCreateNotification` (page 17).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPDeleteQueue

Deletes a message queue.

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

Parameters

queue
The ID of the message queue you want to delete.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73).

Discussion
After calling `MPDeleteQueue`, the specified queue ID becomes invalid, and all internal resources associated with the queue (including queued messages) are reclaimed. Any tasks waiting on the queue are unblocked and their respective `MPWaitOnQueue` (page 46) calls will return with the result code `kMPDeletedErr`.

Also see the function `MPCreateQueue` (page 18).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPDeleteSemaphore

Removes a semaphore.

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

Parameters

semaphore
The ID of the semaphore you want to remove.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73).

Discussion
Calling this function unblocks all tasks waiting on the semaphore and the tasks' respective `MPWaitOnSemaphore` (page 47) calls will return with the result code `kMPDeletedErr`.

Also see the function `MPCreateSemaphore` (page 18).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPDeleteTimer

Removes a timer.

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

Parameters

timerID
The ID of the timer you want to remove.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73).

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

Also see the function `MPCreateTimer` (page 20).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPDisposeTaskException

Removes a task exception.

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

Parameters

task

The task whose exception you want to remove.

action

Any actions to perform on the task. For example, you can enable single-stepping when the task resumes, or you can pass the exception on to another handler. See "Task Exception Disposal Constants" (page 68) for a listing of possible values.

Return Value

A result code. See "Multiprocessing Services Result Codes" (page 73). If the specified action is invalid or unsupported, or if the specified task is not suspended, this function returns `kMPInsufficientResourcesErr`.

Discussion

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

Availability

Available in Mac OS X v10.0 and later.

Declared In

`Multiprocessing.h`

MPEnterCriticalRegion

Attempts to enter a critical region.

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

Parameters

criticalRegion

The ID of the critical region you want to enter.

timeout

The maximum time to wait for entry before timing out. See "Timer Duration Constants" (page 71) for a list of constants you can use to specify the wait interval.

Return Value

A result code. See "Multiprocessing Services Result Codes" (page 73).

Discussion

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

Once a task enters a critical region it can make further calls to `MPEnterCriticalRegion` without blocking (its use count increments for each call). However, each call to `MPEnterCriticalRegion` must be balanced by a call to `MPExitCriticalRegion` (page 27) ; otherwise the region is not released for use by other tasks.

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

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPExit

Allows a task to terminate itself

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

Parameters

status

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

Discussion
When called from within a preemptive task, the task terminates, and the value indicated by the parameter `status` is sent to the termination message queue you specified in `MPCreateTask` (page 19). Note that you cannot call `MPExit` from outside a preemptive task.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPExitCriticalRegion

Exits a critical region.

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

Parameters

criticalRegion

The ID of the critical region you want to exit.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73). If the task does not own the critical region specified by `criticalRegion`, `MPExitCriticalRegion` returns `kMPInsufficientResourcesErr`.

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

Also see the function `MPEnterCriticalRegion` (page 26).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPExtractTaskState

Extracts state information from a suspended task.

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

Parameters

task

The task whose state information you want to obtain.

kind

The kind of state information you want to obtain. See "Task State Constants" (page 70) for a listing of possible values.

info

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

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73). If you attempt to extract state information for a running task, this function returns `kMPInsufficientResourcesErr`.

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

Also see the function `MPSetTaskState` (page 38).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPFree

Frees memory allocated by `MPAllocateAligned`.

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

Parameters

object

A pointer to the memory you want to release.

Discussion
Also see the function `MPAllocateAligned` (page 12).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPGetAllocatedBlockSize

Returns the size of a memory block.

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

Parameters

object

The address of the memory block whose size you want to determine.

Return Value
The size of the allocated memory block, in bytes.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPGetNextCpuID

Obtains the next CPU ID in the list of physical processors of the specified memory coherence group.

```
OSStatus MPGetNextCpuID (
   MPCoherenceID owningCoherenceID,
   MPCpuID *cpuID
);
```

Parameters

owningCoherenceID

The ID of the memory coherence group whose physical processor IDs you want to obtain. Pass `kMPInvalidIDErr`, as only one coherence group, internal RAM, is currently defined.

cpuID

On return, `cpuID` points to the ID of the next physical processor.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73).

Discussion
By iterating on this function (after calling `MPProcessors` (page 33) , for example), you can obtain the IDs of all the processors available on the host computer. Generally, you would only use this function in diagnostic programs.

Availability
Available in Mac OS X v10.4 and later.

Declared In
`MultiprocessingInfo.h`

MPGetNextTaskID

Obtains the next task ID in the list of available tasks.

```
OSStatus MPGetNextTaskID (
   MPProcessID owningProcessID,
   MPTaskID *taskID
);
```

Parameters

owningProcessID

The ID of the process (typically the application) that owns the tasks. This ID is the same as the process ID handled by the Code Fragment Manager.

taskID

On return, `taskID` points to ID of the next task in the list of tasks.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73).

Discussion
By iterating on this function, you can obtain the IDs of all the tasks in a given process. These tasks may be running, ready, or blocked. Generally you would only use this function in diagnostic programs.

Availability
Available in Mac OS X v10.4 and later.

Declared In
`MultiprocessingInfo.h`

MPGetTaskStorageValue

Gets the storage value stored at a specified index number.

```
TaskStorageValue MPGetTaskStorageValue (
   TaskStorageIndex taskIndex
);
```

Parameters

index

The index number of the storage value you want to obtain.

Return Value
The value stored at the specified index number. See the description of the `TaskStorageValue` data type.

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

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

Also see the function `MPSetTaskStorageValue` (page 39).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPModifyNotification

Adds a simple notification to a kernel notification.

```
OSStatus MPModifyNotification (
   MPNotificationID notificationID,
   MPOpaqueID anID,
   void *notifyParam1,
   void *notifyParam2,
   void *notifyParam3
);
```

Parameters

notificationID
The ID of the kernel notification you want to add to..

anID
The ID of the simple notification (semaphore, message group, or event group) you want to add to the kernel notification.

notifyParam1
If `anID` specifies an event group, this parameter should contain the flags to set in the event group when `MPCauseNotification` (page 16) is called. If `anID` specifies a message queue, this parameter should contain the first pointer-sized value of the message to be sent to the message queue when `MPCauseNotification` (page 16) is called.

notifyParam2
If `anID` specifies a message queue, this parameter should contain the second pointer-sized value of the message to be sent to the message queue when `MPCauseNotification` (page 16) is called. Pass `NULL` if you don't need this parameter.

notifyParam3
If `anID` specifies a message queue, this parameter should contain the third pointer-sized value of the message sent to the message queue when `MPCauseNotification` (page 16) is called. Pass `NULL` if you don't need this parameter.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73).

Discussion
You specify the parameters for the simple notifications just as if you were calling the `MPSetTimerNotify` (page 40) function.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPModifyNotificationParameters

```
OSStatus MPModifyNotificationParameters (
   MPNotificationID notificationID,
   MPOpaqueIDClass kind,
   void *notifyParam1,
   void *notifyParam2,
   void *notifyParam3
);
```

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73).

Availability
Available in Mac OS X v10.1 and later.

Declared In
`Multiprocessing.h`

MPNotifyQueue

Sends a message to the specified message queue.

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

Parameters

queue
The queue ID of the message queue you want to notify.

param1
The first pointer-sized value of the message to send.

param2
The second pointer-sized value of the message to send.

param3
The third pointer-sized value of the message to send.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73).

Discussion
This function sends a message to the specified queue, which consist of the three parameters, `param1`, `param2`, and `param3`. The system does not interpret the three values which comprise the text of the message. If tasks are waiting on the specified queue, the first waiting task is unblocked and the task's `MPWaitOnQueue` (page 46) function completes.

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

You can call this function from an interrupt handler if messages are reserved on the queue. For more information about queueing modes and reserving messages, see `MPSetQueueReserve` (page 37).

Also see the function `MPWaitOnQueue` (page 46).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPProcessors

Returns the number of processors on the host computer.

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

Return Value
The number of physical processors on the host computer.

Discussion
See also the function `MPProcessorsScheduled` (page 33).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPProcessorsScheduled

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

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

Return Value
The number of active processors available on the host computer.

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

See also the function `MPProcessors` (page 33).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPRegisterDebugger

Registers a debugger.

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

Parameters

queue

The ID of the queue to which you want exception messages and other information to be sent.

level

The level of this debugger with respect to other debuggers. Exceptions and informational messages are sent first to the debugger with the highest level. If more than one debugger attempts to register at a particular level, only the first debugger is registered. Other attempts return an error.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73).

Discussion
In Mac OS X, this function is available but is not implemented. Use system debugging services to write a debugger for Mac OS X.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPRemoteCall

Calls a non-reentrant function and blocks the current task.

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

Parameters

remoteProc

A pointer to the application-defined function you want to call. See `MPRemoteProcedure` (page 48) for more information about the form of this function.

parameter

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

context

This parameter is ignored; specify `kMPOwningProcessRemoteContext`.

Return Value

The value that your remote procedure callback returned.

Discussion

You use this function to execute code on your application's main task. The *remoteProc* function is scheduled on the application's main run loop and run in the default mode (`kCFRunloopDefaultMode`). If you call this function from your application's main task, the *remoteProc* function is executed immediately in the current mode without blocking the task; otherwise, calling this function blocks the current task until the remote call completes.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`Multiprocessing.h`

MPRemoteCallCFM

Calls a non-reentrant function and blocks the current task.

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

Parameters

remoteProc

A pointer to the application-defined CFM (Code Fragment Manager) function you want to call. See `MPRemoteProcedure` (page 48) for more information about the form of this function.

parameter

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

context

This parameter is ignored; specify `kMPOwningProcessRemoteContext`.

Return Value
The value that your remote procedure callback returned.

Discussion
You use this function to execute code on your application's main task. The *remoteProc* function is scheduled on the application's main run loop and run in the default mode (`kCFRunloopDefaultMode`). If you call this function from your application's main task, the *remoteProc* function is executed immediately in the current mode without blocking the task; otherwise, calling this function blocks the current task until the remote call completes.

Availability
Available in Mac OS X v10.4 and later.

Declared In
`Multiprocessing.h`

MPSetEvent

Merges event flags into a specified event group.

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

Parameters

event
The ID of the event group you want to set.

flags
The flags you want to merge into the event group.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73).

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

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

Also see the function `MPWaitForEvent` (page 45).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPSetExceptionHandler

Sets an exception handler for a task.

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

Parameters

task

The task to associate with the exception handler.

exceptionQ

The message queue to which an exception message will be sent.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73).

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

- The first pointer-sized value contains the ID of the task in which the exception occurred.
- The second pointer-sized value contains the type of exception that occurred. See the header file `MachineExceptions.h` for a listing of exception types.
- The last pointer-sized value is set to `NULL` (reserved for future use).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPSetQueueReserve

Reserves space for messages on a specified message queue.

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

Parameters

queue

The ID of the queue whose messages you want to reserve.

count

The number of messages to reserve.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73).

Discussion
`MPNotifyQueue` (page 32) allocates spaces for messages dynamically; that is, memory to hold the message is allocated for the queue at the time of the call. In most cases this method is both speed and storage efficient. However, it is possible that, due to lack of memory resources, space for the message may not be available at the time of the call; in such cases, `MPNotifyQueue` (page 32) will return `kInsufficientResourcesErr`.

If you must have guaranteed message delivery, or if you need to call `MPNotifyQueue` (page 32) from an interrupt handler, you should reserve space on the specified queue by calling `MPSetQueueReserve`. Because such allocated space is reserved for duration of the queue's existence, you should avoid straining internal system resources by reserving messages only when absolutely necessary. Note that if you have reserved messages on a queue, additional space cannot be added dynamically if the number of messages exceeds the number reserved for that queue.

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

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPSetTaskState

Sets state information for a suspended task.

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

Parameters

task

The task whose state information you want to set.

kind

The kind of state information you want to set. See "Task State Constants" (page 70) for a listing of possible values. Note that some state information is read-only and cannot be changed using this function.

info

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

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73). If you specify `kMPTaskState32BitMemoryException` for the state information, this function returns `kMPInsufficientResourcesErr`, since the exception state information is read-only. Attempting to set state information for a running task will also return `kMPInsufficientResourcesErr`.

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

Also see the function `MPExtractTaskState` (page 28).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPSetTaskStorageValue

Sets the storage value for a given index number.

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

Parameters

index
The index number whose storage value you want to set.

value
The value you want to set.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73).

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

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

Also see the function `MPGetTaskStorageValue` (page 30).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPSetTaskType

Sets the type of the task.

```
OSStatus MPSetTaskType (
   MPTaskID task,
   OSType taskType
);
```

Return Value
The `noErr` result code. See "Multiprocessing Services Result Codes" (page 73).

Discussion
This function does nothing and should not be used.

Availability
Available in Mac OS X v10.1 and later.

Declared In
`Multiprocessing.h`

MPSetTaskWeight

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

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

Parameters

task
The ID of the task to which you want to assign a weighting.

weight
The relative weight to assign. This value can range from 1 to 10,000, with the default value being 100.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73).

Discussion
The approximate processor share is defined as:

```
weight of the task / total weight of available tasks
```

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

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPSetTimerNotify

Sets the notification information associated with a timer.

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

Parameters

timerID

The ID of the timer whose notification information you want to set.

notificationID

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

notifyParam1

If `anID` specifies an event group, this parameter should contain the flags to set in the event group when the timer expires. If `anID` specifies a message queue, this parameter should contain the first pointer-sized value of the message to be sent to the message queue when the timer expires.

notifyParam2

If `anID` specifies a message queue, this parameter should contain the second pointer-sized value of the message to be sent to the message queue when the timer expires. Pass `NULL` if you don't need this parameter.

notifyParam3

If `anID` specifies a message queue, this parameter should contain the third pointer-sized value of the message sent to the message queue when the timer expires. Pass `NULL` if you don't need this parameter.

Return Value

A result code. See "Multiprocessing Services Result Codes" (page 73).

Discussion

When the timer expires, Multiprocessing Services checks the notification ID, and if it is valid, notifies the related notification mechanisms (that is, event groups, queues, or semaphores) you had specified in your `MPSetTimerNotify` (page 40) calls.

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

You can set the notification information at any time. If the timer is armed, it will modify the notification parameters dynamically. If the timer is disarmed, it will modify the notification parameters to be used for the next `MPArmTimer` (page 13) call.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`Multiprocessing.h`

MPSignalSemaphore

Signals a semaphore.

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

Parameters

semaphore

The ID of the semaphore you want to signal.

Return Value

A result code. See "Multiprocessing Services Result Codes" (page 73). If the value of the semaphore was already at the maximum, `MPSignalSemaphore` returns `kInsufficientResourcesErr`.

Discussion

If tasks are waiting on the semaphore, the oldest (first queued) task is unblocked so that the corresponding `MPWaitOnSemaphore` (page 47) call for that task completes. Otherwise, if the value of the semaphore is not already equal to its maximum value, it is incremented by one.

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

Availability

Available in Mac OS X v10.0 and later.

Declared In

`Multiprocessing.h`

MPTaskIsPreemptive

Determines whether a task is preemptively scheduled.

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

Parameters

taskID

The task you want to check. Pass `kMPNoID` or `kInvalidID` if you want to specify the current task.

Return Value

If true, the task is preemptively scheduled. If false, the task is cooperatively scheduled.

Discussion

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

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

Availability

Available in Mac OS X v10.0 and later.

Declared In

`Multiprocessing.h`

MPTerminateTask

Terminates an existing task.

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

Parameters

task

The ID of the task you wish to terminate.

terminationStatus

A value of type `OSStatus` indicating termination status. This value is sent to the termination status message queue you specified in `MPCreateTask` (page 19) in place of the task function's result code.

Return Value

A result code. See "Multiprocessing Services Result Codes" (page 73). If the task to be terminated is already in the process of termination, `MPTerminateTask` returns `kMPInsufficientResourcesErr`. You do not need to take any additional action if this occurs.

Discussion

You should be very careful when calling `MPTerminateTask`. As defined, this call will asynchronously and abruptly terminate a task, potentially leaving whatever structures or resources it was operating upon in an indeterminate state. Mac OS X exacerbates this problem, as MP tasks can use many more system services that are not expecting client threads to asynchronously terminate, and these services do not take the rather complicated steps necessary to protect against, or recover from, such a situation.

However, there are situations in which calling `MPTerminateTask` is useful and relatively safe. One such situation is when your application or service is quitting and you know that the task you wish to terminate is waiting on an MP synchonization construct (queue, event, semaphore or critical region). While you could do this more cleanly by waking the task and causing it to exit on its own, doing so may not always be practical.

For example, suppose you have several service tasks performing background processing for your application. These service tasks wait on a queue, onto which the application places requests for processing. When the task is done with a request, it notifies another queue, which the application polls. Since the main application task is placing items on the shared queue, and receiving notifications when the requests are done, it can track whether or not there are outstanding requests being processed. If all outstanding requests have, in fact, been processed, it is relatively safe to terminate a task (or all tasks) waiting on the request queue.

You should not assume that the task has completed termination when this call returns; the proper way to synchronize with task termination is to wait on the termination queue (specified in `MPCreateTask` (page 19)) until a message appears. Because task termination is a multistage activity, it is possible for a preemptive task to attempt to terminate a task that is already undergoing termination. In such cases, `MPTerminateTask` returns `kMPInsufficientResourcesErr`.

Note that Multiprocessing Services resources (event groups, queues, semaphores, and critical regions) owned by a preemptive task are not released when that task terminates. If a task has a critical region locked when it terminates, the critical region remains in the locked state. Multiprocessing Services resources no longer needed should be explicitly deleted by the task that handles the termination message. All Multiprocessing Services resources created by tasks are released when their owning process (that is, the host application) terminates.

Availability

Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPThrowException

Throws an exception to a specified task.

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

Parameters

task

The task to which the exception should be thrown.

kind

The type of exception to give to the task.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73). If the task is already suspended or if the task is not defined to take thrown exceptions, the function returns `kMPInsufficientResourcesErr`.

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

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

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

In Mac OS X, this function is available but is not implemented.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPUnregisterDebugger

Unregisters a debugger.

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

Parameters

queue

The ID of the queue whose debugger you want to unregister.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73).

Discussion
In Mac OS X, this function is available but is not implemented. Use system debugging services to write a debugger for Mac OS X.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPWaitForEvent

Retrieves event flags from a specified event group.

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

Parameters

event

The event group whose flags you want to retrieve.

flags

On return, `flags` contains the flags of the specified event group. Pass `NULL` if you do not need any flag information.

timeout

The maximum time to wait for events before timing out. See "Timer Duration Constants" (page 71) for a list of constants you can use to specify the wait interval.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73).

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

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

Also see the function `MPSetEvent` (page 36).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPWaitOnQueue

Obtains a message from a specified message queue.

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

Parameters

queue
The ID of the message queue from which to receive the notification.

param1
On return, the first pointer-sized value of the notification message. Pass `NULL` if you do not need this portion of the message.

param2
On return, the second pointer-sized value of the notification message. Pass `NULL` if you do not need this portion of the message.

param3
On return, the third pointer-sized value of the notification message. Pass `NULL` if you do not need this portion of the message.

timeout
The time to wait for a notification before timing out. See "Timer Duration Constants" (page 71) for a list of constants you can use to specify the wait interval.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73).

Discussion
This function receives a message from the specified message queue. If no messages are currently available, the timeout specifies how long the function should wait for one. Tasks waiting on the queue are handled in a first in, first out manner; that is, the first task to wait on the queue receives the message from the `MPNotifyQueue` (page 32) call.

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

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

Also see the function `MPNotifyQueue` (page 32).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPWaitOnSemaphore

Waits on a semaphore

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

Parameters

semaphore
The ID of the semaphore you want to wait on.

timeout
The maximum time the function should wait before timing out. See "Timer Duration Constants" (page 71) for a list of constants you can use to specify the wait interval.

Return Value
A result code. See "Multiprocessing Services Result Codes" (page 73).

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

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

Also see the function `MPSignalSemaphore` (page 42).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPYield

Allows a task to yield the processor to another task.

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

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

In most cases you should not need to call this function. The most common use of `MPYield` is to release the processor when a task is in a loop in which further progress is dependent on other tasks, and the task cannot be blocked by waiting on a Multiprocessing Services resource. You should avoid such busy waiting whenever possible.

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

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

_MPIsFullyInitialized

Indicates whether Multiprocessing Services is available for use.

```
Boolean _MPIsFullyInitialized (
   void
);
```

Return Value
If true, Multiprocessing Services is available for use; otherwise, false.

Declared In
`Multiprocessing.h`

Callbacks

MPRemoteProcedure

Defines a remote procedure call.

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

For example, this is how you would declare the application-defined function if you were to name the function `MyRemoteProcedure`:

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

Parameters

parameter
A pointer to the application-defined value you passed to the function `MPRemoteCallCFM` (page 35). For example, this value could point to a data structure or a memory location.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

TaskProc

Defines the entry point of a task.

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

For example, this is how you would declare the application-defined function if you were to name the function `MyTaskProc`:

```
OSStatus MyTaskProc (
    void *parameter
);
```

Parameters

parameter

A pointer to the application-defined value you passed to the function `MPCreateTask` (page 19). For example, this value could point to a data structure or a memory location.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

Data Types

MPAddressSpaceID

```
typedef struct OpaqueMPAddressSpaceID * MPAddressSpaceID;
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPAddressSpaceInfo

```
struct MPAddressSpaceInfo {
    PBVersion version;
    MPProcessID processID;
    MPCoherenceID groupID;
    ItemCount nTasks;
    UInt32 vsid[16];
};
typedef struct MPAddressSpaceInfo MPAddressSpaceInfo;
```

Availability
Available in Mac OS X v10.1 and later.

Declared In
`MultiprocessingInfo.h`

MPAreaID

```
typedef struct OpaqueMPAreaID * MPAreaID;
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPCoherenceID

Represents a memory coherence group.

```
typedef struct OpaqueMPCoherenceID * MPCoherenceID;
```

Discussion
A coherence group is the set of processors and other bus controllers that have cache-coherent access to memory. Mac OS 9 defines only one coherence group, which is all the processors that can access internal memory (RAM). Other coherence groups are possible; for example, a PCI card with its own memory and processors can comprise a coherence group.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPConsoleID

```
typedef struct OpaqueMPConsoleID * MPConsoleID;
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPCpuID

Represents a CPU ID.

```
typedef struct OpaqueMPCpuID * MPCpuID;
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPCriticalRegionID

Represents a critical region ID, which Multiprocessing Services uses to manipulate critical regions.

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

Discussion
You obtain a critical region ID by calling the function `MPCreateCriticalRegion` (page 16).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPCriticalRegionInfo

```
struct MPCriticalRegionInfo {
    PBVersion version;
    MPProcessID processID;
    OSType regionName;
    ItemCount nWaiting;
    MPTaskID waitingTaskID;
    MPTaskID owningTask;
    ItemCount count;
};
typedef struct MPCriticalRegionInfo MPCriticalRegionInfo;
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MultiprocessingInfo.h`

MPEventFlags

Represents event information for an event group.

```
typedef UInt32 MPEventFlags;
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPEventID

Represents an event group ID, which Multiprocessing Services uses to manipulate event groups.

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

Discussion
You obtain an event group ID by calling the function `MPCreateEvent` (page 17).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPEventInfo

```
struct MPEventInfo {
    PBVersion version;
    MPProcessID processID;
    OSType eventName;
    ItemCount nWaiting;
    MPTaskID waitingTaskID;
    MPEventFlags events;
};
typedef struct MPEventInfo MPEventInfo;
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MultiprocessingInfo.h`

MPExceptionKind

Represents the kind of exception thrown.

```
typedef UInt32 MPExceptionKind;
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
Multiprocessing.h

MPNotificationID

Represents a notification ID, which Multiprocessing Services uses to manipulate kernel notifications.

```
typedef struct OpaqueMPNotificationID * MPNotificationID;
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
Multiprocessing.h

MPNotificationInfo

```
struct MPNotificationInfo {
    PBVersion version;
    MPProcessID processID;
    OSType notificationName;
    MPQueueID queueID;
    void * p1;
    void * p2;
    void * p3;
    MPEventID eventID;
    MPEventFlags events;
    MPSemaphoreID semaphoreID;
};
typedef struct MPNotificationInfo MPNotificationInfo;
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
MultiprocessingInfo.h

MPOpaqueID

Represents a generic notification ID (that is, an ID that could be a queue ID, event ID, kernel notification ID, or semaphore ID).

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

Availability
Available in Mac OS X v10.0 and later.

Declared In
Multiprocessing.h

MPOpaqueIDClass

```
typedef UInt32 MPOpaqueIDClass;
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPPageSizeClass

```
typedef UInt32 MPPageSizeClass;
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPProcessID

Represents a process ID.

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

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

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPQueueID

Represents a queue ID, which Multiprocessing Services uses to manipulate message queues.

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

Discussion
You obtain a queue ID by calling the function `MPCreateQueue` (page 18).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPQueueInfo

```
struct MPQueueInfo {
    PBVersion version;
    MPProcessID processID;
    OSType queueName;
    ItemCount nWaiting;
    MPTaskID waitingTaskID;
    ItemCount nMessages;
    ItemCount nReserved;
    void * p1;
    void * p2;
    void * p3;
};
typedef struct MPQueueInfo MPQueueInfo;
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MultiprocessingInfo.h`

MPSemaphoreCount

Represents a semaphore count.

```
typedef ItemCount MPSemaphoreCount;
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPSemaphoreID

Represents a semaphore ID, which Multiprocessing Services uses to manipulate semaphores.

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

Discussion
You obtain a semaphore ID by calling the function `MPCreateSemaphore` (page 18).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPSemaphoreInfo

```
struct MPSemaphoreInfo {
    PBVersion version;
    MPProcessID processID;
    OSType semaphoreName;
    ItemCount nWaiting;
    MPTaskID waitingTaskID;
    ItemCount maximum;
    ItemCount count;
};
typedef struct MPSemaphoreInfo MPSemaphoreInfo;
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MultiprocessingInfo.h`

MPTaskID

Represents a task ID.

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

Discussion
You obtain a task ID by calling the function `MPCreateTask` (page 19).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPTaskInfo

Contains information about a task.

```
struct MPTaskInfo {
    PBVersion version;
    OSType name;
    OSType queueName;
    UInt16 runState;
    UInt16 lastCPU;
    UInt32 weight;
    MPProcessID processID;
    AbsoluteTime cpuTime;
    AbsoluteTime schedTime;
    AbsoluteTime creationTime;
    ItemCount codePageFaults;
    ItemCount dataPageFaults;
    ItemCount preemptions;
    MPCpuID cpuID;
    MPOpaqueID blockedObject;
    MPAddressSpaceID spaceID;
    LogicalAddress stackBase;
    LogicalAddress stackLimit;
    LogicalAddress stackCurr;
};
typedef struct MPTaskInfo MPTaskInfo;
```

Fields

`version`
: The version of this data structure.

`name`
: The name of the task.

`queueName`
: A four-byte code indicating the status of the queue waiting on the task.

`runState`
: The current state of the task (running, ready, or blocked).

`lastCPU`
: The address of the last processor that ran this task.

`weight`
: The weighting assigned to this task.

`processID`
: The ID of the process that owns this task.

`cpuTime`
: The accumulated CPU time used by the task.

`schedTime`
: The time when the task was last scheduled.

`creationTime`
: The time when the task was created.

`codePageFaults`
: The number of page faults that occurred during code execution.

`dataPageFaults`
: The number of page faults that occurred during data access.

`preemptions`
: The number of times this task was preempted.

`cpuID`
The ID of the last processor that ran this task.

`blockedObject`
Reserved for use by Mac OS X.

`spaceID`
Address space ID of this task.

`stackBase`
The lowest memory address of the task's stack.

`stackLimit`
The highest memory address of the task's stack.

`stackCurr`
The current stack address.

Discussion
If you specify the `kMPTaskStateTaskInfo` constant when calling the function `MPExtractTaskState` (page 28) , Multiprocessing Services returns state information in an `MPTaskInfo` structure.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPTaskInfoVersion2

```
struct MPTaskInfoVersion2 {
    PBVersion version;
    OSType name;
    OSType queueName;
    UInt16 runState;
    UInt16 lastCPU;
    UInt32 weight;
    MPProcessID processID;
    AbsoluteTime cpuTime;
    AbsoluteTime schedTime;
    AbsoluteTime creationTime;
    ItemCount codePageFaults;
    ItemCount dataPageFaults;
    ItemCount preemptions;
    MPCpuID cpuID;
};
typedef struct MPTaskInfoVersion2 MPTaskInfoVersion2;
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPTaskStateKind

```
typedef UInt32 MPTaskStateKind;
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPTaskWeight

Represents the relative processor weighting of a task.

```
typedef UInt32 MPTaskWeight;
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

MPTimerID

Represents a timer ID.

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

Discussion
You obtain a timer ID by calling the function `MPCreateTimer` (page 20).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

TaskStorageIndex

Represents a task storage index value used by functions described in "Accessing Per-Task Storage Variables."

```
typedef ItemCount TaskStorageIndex;
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

TaskStorageValue

Represents a task storage value used by functions described in "Accessing Per-Task Storage Variables."

```
typedef LogicalAddress TaskStorageValue;
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Multiprocessing.h`

Constants

Allocation constants

The maximum memory allocation size.

```
enum {
    kMPMaxAllocSize = 1024L * 1024 * 1024
};
```

Constants
`kMPMaxAllocSize`
The maximum allocation size: 1GB.
Available in Mac OS X v10.0 and later.
Declared in `Multiprocessing.h`.

Task IDs

Use to specify no task ID.

```
enum {
    kMPNoID = kInvalidID
};
```

Constants
`kMPNoID`
No task ID.
Available in Mac OS X v10.0 and later.
Declared in `Multiprocessing.h`.

Discussion
Used when calling `MPTaskIsPreemptive` (page 42) if you want to specify the current task.

Data Structure Version Constants

Data structure version information constants.

```
enum {
    kMPQueueInfoVersion = 1L | (kOpaqueQueueID << 16),
    kMPSemaphoreInfoVersion = 1L | (kOpaqueSemaphoreID << 16),
    kMPEventInfoVersion = 1L | (kOpaqueEventID << 16),
    kMPCriticalRegionInfoVersion = 1L | (kOpaqueCriticalRegionID << 16),
    kMPNotificationInfoVersion = 1L | (kOpaqueNotificationID << 16),
    kMPAddressSpaceInfoVersion = 1L | (kOpaqueAddressSpaceID << 16)
};
```

Constants

`kMPQueueInfoVersion`
The `MPQueueInfo` structure version.
Available in Mac OS X v10.0 and later.
Declared in `MultiprocessingInfo.h`.

`kMPSemaphoreInfoVersion`
The `MPSemaphoreInfo` structure version.
Available in Mac OS X v10.0 and later.
Declared in `MultiprocessingInfo.h`.

`kMPEventInfoVersion`
The `MPEventInfo` structure version.
Available in Mac OS X v10.0 and later.
Declared in `MultiprocessingInfo.h`.

`kMPCriticalRegionInfoVersion`
The `MPCriticalRegionInfo` structure version.
Available in Mac OS X v10.0 and later.
Declared in `MultiprocessingInfo.h`.

`kMPNotificationInfoVersion`
The `MPNotificationInfo` structure version.
Available in Mac OS X v10.0 and later.
Declared in `MultiprocessingInfo.h`.

`kMPAddressSpaceInfoVersion`
The `MPAddressSpaceInfo` structure version.
Available in Mac OS X v10.1 and later.
Declared in `MultiprocessingInfo.h`.

Values for the MPOpaqueIDClass type

Constants indicating the source of a generic notification.

```
enum {
    kOpaqueAnyID = 0,
    kOpaqueProcessID = 1,
    kOpaqueTaskID = 2,
    kOpaqueTimerID = 3,
    kOpaqueQueueID = 4,
    kOpaqueSemaphoreID = 5,
    kOpaqueCriticalRegionID = 6,
    kOpaqueCpuID = 7,
    kOpaqueAddressSpaceID = 8,
    kOpaqueEventID = 9,
    kOpaqueCoherenceID = 10,
    kOpaqueAreaID = 11,
    kOpaqueNotificationID = 12,
    kOpaqueConsoleID = 13
};
```

Constants

`kOpaqueAnyID`
Any source.
Available in Mac OS X v10.0 and later.
Declared in `Multiprocessing.h`.

`kOpaqueProcessID`
A process.
Available in Mac OS X v10.0 and later.
Declared in `Multiprocessing.h`.

`kOpaqueTaskID`
A task.
Available in Mac OS X v10.0 and later.
Declared in `Multiprocessing.h`.

`kOpaqueTimerID`
A timer.
Available in Mac OS X v10.0 and later.
Declared in `Multiprocessing.h`.

`kOpaqueQueueID`
A queue.
Available in Mac OS X v10.0 and later.
Declared in `Multiprocessing.h`.

`kOpaqueSemaphoreID`
A semaphore.
Available in Mac OS X v10.0 and later.
Declared in `Multiprocessing.h`.

`kOpaqueCriticalRegionID`
A critical region.
Available in Mac OS X v10.0 and later.
Declared in `Multiprocessing.h`.

`kOpaqueCpuID`
A CPU.

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

`kOpaqueAddressSpaceID`
An address space.

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

`kOpaqueEventID`
An event.

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

`kOpaqueCoherenceID`
A coherence group.

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

`kOpaqueAreaID`
An area.

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

`kOpaqueNotificationID`
A notification.

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

`kOpaqueConsoleID`
A console.

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

Memory Allocation Alignment Constants

Specify the alignment of the desired memory block when calling the `MPAllocateAligned` function.

```
enum {
    kMPAllocateDefaultAligned = 0,
    kMPAllocate8ByteAligned = 3,
    kMPAllocate16ByteAligned = 4,
    kMPAllocate32ByteAligned = 5,
    kMPAllocate1024ByteAligned = 10,
    kMPAllocate4096ByteAligned = 12,
    kMPAllocateMaxAlignment = 16,
    kMPAllocateAltiVecAligned = kMPAllocate16ByteAligned,
    kMPAllocateVMXAligned = kMPAllocateAltiVecAligned,
    kMPAllocateVMPageAligned = 254,
    kMPAllocateInterlockAligned = 255
};
```

Constants

`kMPAllocateDefaultAligned`
Use the default alignment.
Available in Mac OS X v10.0 and later.
Declared in `Multiprocessing.h`.

`kMPAllocate8ByteAligned`
Use 8-byte alignment.
Available in Mac OS X v10.0 and later.
Declared in `Multiprocessing.h`.

`kMPAllocate16ByteAligned`
Use 16-byte alignment.
Available in Mac OS X v10.0 and later.
Declared in `Multiprocessing.h`.

`kMPAllocate32ByteAligned`
Use 32-byte alignment.
Available in Mac OS X v10.0 and later.
Declared in `Multiprocessing.h`.

`kMPAllocate1024ByteAligned`
Use 1024-byte alignment.
Available in Mac OS X v10.0 and later.
Declared in `Multiprocessing.h`.

`kMPAllocate4096ByteAligned`
Use 4096-byte alignment.
Available in Mac OS X v10.0 and later.
Declared in `Multiprocessing.h`.

`kMPAllocateMaxAlignment`
Use the maximum alignment (65536 byte).
Available in Mac OS X v10.0 and later.
Declared in `Multiprocessing.h`.

`kMPAllocateAltiVecAligned`
Use AltiVec alignment.
Available in Mac OS X v10.0 and later.
Declared in `Multiprocessing.h`.

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

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

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

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

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

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

Memory Allocation Option Constants

Specify optional actions when calling the `MPAllocateAligned` function.

```
enum {
    kMPAllocateClearMask = 0x0001,
    kMPAllocateGloballyMask = 0x0002,
    kMPAllocateResidentMask = 0x0004,
    kMPAllocateNoGrowthMask = 0x0010,
    kMPAllocateNoCreateMask = 0x0020
};
```

Constants

`kMPAllocateClearMask`
Zero out the allocated memory block.

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

`kMPAllocateGloballyMask`
Allocate memory from in memory space that is visible to all processes. Note that such globally-allocated space is not automatically reclaimed when the allocating process terminates. By default, `MPAllocateAligned` (page 12) allocates memory from process-specific (that is, not global) memory.

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

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

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

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

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

`kMPAllocateNoCreateMask`
Do not attempt to create the pool if it does not yet exist.

Available in Mac OS X v10.1 and later.

Declared in `Multiprocessing.h`.

MPDebuggerLevel

Indicates the debugger level.

```
typedef UInt32 MPDebuggerLevel;
enum {
    kMPLowLevelDebugger = 0x00000000,
    kMPMidLevelDebugger = 0x10000000,
    kMPHighLevelDebugger = 0x20000000
};
```

Constants

`kMPLowLevelDebugger`
The low-level debugger.

Available in Mac OS X v10.1 and later.

Declared in `Multiprocessing.h`.

`kMPMidLevelDebugger`
The mid-level debugger.

Available in Mac OS X v10.1 and later.

Declared in `Multiprocessing.h`.

`kMPHighLevelDebugger`
The high-level debugger.

Available in Mac OS X v10.1 and later.

Declared in `Multiprocessing.h`.

Library Version Constants

Identifies the current library version.

```
enum {
    MPLibrary_MajorVersion = 2,
    MPLibrary_MinorVersion = 3,
    MPLibrary_Release = 1,
    MPLibrary_DevelopmentRevision = 1
};
```

Constants

`MPLibrary_MajorVersion`
Major version number.

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

`MPLibrary_MinorVersion`
Minor version number.

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

`MPLibrary_Release`
Release number.

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

`MPLibrary_DevelopmentRevision`
Development revision number.

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

Remote Call Context Option Constants

Specify which contexts are allowed to execute the callback function when using `MPRemoteCall`.

```
enum {
    kMPAnyRemoteContext = 0,
    kMPOwningProcessRemoteContext = 1,
    kMPInterruptRemoteContext = 2,
    kMPAsyncInterruptRemoteContext = 3
};
typedef UInt8 MPRemoteContext;
```

Constants

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

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

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

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

`kMPInterruptRemoteContext`
Unsupported in Mac OS X.

Available in Mac OS X v10.1 and later.

Declared in `Multiprocessing.h`.

`kMPAsyncInterruptRemoteContext`
Unsupported in Mac OS X.

Available in Mac OS X v10.1 and later.

Declared in `Multiprocessing.h`.

Discussion

These constants are used to support older versions of Mac OS and are ignored in Mac OS X.

Task Creation Options

Specify optional actions when calling the `MPCreateTask` function.

```
enum {
    kMPCreateTaskSuspendedMask = 1L << 0,
    kMPCreateTaskTakesAllExceptionsMask = 1L << 1,
    kMPCreateTaskNotDebuggableMask = 1L << 2,
    kMPCreateTaskValidOptionsMask = kMPCreateTaskSuspendedMask |
kMPCreateTaskTakesAllExceptionsMask | kMPCreateTaskNotDebuggableMask
};
typedef OptionBits MPTaskOptions;
```

Constants

`kMPCreateTaskSuspendedMask`

Unsupported in Mac OS X.

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

`kMPCreateTaskTakesAllExceptionsMask`

The task will take all exceptions, including those normally handled by the system, such as page faults.

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

`kMPCreateTaskNotDebuggableMask`

Unsupported in Mac OS X.

Available in Mac OS X v10.1 and later.

Declared in `Multiprocessing.h`.

`kMPCreateTaskValidOptionsMask`

Include all valid options for this task.

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

Task Exception Disposal Constants

Specify actions to take on an exception when passed in the `action` parameter of the `MPDisposeTaskException` function.

```
enum {
    kMPTaskPropagate = 0,
    kMPTaskResumeStep = 1,
    kMPTaskResumeBranch = 2,
    kMPTaskResumeMask = 0x0000,
    kMPTaskPropagateMask = 1 << kMPTaskPropagate,
    kMPTaskResumeStepMask = 1 << kMPTaskResumeStep,
    kMPTaskResumeBranchMask = 1 << kMPTaskResumeBranch
};
```

Constants

`kMPTaskPropagate`
The exception is propagated.
Available in Mac OS X v10.0 and later.
Declared in `Multiprocessing.h`.

`kMPTaskResumeStep`
The task is resumed and single step is enabled.
Available in Mac OS X v10.0 and later.
Declared in `Multiprocessing.h`.

`kMPTaskResumeBranch`
The task is resumed and branch stepping is enabled.
Available in Mac OS X v10.0 and later.
Declared in `Multiprocessing.h`.

`kMPTaskResumeMask`
Resume the task.
Available in Mac OS X v10.0 and later.
Declared in `Multiprocessing.h`.

`kMPTaskPropagateMask`
Propagate the exception to the next debugger level.
Available in Mac OS X v10.0 and later.
Declared in `Multiprocessing.h`.

`kMPTaskResumeStepMask`
Resume the task and enable single stepping.
Available in Mac OS X v10.0 and later.
Declared in `Multiprocessing.h`.

`kMPTaskResumeBranchMask`
Resume the task and enable branch stepping.
Available in Mac OS X v10.0 and later.
Declared in `Multiprocessing.h`.

Task Information Structure Version Constant

Indicates the current version of the `MPTaskInfo` structure (returned as the first field).

```
enum {
    kMPTaskInfoVersion = 3
};
```

Constants

`kMPTaskInfoVersion`

The current version of the task information structure.

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

Task Run State Constants

Indicate the state of the task when returned as part of the `MPTaskInfo` data structure.

```
enum {
    kMPTaskBlocked = 0,
    kMPTaskReady = 1,
    kMPTaskRunning = 2
};
```

Constants

`kMPTaskBlocked`

The task is blocked..

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

`kMPTaskReady`

The task is ready for execution.

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

`kMPTaskRunning`

The task is currently running.

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

Task State Constants

Specify what states you want to set or obtain when calling the `MPExtractTaskState` or `MPSetTaskState` functions.

```
enum {
    kMPTaskStateRegisters = 0,
    kMPTaskStateFPU = 1,
    kMPTaskStateVectors = 2,
    kMPTaskStateMachine = 3,
    kMPTaskState32BitMemoryException = 4,
    kMPTaskStateTaskInfo = 5
};
```

Constants

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

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

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

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

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

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

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

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

`kMPTaskState32BitMemoryException`
The task's exception information for older 32-bit memory exceptions (that is, memory exceptions on 32-bit CPUs). The `MemoryExceptionInformation` structure in `MachineExceptions.h` defines the format of this information. This exception information is read-only.

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

`kMPTaskStateTaskInfo`
Static and dynamic information about the task, as described by the data structure `MPTaskInfo` (page 56). This task information is read-only.

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

Timer Duration Constants

Specify the maximum time a task should wait for an event to occur.

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

Constants

`kDurationImmediate`

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

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

`kDurationForever`

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

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

`kDurationMillisecond`

The task waits one millisecond before timing out.

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

`kDurationMicrosecond`

The task waits one microsecond before timing out.

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

Discussion

You can use these constants in conjunction with other values to indicate specific wait intervals. For example, to wait 1 second, you can pass `kDurationMillisecond * 1000`.

Timer Option Masks

Indicate optional actions when calling `MPArmTimer`.

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

Constants

`kMPPreserveTimerIDMask`

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

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

`kMPTimeIsDeltaMask`

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

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

`kMPTimeIsDurationMask`

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

Available in Mac OS X v10.0 and later.

Declared in `Multiprocessing.h`.

Result Codes

Result codes defined for Multiprocessing Services are listed below.

Result Code	Value	Description
`kMPIterationEndErr`	-29275	Available in Mac OS X v10.0 and later.
`kMPPrivilegedErr`	-29276	Available in Mac OS X v10.0 and later.
`kMPProcessCreatedErr`	-29288	Available in Mac OS X v10.0 and later.
`kMPProcessTerminatedErr`	-29289	Available in Mac OS X v10.0 and later.
`kMPTaskCreatedErr`	-29290	Available in Mac OS X v10.0 and later.
`kMPTaskBlockedErr`	-29291	The desired task is blocked. Available in Mac OS X v10.0 and later.
`kMPTaskStoppedErr`	-29292	The desired task is stopped. Available in Mac OS X v10.0 and later.
`kMPDeletedErr`	-29295	The desired notification the function was waiting upon was deleted. Available in Mac OS X v10.0 and later.
`kMPTimeoutErr`	-29296	The designated timeout interval passed before the function could take action. Available in Mac OS X v10.0 and later.
`kMPInsufficientResourcesErr`	-29298	Could not complete task due to unavailable Multiprocessing Services resources. Note that many functions return this value as a general error when the desired action could not be performed. Available in Mac OS X v10.0 and later.

Result Code	Value	Description
`kMPInvalidIDErr`	-29299	Invalid ID value. For example, an invalid message queue ID was passed to `MPNotifyQueue`. Available in Mac OS X v10.0 and later.

Gestalt Constants

You can determine which system software calls are preemptively-safe for Multiprocessing Services by using the preemptive function attribute selectors defined in the Gestalt Manager. For more information, see *Gestalt Manager Reference*.

Document Revision History

This table describes the changes to *Multiprocessing Services Reference*.

Date	Notes
2008-02-08	Updated the description of the MPRemoteCall function.
2007-10-31	Updated for Mac OS X v10.5.
2005-07-07	Made minor technical corrections.
2003-02-01	Updated formatting and linking.

Document Revision History

Index

Symbols

A

D

K

L

M

R

T

V