Thread Manager Reference

Carbon > Process Management

2007-04-04

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

Simultaneously published in the United States and Canada.

Contents

Thread Manager Reference

Framework: CoreServices/CoreServices.h

Declared in Threads.h

Overview

You can use the Thread Manager to provide cooperatively scheduled threads, or multiple points of execution, in an application. You can think of the Thread Manager as an enhancement to the classic Mac OS Process Manager, which governs how applications work together in the Mac OS cooperative multitasking environment.

Important: Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

Consider using the Thread Manager for applications with more than one thread if these threads can execute only in the cooperative multitasking environment of the classic Mac OS Process Manager.

Alternatively, you should consider using the Multiprocessing Services to implement separate paths of execution for tasks that are reentrant and can therefore can be preemptively scheduled.

Using Thread Manager routines, you can create threads and thread pools and set them up to run; turn scheduling on and off; work with stacks; create dialog boxes that yield control to other threads; pass information between threads; install custom scheduling and context-switching functions; and use threads to make asynchronous I/O calls.

The Thread Manager provides only cooperative threading for PowerPC applications. Applications can use the Multiprocessing Services API to create preemptively scheduled tasks.

Note that several Thread Manager functions that did not require you to pass universal procedure pointers (UPPs) for callbacks now require them in Carbon. See the Carbon Porting Notes for more information.

Functions by Task

Creating and Disposing of Threads

`DisposeThread` (page 13)
Deletes a thread when it finishes executing.

`NewThread` (page 24)
Creates a new thread or allocates one from the existing pool of threads.

Creating and Getting Information About Thread Pools

`CreateThreadPool` (page 10)
Creates a pool of threads for your application.

`GetDefaultThreadStackSize` (page 17)
Determines the default stack size required by a thread.

`GetFreeThreadCount` (page 55) Deprecated in Mac OS X v10.3
Determines how many threads are available to be allocated in a thread pool. (Deprecated. There is no replacement.)

`GetSpecificFreeThreadCount` (page 56) Deprecated in Mac OS X v10.3
Determines how many threads with a stack size equal to or greater than the specified size are available to be allocated in a thread pool. (Deprecated. There is no replacement.)

Getting Information About Specific Threads

`GetCurrentThread` (page 16)
Obtains the thread ID of the currently executing thread.

`GetThreadState` (page 18)
Obtains the state of a thread.

`ThreadCurrentStackSpace` (page 37)
Determines the amount of stack space that is available for any thread in your application.

Getting Information and Scheduling Threads During Interrupts

`GetThreadCurrentTaskRef` (page 17)
Obtains a thread task reference.

`GetThreadStateGivenTaskRef` (page 19)
Obtains the state of a thread when your application is not necessarily the current process—for example, during execution of an interrupt function.

`SetThreadReadyGivenTaskRef` (page 30)
Changes the state of a thread from stopped to ready when your application is not the current process.

Installing Custom Scheduling, Switching, Terminating, and Debugging Functions

`SetDebuggerNotificationProcs` (page 28)
Installs functions that notify the debugger when a thread is created, disposed of, or scheduled.

`SetThreadScheduler` (page 30)
Installs a custom scheduling function (custom scheduler).

SetThreadSwitcher (page 34)
Installs a custom context-switching function for any thread.

SetThreadTerminator (page 35)
Installs a custom thread-termination function for a thread.

Preventing Scheduling

SetThreadStateEndCritical (page 33)
Changes the state of the current thread and exits that thread's critical section at the same time.

ThreadBeginCritical (page 36)
Indicates that the thread is entering a critical code section.

ThreadEndCritical (page 38)
Indicates that the thread is leaving a critical code section.

Scheduling Threads

SetThreadState (page 32)
Changes the state of any thread.

YieldToAnyThread (page 38)
Relinquishes the current thread's control.

YieldToThread (page 39)
Relinquishes the current thread's control to a particular thread.

Miscellaneous

DisposeDebuggerDisposeThreadUPP (page 12)

DisposeDebuggerNewThreadUPP (page 12)

DisposeDebuggerThreadSchedulerUPP (page 13)

DisposeThreadEntryUPP (page 14)

DisposeThreadSchedulerUPP (page 15)

DisposeThreadSwitchUPP (page 15)

DisposeThreadTerminationUPP (page 16)

InvokeDebuggerDisposeThreadUPP (page 20)

InvokeDebuggerNewThreadUPP (page 20)

Functions

CreateThreadPool

Creates a pool of threads for your application.

```
OSErr CreateThreadPool (
   ThreadStyle threadStyle,
   SInt16 numToCreate,
   Size stackSize
);
```

Parameters

threadStyle

The type of thread to create for this set of threads in the pool. Cooperative is the only type that you can specify. Historically, the Thread Manger supported two types of threads, preemptive and cooperative. However, due to severe limitations on their use, the Thread Manager no longer supports preemptive threads.

numToCreate

The number of threads to create for the pool.

stackSize

The stack size for this set of threads in the pool. This stack must be large enough to handle saved thread context, normal application stack usage, interrupt handling functions, and CPU exceptions. Specify a stack size of 0 to request the Thread Manager's default stack size for the specified type of thread.

Return Value
A result code. See "Thread Manager Result Codes" (page 54).

Discussion
The `CreateThreadPool` function creates the specified number of threads with the specified stack requirements. It places the threads that it creates into a pool for use by your application.

When you call `CreateThreadPool`, if the Thread Manager is unable to create all the threads that you specify, it does not create any at all and returns the `memFullErr` result code.

The threads in the pool are indistinguishable except by stack size. That is, you cannot refer to them individually. When you want to use a thread to execute some code in your application, you allocate a thread of a specific size from the pool using the `NewThread` function. The `NewThread` function assigns a thread ID to the thread and specifies the function that is the entry point to the thread.

Note that it is not strictly necessary to create a pool of threads before allocating a thread. If you wish, you can use the `NewThread` function to create and allocate a thread in one step. The advantage of using `CreateThreadPool` is that you can allocate memory for threads early in your application's execution before memory is used or fragmented.

Before making any calls to `CreateThreadPool`, be certain that you first have called the Memory Manager function `MaxApplZone` to extend the application heap to its limit. You must call `MaxApplZone` from the main application thread before any other threads in your application run.

To allocate a thread from the pool created with `CreateThreadPool`, use the `NewThread` (page 24) function.

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

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

Declared In
`Threads.h`

DisposeDebuggerDisposeThreadUPP

```
void DisposeDebuggerDisposeThreadUPP (
   DebuggerDisposeThreadUPP userUPP
);
```

Parameters

userUPP

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`Threads.h`

DisposeDebuggerNewThreadUPP

```
void DisposeDebuggerNewThreadUPP (
   DebuggerNewThreadUPP userUPP
);
```

Parameters

userUPP

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`Threads.h`

DisposeDebuggerThreadSchedulerUPP

```
void DisposeDebuggerThreadSchedulerUPP (
   DebuggerThreadSchedulerUPP userUPP
);
```

Parameters

userUPP

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`Threads.h`

DisposeThread

Deletes a thread when it finishes executing.

```
OSErr DisposeThread (
   ThreadID threadToDump,
   void *threadResult,
   Boolean recycleThread
);
```

Parameters

threadToDump

The thread ID of the thread to delete.

threadResult

A pointer to the thread's result. The `DisposeThread` function places this result to an address which you originally specify with the `threadResult` parameter of the `NewThread` function when you create or allocate the thread. Pass a value of `NULL` if you are not interested in obtaining a function result.

recycleThread

A Boolean value that specifies whether to return the thread to the allocation pool or to remove it entirely. Specify `False` to dispose of the thread entirely and `True` to return it to the thread pool.

Return Value

A result code. See "Thread Manager Result Codes" (page 54).

Discussion

When a thread finishes executing, the Thread Manager automatically calls `DisposeThread` to delete it. Therefore, the only reason for you to explicitly call `DisposeThread` is to recycle a terminating thread. To do so, set the `recycleThread` parameter to `True`. The Thread Manager clears out the thread's internal data structure, resets it, and puts the thread in the thread pool where it can be used again as necessary.

The `DisposeThread` function sets the `threadResult` parameter to the thread's function result. You allocate the storage for the thread result when you create or allocate a thread with the `NewThread` function.

You cannot explicitly dispose of the main application thread. If you attempt to do so, `DisposeThread` returns the `threadProtocolErr` result code.

When your application terminates, the Thread Manager calls `DisposeThread` to terminate any active threads. It terminates stopped and ready threads first but in no special order. It terminates the currently running thread last. This thread should always be the main application thread.

To install a callback function to do special cleanup when a thread terminates, use the `SetThreadTerminator` (page 35) function.

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

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

Declared In
`Threads.h`

DisposeThreadEntryUPP

```
void DisposeThreadEntryUPP (
   ThreadEntryUPP userUPP
);
```

Parameters

userUPP

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

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

Declared In
`Threads.h`

DisposeThreadSchedulerUPP

```
void DisposeThreadSchedulerUPP (
   ThreadSchedulerUPP userUPP
);
```

Parameters

userUPP

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

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

Declared In
Threads.h

DisposeThreadSwitchUPP

```
void DisposeThreadSwitchUPP (
   ThreadSwitchUPP userUPP
);
```

Parameters

userUPP

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

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

Declared In
Threads.h

DisposeThreadTerminationUPP

```
void DisposeThreadTerminationUPP (
   ThreadTerminationUPP userUPP
);
```

Parameters

userUPP

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`Threads.h`

GetCurrentThread

Obtains the thread ID of the currently executing thread.

```
OSErr GetCurrentThread (
    ThreadID * currentThreadID
);
```

Parameters

currentThreadID

On return, a pointer to the thread ID of the current thread.

Return Value

A result code. See "Thread Manager Result Codes" (page 54).

Discussion

You can use the thread ID obtained by `GetCurrentThread` in functions such as `GetThreadState` and `SetThreadState` to get and set the state of a thread.

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`Threads.h`

GetDefaultThreadStackSize

Determines the default stack size required by a thread.

```
OSErr GetDefaultThreadStackSize (
   ThreadStyle threadStyle,
   Size *stackSize
);
```

Parameters

threadStyle

The type of thread to get information about. Cooperative is the only type that you can specify. Historically, the Thread Manger supported two types of threads, preemptive and cooperative, but the Thread Manager no longer supports preemptive threads.

stackSize

On return, a pointer to the default stack size (in bytes). When you create a thread pool or an individual thread, this is the stack size that the Thread Manager allocates when you specify the default size.

Return Value

A result code. See "Thread Manager Result Codes" (page 54).

Discussion

Keep in mind that the default stack size is not an absolute value that you must use but is a rough estimate.

To determine how much stack space is available for a particular thread, use the `ThreadCurrentStackSpace` (page 37) function.

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`Threads.h`

GetThreadCurrentTaskRef

Obtains a thread task reference.

```
OSErr GetThreadCurrentTaskRef (
   ThreadTaskRef *threadTRef
);
```

Parameters

threadTRef

On return, a pointer to a thread task reference.

Return Value

A result code. See "Thread Manager Result Codes" (page 54).

Discussion
The thread task reference is somewhat of a misnomer because it identifies your application context, not a particular thread. Identifying your application context is necessary in situations where you aren't guaranteed that your application is the current context—such as during the execution of an interrupt function. In such cases, you need both the thread ID to identify the thread and the thread task reference to identify the application context.

After you obtain the thread task reference, you can use it in the `GetThreadStateGivenTaskRef` (page 19) and `SetThreadReadyGivenTaskRef` (page 30) functions to get and set information about specific threads in your application at times when you are not guaranteed that your application is the current context.

To get information about a thread when your application is not the current process, use the `GetThreadStateGivenTaskRef` function.

To change the state of a thread from stopped to ready when your application is not the current process, use the `SetThreadReadyGivenTaskRef` function.

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

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

Declared In
`Threads.h`

GetThreadState

Obtains the state of a thread.

```
OSErr GetThreadState (
   ThreadID threadToGet,
   ThreadState *threadState
);
```

Parameters

`threadToGet`
The thread ID of the thread about which you want information.

`threadState`
On return, a pointer to the state of the thread specified by `threadToGet`.

Return Value
A result code. See "Thread Manager Result Codes" (page 54).

Discussion
A thread can be in one of three states: ready to execute (`kReadyThreadState`), stopped (`kStoppedThreadState`), or executing (`kRunningThreadState`).

To change the state of a specified thread, use `SetThreadState` (page 32).

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`Threads.h`

GetThreadStateGivenTaskRef

Obtains the state of a thread when your application is not necessarily the current process—for example, during execution of an interrupt function.

```
OSErr GetThreadStateGivenTaskRef (
   ThreadTaskRef threadTRef,
   ThreadID threadToGet,
   ThreadState *threadState
);
```

Parameters

threadTRef

The thread task reference of the application containing the thread whose state you want to determine.

threadToGet

The thread ID of the thread whose state you want to determine.

threadState

A pointer to a thread state variable in which the function places the state of the specified thread.

Return Value

A result code. See "Thread Manager Result Codes" (page 54).

Discussion

You can use `GetThreadStateGivenTaskRef` at times when you aren't guaranteed that your application is the current context, such as during execution of an interrupt function. In such cases you must identify the thread task reference (the application context) as well as the thread ID.

To determine the thread task reference (application context) for your application, use the `GetThreadCurrentTaskRef` (page 17) function.

To change the state of a thread from stopped to ready when your application is not the current process, use the `SetThreadReadyGivenTaskRef` (page 30) function.

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

Availability

Available in Mac OS X v10.0 and later.

Declared In
Threads.h

InvokeDebuggerDisposeThreadUPP

```
void InvokeDebuggerDisposeThreadUPP (
   ThreadID threadDeleted,
   DebuggerDisposeThreadUPP userUPP
);
```

Parameters

userUPP

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

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

Declared In
Threads.h

InvokeDebuggerNewThreadUPP

```
void InvokeDebuggerNewThreadUPP (
   ThreadID threadCreated,
   DebuggerNewThreadUPP userUPP
);
```

Parameters

userUPP

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

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

Declared In
Threads.h

InvokeDebuggerThreadSchedulerUPP

```
ThreadID InvokeDebuggerThreadSchedulerUPP (
   SchedulerInfoRecPtr schedulerInfo,
   DebuggerThreadSchedulerUPP userUPP
);
```

Parameters

schedulerInfo

userUPP

Return Value
See the description of the `ThreadID` data type.

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

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

Declared In
`Threads.h`

InvokeThreadEntryUPP

```
voidPtr InvokeThreadEntryUPP (
   void *threadParam,
   ThreadEntryUPP userUPP
);
```

Parameters

userUPP

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

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

Declared In
`Threads.h`

InvokeThreadSchedulerUPP

```
ThreadID InvokeThreadSchedulerUPP (
   SchedulerInfoRecPtr schedulerInfo,
   ThreadSchedulerUPP userUPP
);
```

Parameters

schedulerInfo

userUPP

Return Value
See the description of the `ThreadID` data type.

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

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

Declared In
`Threads.h`

InvokeThreadSwitchUPP

```
void InvokeThreadSwitchUPP (
   ThreadID threadBeingSwitched,
   void *switchProcParam,
   ThreadSwitchUPP userUPP
);
```

Parameters

userUPP

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

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

Declared In
`Threads.h`

InvokeThreadTerminationUPP

```
void InvokeThreadTerminationUPP (
   ThreadID threadTerminated,
   void *terminationProcParam,
   ThreadTerminationUPP userUPP
);
```

Parameters

userUPP

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

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

Declared In
`Threads.h`

NewDebuggerDisposeThreadUPP

```
DebuggerDisposeThreadUPP NewDebuggerDisposeThreadUPP (
   DebuggerDisposeThreadProcPtr userRoutine
);
```

Parameters

userRoutine

Return Value
See the description of the `DebuggerDisposeThreadUPP` data type.

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

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

Declared In
`Threads.h`

NewDebuggerNewThreadUPP

```
DebuggerNewThreadUPP NewDebuggerNewThreadUPP (
   DebuggerNewThreadProcPtr userRoutine
);
```

Parameters

userRoutine

Return Value
See the description of the `DebuggerNewThreadUPP` data type.

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

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

Declared In
`Threads.h`

NewDebuggerThreadSchedulerUPP

```
DebuggerThreadSchedulerUPP NewDebuggerThreadSchedulerUPP (
   DebuggerThreadSchedulerProcPtr userRoutine
);
```

Parameters

userRoutine

Return Value
See the description of the `DebuggerThreadSchedulerUPP` data type.

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

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

Declared In
`Threads.h`

NewThread

Creates a new thread or allocates one from the existing pool of threads.

Modified

```
OSErr NewThread (
   ThreadStyle threadStyle,
   ThreadEntryTPP threadEntry,
   void *threadParam,
   Size stackSize,
   ThreadOptions options,
   void **threadResult,
   ThreadID *threadMade
);
```

Parameters

threadStyle

The type of thread to create. Cooperative is the only type that you can specify. Historically, the Thread Manger supported two types of threads, preemptive and cooperative, but the Thread Manager no longer supports preemptive threads.

threadEntry

A pointer to the thread entry function.

threadParam

A pointer to a value that the Thread Manager passes as a parameter to the thread entry function. Specify `NULL` if you are passing no information.

stackSize

The stack size (in bytes) to allocate for this thread. This stack must be large enough to handle saved thread context, normal application stack usage, interrupt handling functions, and CPU exceptions. Specify a stack size of 0 (zero) to request the Thread Manager's default stack size.

options

Options that define characteristics of the new thread. See the `Thread Option Constants` (page 52) data type for details on the options. You sum the options together to create a single `options` parameter.

threadResult

On return, a pointer to the address of a location to hold the function result provided by the `Thread Option Constants` (page 52) function when the thread terminates. Specify `NULL` for this parameter if you are not interested in the function result.

threadMade

On return, a pointer to the thread ID of the newly created or allocated thread. If there is an error, `threadMade` points to a value of `kNoThreadID`.

Return Value

A result code. See "Thread Manager Result Codes" (page 54).

Discussion

The `NewThread` function obtains a thread ID that you can use in other Thread Manager functions to identify the thread. If you want to allocate a thread from the pool of threads, specify the `kUsePremadeThread` option of the `options` parameter. Otherwise, `NewThread` creates a new thread.

When you request a thread from the existing pool, the Thread Manager allocates one that best fits your specified stack size. If you specify the `kExactMatchThread` option of the `options` parameter, the Thread Manager allocates a thread whose stack exactly matches your stack-size requirement or, if it can't allocate one because no such thread exists, it returns the `threadTooManyReqsErr` result code.

Before making any calls to `NewThread`, be certain that you first have called the Memory Manager function `MaxApplZone` to extend the application heap to its limit. You must call `MaxApplZone` from the main application thread before any other threads in your application run.

When you call the `NewThread` function, you pass, as the `threadEntry` parameter, a pointer to the name of the entry function to the thread. When the newly created thread runs initially, it begins by executing this function.

You can use the `threadParam` parameter to pass thread-specific information to a newly created or allocated thread. In the data structure pointed to by this parameter, you could place something like A5 information or the address of a window to update. You could also use this parameter to specify a place for a thread's local storage.

Be sure to create the storage for the `threadResult` parameter in a place that is guaranteed to be available when the thread terminates—for example, in an application global variable or in a local variable of the application's main function (the main thread, by definition, cannot be disposed of so it is always available). Do not create the storage in a local variable of a subfunction that completes before the thread terminates or the storage will become invalid.

For Carbon applications, the pointer to your thread entry function must be a universal procedure pointer (UPP).

To dispose of a thread, use the DisposeThread function.

See the description of the `Thread Option Constants` (page 52) data type for details on the characteristics you can specify in the `options` parameter.

For more information about the thread entry function, see the `ThreadEntryProcPtr` (page 42) function.

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

Carbon Porting Notes

For Carbon applications, you must create and pass a universal procedure pointer (UPP) to specify the new thread callback. Use the `NewThreadEntryUPP` (page 26) and `DisposeThreadEntryUPP` (page 14) functions to create and remove the UPP.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`Threads.h`

NewThreadEntryUPP

```
ThreadEntryUPP NewThreadEntryUPP (
   ThreadEntryProcPtr userRoutine
);
```

Parameters

`userRoutine`

Return Value

See the description of the `ThreadEntryUPP` data type.

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

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

Declared In
`Threads.h`

NewThreadSchedulerUPP

```
ThreadSchedulerUPP NewThreadSchedulerUPP (
   ThreadSchedulerProcPtr userRoutine
);
```

Parameters

userRoutine

Return Value
See the description of the `ThreadSchedulerUPP` data type.

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

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

Declared In
`Threads.h`

NewThreadSwitchUPP

```
ThreadSwitchUPP NewThreadSwitchUPP (
   ThreadSwitchProcPtr userRoutine
);
```

Parameters

userRoutine

Return Value
See the description of the `ThreadSwitchUPP` data type.

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

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

Declared In
`Threads.h`

NewThreadTerminationUPP

```
ThreadTerminationUPP NewThreadTerminationUPP (
   ThreadTerminationProcPtr userRoutine
);
```

Parameters

`userRoutine`

Return Value
See the description of the `ThreadTerminationUPP` data type.

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

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

Declared In
`Threads.h`

SetDebuggerNotificationProcs

Installs functions that notify the debugger when a thread is created, disposed of, or scheduled.

Modified

```
OSErr SetDebuggerNotificationProcs (
   DebuggerNewThreadTPP notifyNewThread,
   DebuggerDisposeThreadTPP notifyDisposeThread,
   DebuggerThreadSchedulerTPP notifyThreadScheduler
);
```

Parameters

`notifyNewThread`

A pointer to the callback function that notifies the debugger when a thread is created.

notifyDisposeThread

A pointer to the callback function that notifies the debugger when a thread is disposed of. This function is called whether you manually dispose of a thread with the `DisposeThread` function or if a thread disposes of itself automatically when it returns from its highest level of code.

notifyThreadScheduler

A pointer to the callback function that notifies the debugger when a thread is scheduled.

Return Value
A result code. See "Thread Manager Result Codes" (page 54).

Discussion
You generally use this function only during development of an application.

The `SetDebuggerNotificationProcs` function installs three separate callback functions that return the thread ID of a newly created thread, the thread ID of a newly disposed of thread, and the thread ID of a newly scheduled thread.

The `SetDebuggerNotificationProcs` function always installs all three of the debugging functions. You cannot set only one or two of these functions, nor can you chain them together. These restrictions ensure that the function that calls `SetDebuggerNotificationProcs` owns all three of the debugging functions. If you want to prevent one or two of these debugging functions from being called, you can do so by setting them to `NULL`.

To guarantee that the debugger is getting an accurate view of scheduling, the Thread Manager doesn't call the scheduling-notification callback function until both the generic Thread Manager scheduler and any custom thread scheduler have decided on a thread to schedule.

For Carbon applications, the pointers you pass to specify the callbacks must be universal procedure pointers (UPPs).

To create or allocate a new thread, use the `NewThread` (page 24) function.

To dispose of a thread, use the `DisposeThread` function.

To schedule a thread, you can use a yield function such as `YieldToAnyThread` (page 38) or `YieldToThread` (page 39) or a function to change the state of a thread, such as `SetThreadState` (page 32).

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

Carbon Porting Notes

For Carbon applications, you must create and pass a universal procedure pointer (UPP) to specify the notification callbacks. You must use the designated UPP creation and disposal functions. For example, for the new thread notifier, you call the `NewDebuggerNewThreadUPP` (page 24) and `DisposeDebuggerNewThreadUPP` (page 12) functions.

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

Declared In
`Threads.h`

SetThreadReadyGivenTaskRef

Changes the state of a thread from stopped to ready when your application is not the current process.

```
OSErr SetThreadReadyGivenTaskRef (
   ThreadTaskRef threadTRef,
   ThreadID threadToSet
);
```

Parameters

threadTRef

The thread task reference of the application containing the thread whose state you want to change.

threadToSet

The thread ID of the thread whose state you want to change.

Return Value

A result code. See "Thread Manager Result Codes" (page 54).

Discussion

When you mark a thread as ready to run with this function, the Thread Manager does not put it immediately into the scheduling queue but does so the next time it reschedules threads.

You can use `SetThreadStateGivenTaskRef` at times when you aren't guaranteed that your application is the current context, such as during execution of an interrupt function. In such cases you must identify the thread task reference (the application context) as well as the thread ID.

You obtain the thread task reference for your application with the `GetThreadCurrentTaskRef` (page 17) function.

The `SetThreadReadyGivenTaskRef` function allows you to do one thing only—change a thread from stopped to ready to execute. You cannot change the state of an executing thread to ready or stopped, nor can you change the state of a ready thread to executing or stopped with this call.

To determine the state of a thread when your application is not the current process, use the `GetThreadStateGivenTaskRef` (page 19) function.

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`Threads.h`

SetThreadScheduler

Installs a custom scheduling function (custom scheduler).

Modified

```
OSErr SetThreadScheduler (
   ThreadSchedulerTPP threadScheduler
);
```

Parameters

threadScheduler

A pointer to a custom scheduler. Specify `NULL` if you want to remove an installed custom scheduler and use the default Thread Manager scheduling mechanism.

Return Value

A result code. See "Thread Manager Result Codes" (page 54).

Discussion

The `SetThreadScheduler` function installs a custom scheduler that runs in conjunction with the default Thread Manager scheduling mechanism. The Thread Manager uses a scheduler information structure to pass the custom scheduler the ID of the current thread and the ID of the thread that the Thread Manager has scheduled to run next.

A custom scheduler should return to the Thread Manager the ID of the thread that it determines to schedule. If it does not determine a particular thread to schedule, it should return the constant `kNoThreadID` and the Thread Manager default scheduling mechanism schedules the next thread.

If you already have a custom scheduler installed when you call `SetThreadScheduler`, it replaces the old one with a new one. If you want to remove your custom scheduler and return to using the default Thread Manager scheduling mechanism, call `SetThreadScheduler` and specify a value of `NULL` for the parameter.

The `SetThreadScheduler` function automatically disables scheduling to avoid any reentrancy problems with the custom scheduling function. Therefore, in your custom scheduling function, you should make no yield calls or other calls that would cause scheduling to occur.

For Carbon applications, the pointer to your thread scheduler function must be a universal procedure pointer (UPP).

For more information on the custom scheduling function, see the `ThreadSchedulerProcPtr` (page 43) function.

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

Carbon Porting Notes

For Carbon applications, you must create and pass a universal procedure pointer (UPP) to specify the thread scheduler callback. Use the `NewThreadSchedulerUPP` (page 27) and `DisposeThreadSchedulerUPP` (page 15) functions to create and remove the UPP.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`Threads.h`

SetThreadState

Changes the state of any thread.

```
OSErr SetThreadState (
   ThreadID threadToSet,
   ThreadState newState,
   ThreadID suggestedThread
);
```

Parameters

threadToSet

The thread ID of the thread whose state is to be changed.

newState

The new state for the thread. You can specify ready to execute (`kReadyThreadState`), stopped (`kStoppedThreadState`), or executing (`kRunningThreadState`).

suggestedThread

The thread ID of the next thread to run. You specify this thread if you are changing the state of the currently executing thread to stopped or ready to run. Pass `kNoThreadID` if you do not want to specify a particular thread to run next. In this case, the Thread Manager schedules the next available thread to run.

Return Value

A result code. See "Thread Manager Result Codes" (page 54).

Discussion

The effect of `SetThreadState` depends on whether the thread you specify for changing is the currently executing thread or another thread. If you specify the current thread and thus change the state to stopped or ready, `SetThreadState` invokes the Thread Manager scheduling mechanism. The current thread relinquishes control (it is put in the state you specify, stopped or ready) and the Thread Manager schedules the thread that you specify with the `suggestedThread` parameter. If this thread is unavailable for running, or if you passed `kNoThreadID`, the Thread Manager schedules the next available thread.

If you change the state of the current thread to ready, the Thread Manager suspends it awaiting of the CPU. When it is rescheduled, `SetThreadState` regains control and returns to the function that called it.

If you have installed a custom scheduler, the Thread Manager passes it the thread ID of the suspended thread.

If you specify a thread other than the currently executing thread, no rescheduling occurs. If you change the state from ready to stopped, the thread is removed from the scheduling queue. The Thread Manager does not schedule this thread for execution again until you change its state to ready. On the other hand, if you change the state from stopped to ready, you have in effect put the thread in the scheduling queue, and the Thread Manager gives it CPU time as soon as it reaches the top of the scheduling queue.

Threads must yield in the CPU addressing mode (24-bit or 32-bit) in which the application was launched.

To obtain the state of any thread, use the `GetThreadState` (page 18) function.

To relinquish control to the next available thread, use the `YieldToAnyThread` (page 38) function. To relinquish control to a specific thread, use the `YieldToThread` (page 39) function.

To set the state of the current thread before it exits a critical section of code, use the `SetThreadStateEndCritical` (page 33) function.

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`Threads.h`

SetThreadStateEndCritical

Changes the state of the current thread and exits that thread's critical section at the same time.

```
OSErr SetThreadStateEndCritical (
   ThreadID threadToSet,
   ThreadState newState,
   ThreadID suggestedThread
);
```

Parameters

threadToSet

The thread ID of the thread whose state is to be changed.

newState

The new state for the thread. You can specify ready to execute (`kReadyThreadState`), stopped (`kStoppedThreadState`) or executing (`kRunningThreadState`).

suggestedThread

The thread ID of the next thread to run. You specify this thread if you are changing the state of the currently executing thread to stopped or ready to run. Pass `kNoThreadID` if you do not want to specify a particular thread to run next. In this case, the Thread Manager schedules the next available thread to run.

Return Value

A result code. See "Thread Manager Result Codes" (page 54).

Discussion

The `SetThreadStateEndCritical` function does in one step the same thing that `ThreadEndCritical` and `SetThreadState` functions do in two steps.

Historically, the primary purpose of the `SetThreadStateEndCritical` function was to close the scheduling window at the end of a critical section. A preemptive thread that was waiting while the critical section of code was executing could begin executing before you changed the state of the current thread to stopped with the `SetThreadState` function. Obviously, because the Thread Manager no longer supports preemptive threads, this function is no longer necessary to close the scheduling window, but you can still use it to change the state of a thread and exit a critical section in one step instead of two.

When you change the state of the currently executing thread, the Thread Manager schedules the thread you specify with the `suggestedThread` parameter. If this thread is unavailable or if you pass `kNoThreadID`, the Thread Manager schedules the next available thread.

To mark a section of code as critical, use the `ThreadBeginCritical` (page 36) and the `ThreadEndCritical` (page 38) functions.

To change the state of any thread, use the `SetThreadState` (page 32) function.

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`Threads.h`

SetThreadSwitcher

Installs a custom context-switching function for any thread.

Modified

```
OSErr SetThreadSwitcher (
   ThreadID thread,
   ThreadSwitchTPP threadSwitcher,
   void *switchProcParam,
   Boolean inOrOut
);
```

Parameters

thread

The thread ID of the thread to associate with a context-switching function.

threadSwitcher

A pointer to the context-switching function.

switchProcParam

A pointer to a thread-specific parameter that you pass to the context-switching function.

inOrOut

A Boolean value that indicates whether the Thread Manager calls the context-switching function when the specified thread switches in (`True`) or when it is switched out by another thread (`False`).

Return Value

A result code. See "Thread Manager Result Codes" (page 54).

Discussion

The custom switching function allows you to save context information in addition to the default context information that the Thread Manager automatically saves when it switches contexts. The default context information consists of the CPU registers, the FPU registers (if any), and the location of the thread's context.

You must actually define two context-switching functions, one for leaving a thread and another for entering a thread. When leaving a thread, you call the outer context-switching function to save additional context information. When reentering a thread, you call the inner context-switching function to restore the extra information that was saved on exit. Use the `inOrOut` parameter of the `SetThreadSwitcher` function to specify which type of context-switching function is being installed.

You can pass a different `switchProcParam` parameter to each thread, which allows you to write a single, application-wide custom switching function and then pass any thread-specific information when the Thread Manager calls the switching function for that thread.

The `SetThreadSwitcher` function automatically disables scheduling to avoid any reentrancy problems with the custom switching function. Therefore, in the custom switching function, you should make no yield calls or other calls that would cause scheduling to occur.

For Carbon applications, the pointer to your thread switcher function must be a universal procedure pointer (UPP).

For more information on the custom context-switching function, see the `ThreadSwitchProcPtr` (page 44) function.

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

Carbon Porting Notes

For Carbon applications, you must create and pass a universal procedure pointer (UPP) to specify the thread switcher callback. Use the `NewThreadSwitchUPP` (page 27) and `DisposeThreadSwitchUPP` (page 15) functions to create and remove the UPP.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`Threads.h`

SetThreadTerminator

Installs a custom thread-termination function for a thread.

Modified

```
OSErr SetThreadTerminator (
   ThreadID thread,
   ThreadTerminationTPP threadTerminator,
   void *terminationProcParam
);
```

Parameters

thread

The thread ID of the thread to associate with the thread-termination function.

threadTerminator
A pointer to the thread-termination function.

terminationProcParam
A pointer to a thread-specific parameter that you pass to the thread-termination function.

Return Value
A result code. See "Thread Manager Result Codes" (page 54).

Discussion
The Thread Manager calls the custom termination function whenever the specified thread completes execution of its code or when you manually dispose of the thread with the `DisposeThread` (page 13) function.

You can pass a different `terminationProcParam` parameter to each thread, which allows you to write a single, application-wide custom thread-termination function and then pass any thread-specific information when the Thread Manager calls the termination function for that thread.

For Carbon applications, the pointer to your thread terminator function must be a universal procedure pointer (UPP).

For more information on the custom thread-termination function, see the `ThreadTerminationProcPtr` (page 45) function.

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

Carbon Porting Notes

For Carbon applications, you must create and pass a universal procedure pointer (UPP) to specify the thread terminator callback. Use the `NewThreadTerminationUPP` (page 28) and `DisposeThreadTerminationUPP` (page 16) functions to create and remove the UPP.

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

Declared In
`Threads.h`

ThreadBeginCritical

Indicates that the thread is entering a critical code section.

```
OSErr ThreadBeginCritical (
   void
);
```

Return Value
A result code. See "Thread Manager Result Codes" (page 54).

Discussion
The `ThreadBeginCritical` function disables scheduling by marking the beginning of a section of critical code. That is, no other threads in the current application can run—even if the current thread yields control—until the current thread exits the critical section (by calling the `ThreadEndCritical` function). Disabling scheduling allows the currently executing function to look at or change shared or global data safely. You can nest critical sections within a thread.

To mark the end of a critical code section and turn scheduling back on, use the `ThreadEndCritical` (page 38) function. If you also need to set the state of the current thread before scheduling is turned back on, use the `SetThreadStateEndCritical` (page 33) function.

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

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

Declared In
`Threads.h`

ThreadCurrentStackSpace

Determines the amount of stack space that is available for any thread in your application.

```
OSErr ThreadCurrentStackSpace (
   ThreadID thread,
   ByteCount *freeStack
);
```

Parameters

thread
The thread ID of the thread about which you want information.

freeStack
On return, a pointer to the amount of stack space (in bytes) that is available to the specified thread.

Return Value
A result code. See "Thread Manager Result Codes" (page 54).

Discussion
This function is primarily useful during debugging since you determine the maximum amount of stack space you need for any particular thread before you ship your application. However, if your application calls a recursive function that could call itself many times, you might want to use `ThreadCurrentStackSpace` to keep track of the stack space and take appropriate action if it becomes too low.

To determine the default size that the Thread Manager assigns to threads use the `GetDefaultThreadStackSize` (page 17) function.

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

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

Declared In
`Threads.h`

ThreadEndCritical

Indicates that the thread is leaving a critical code section.

```
OSErr ThreadEndCritical (
   void
);
```

Return Value
A result code. See "Thread Manager Result Codes" (page 54).

Discussion
After a call to the Thread, all scheduling operations are now available to the application.

Use the `ThreadBeginCritical` (page 36) function to mark the beginning of a critical code section and turn scheduling off.

If you need to set the state of the current thread before scheduling is turned back on, use the `SetThreadStateEndCritical` (page 33) function.

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

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

Declared In
`Threads.h`

YieldToAnyThread

Relinquishes the current thread's control.

```
OSErr YieldToAnyThread (
   void
);
```

Return Value
A result code. See "Thread Manager Result Codes" (page 54).

Discussion
The `YieldToAnyThread` function invokes the Thread Manager's scheduling mechanism. The current thread relinquishes control and the Thread Manager schedules the next available thread.

The current thread is suspended in the ready state and awaits rescheduling when the CPU is available. When the suspended thread is scheduled again, `YieldToAnyThread` regains control and returns to the function that called it.

If you have installed a custom scheduler, the Thread Manager passes it the thread ID of the suspended thread.

In each thread you must make one or more strategically placed calls to relinquish control to another thread. You can either make this yield call or another yield call such as `YieldToThread`; or you can make a call such as `SetThreadState` to explicitly change the state of the thread.

Threads must yield in the CPU addressing mode (24-bit or 32-bit) in which the application was launched.

To relinquish control to a specific thread, use the `YieldToThread` (page 39) function.

To change the state of a specified thread, use the `SetThreadState` (page 32) function.

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

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

Declared In
`Threads.h`

YieldToThread

Relinquishes the current thread's control to a particular thread.

```
OSErr YieldToThread (
   ThreadID suggestedThread
);
```

Parameters

`suggestedThread`
The ID of the thread to yield control to.

Return Value
A result code. See "Thread Manager Result Codes" (page 54).

Discussion
The `YieldToThread` function invokes the Thread Manager's scheduling mechanism. The current thread relinquishes control and passes the thread ID of a thread for the Thread Manager to schedule. The Thread Manager schedules this thread if it is available. Otherwise, the Thread Manager schedules the next available thread.

The current thread is suspended in the ready state and awaits rescheduling when the CPU is available. When the suspended thread is scheduled again, `YieldToThread` regains control and returns to the function that called it.

If you have installed a custom scheduler, the Thread Manager passes it the thread ID of the suspended thread.

In each thread you must make one or more strategically placed calls to relinquish control to another thread. You can either make this yield call or another yield call such as `YieldToAnyThread`; or you can make a call such as `SetThreadState` to explicitly change the state of the thread.

Threads must yield in the CPU addressing mode (24-bit or 32-bit) in which the application was launched.

To relinquish control without naming a specific thread, use the `YieldToAnyThread` (page 38) function.

To change the state of a specified thread, use the `SetThreadState` (page 32) function.

Special Considerations
Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

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

Declared In
`Threads.h`

Callbacks

DebuggerDisposeThreadProcPtr

Defines a pointer to a dispose thread debugging callback function. A dispose thread debugging callback function is a debugging function that the Thread Manager calls whenever it disposes of a thread.

```
typedef void (*DebuggerDisposeThreadProcPtr)
(
    ThreadID threadDeleted
);
```

If you name your function `MyDebuggerDisposeThreadProc`, you would declare it like this:

```
void MyDebuggerDisposeThreadProcPtr (
    ThreadID threadDeleted
);
```

Parameters

threadDeleted

The thread ID of the thread being disposed of.

Return Value

Discussion

The `MyDebuggerDisposeThreadCallback` function is one of three debugging functions that you can install with the `SetDebuggerNotificationProcs` (page 28) function. The Thread Manager calls `MyDebuggerDisposeThreadCallback` whenever an application disposes of a thread. The thread manager calls this debugging function whether you manually call `DisposeThread` (page 13) to dispose of a thread or if a thread finishes executing its code and the Thread Manager automatically disposes of it.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`Threads.h`

DebuggerNewThreadProcPtr

Defines a pointer to a new thread debugging callback function. A new thread debugging callback function is a debugging function that the Thread Manager calls whenever it creates a new thread.

```
typedef void (*DebuggerNewThreadProcPtr)
(
    ThreadID threadCreated
);
```

If you name your function `MyDebuggerNewThreadProc`, you would declare it like this:

```
void MyDebuggerNewThreadProcPtr (
    ThreadID threadCreated
);
```

Parameters

threadCreated

The thread ID of the thread being created.

Return Value

Discussion

The `MyDebuggerNewThreadCallback` function is one of three debugging functions that you can install with the `SetDebuggerNotificationProcs` (page 28) function. The Thread Manager calls `MyDebuggerNewThreadCallback` whenever an application creates or allocates a new thread with the `NewThread` (page 24) function. The Thread Manager does not call `MyDebuggerNewThreadCallback` when an application creates a thread pool with the `CreateThreadPool` function.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`Threads.h`

DebuggerThreadSchedulerProcPtr

Defines a pointer to a thread scheduler debugging callback function. A thread scheduler debugging callback function is a debugging function that the Thread Manager calls whenever a thread is scheduled.

```
typedef ThreadID (*DebuggerThreadSchedulerProcPtr)
(
    SchedulerInfoRecPtr schedulerInfo
);
```

If you name your function `MyDebuggerThreadSchedulerProc`, you would declare it like this:

```
ThreadID MyDebuggerThreadSchedulerProcPtr
(
    SchedulerInfoRecPtr schedulerInfo
);
```

Parameters

schedulerInfo

A pointer to a scheduler information structure that the `SetDebuggerNotificationProcs` function passes to the `MyDebuggerThreadSchedulerCallback` function. Among other information, the scheduler information structure contains the ID of the current thread and the ID of the thread that the Thread Manager has scheduled to run next.

Return Value

See the description of the `ThreadID` data type.

Discussion

The `MyDebuggerThreadSchedulerCallback` function is one of three debugging functions that you can install with the `SetDebuggerNotificationProcs` (page 28) function. The Thread Manager calls `MyDebuggerThreadSchedulerCallback` whenever an application schedules a new thread to run. The `MyDebuggerThreadSchedulerCallback` function gets the last look at the thread being scheduled—that is, the Thread Manager calls this function after the Thread Manager default scheduling mechanism and a custom scheduler, if you have installed one, decide on the next thread to schedule.

If you wish, you can use this debugging callback function to schedule a different thread than that chosen by the Thread Manager and any custom scheduling function. The `MyDebuggerThreadSchedulerCallback` returns the thread ID of the next thread to schedule. The `MyDebuggerThreadSchedulerCallback` can specify `kNoThreadID` for the thread ID if you do not want to change the decision of the Thread Manager default scheduler or a custom scheduler.

To schedule a thread, use functions such as `YieldToAnyThread` (page 38) , `YieldToThread` (page 39) , and `SetThreadState` (page 32).

Availability

Available in Mac OS X v10.0 and later.

Declared In

`Threads.h`

ThreadEntryProcPtr

Defines a pointer to a thread entry callback function. Your thread entry callback function provides an entry point to a thread that you create in your application.

```
typedef voidPtr (*ThreadEntryProcPtr)
(
    void * threadParam
);
```

If you name your function `MyThreadEntryProc`, you would declare it like this:

```
voidPtr MyThreadEntryProcPtr (
    void * threadParam
);
```

Parameters

threadParam

A pointer to a `void` data structure passed to this function by the `NewThread` function.

Return Value

Discussion

When you create or allocate a new thread with the `NewThread` function, you pass the name of this entry function. You also pass a parameter that the Thread Manager passes on to the `MyThreadEntryCallback` function. You can use this parameter to pass thread-specific information to the newly created or allocated thread. For example, you could pass something like A5 information or the address of a window to update. Or you could use this parameter to specify local storage for a thread that other threads could access.

When the code in a thread finishes executing, the Thread Manager automatically calls the `DisposeThread` (page 13) function to dispose of the thread. The `MyThreadEntryCallback` function passes its function result to `DisposeThread`. The `DisposeThread` function passes this result back to the `NewThread` function that called `MyThreadEntryCallback` to begin with.

This mechanism allows you to spawn a thread that does some work and then continue with your original thread. When the spawned thread is finished doing its work—for example a calculation—it returns the result to the original thread.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`Threads.h`

ThreadSchedulerProcPtr

Defines a pointer to a thread scheduler callback function. Your thread scheduler callback function supplements the Thread Manager default scheduling mechanism.

```
typedef ThreadID (*ThreadSchedulerProcPtr)
(
    SchedulerInfoRecPtr schedulerInfo
);
```

If you name your function `MyThreadSchedulerProc`, you would declare it like this:

```
ThreadID MyThreadSchedulerProcPtr (
    SchedulerInfoRecPtr schedulerInfo
);
```

Parameters

schedulerInfo

A pointer to the scheduler information structure that the Thread Manager uses to pass information to `MyThreadSchedulerCallback`.

Return Value

See the description of the `ThreadID` data type.

Discussion

The `MyThreadSchedulerCallback` function does not supplant the Thread Manager scheduling mechanism but rather works in conjunction with it.

Whenever scheduling occurs, the Thread Manager passes a scheduler information structure to `MyThreadSchedulerCallback`. Among other information, the scheduler information structure contains the thread ID of the current thread and the thread ID of the thread that the application has scheduled to run next.

The `MyThreadSchedulerCallback` function returns to the Thread Manager the thread ID of the thread that it has chosen to schedule and the Thread Manager does the actual scheduling. If `MyThreadSchedulerCallback` decides not to schedule a thread, it returns the constant `kNoThreadID` and the Thread Manager default scheduling mechanism schedules the next thread.

When the `SetThreadScheduler` function installs the custom scheduler, it automatically disables scheduling to avoid any reentrancy problems. Therefore, in the custom scheduler, you should make no yield calls or other calls that would cause scheduling to occur.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`Threads.h`

ThreadSwitchProcPtr

Defines a pointer to a thread switch callback function. Your thread switch callback function adds to the thread context information that the Thread Manager saves and restores.

```
typedef void (*ThreadSwitchProcPtr) (
    ThreadID threadBeingSwitched,
    void * switchProcParam
);
```

If you name your function `MyThreadSwitchProc`, you would declare it like this:

```
void MyThreadSwitchProcPtr (
    ThreadID threadBeingSwitched,
    void * switchProcParam
);
```

Parameters

threadBeingSwitched

The thread ID of the thread whose context is being switched.

switchProcParam

A pointer to a parameter that the `SetThreadSwitcher` function passes to `MyThreadSwitchCallback`.

Return Value

Discussion

The custom switching function allows you to save and restore context information in addition to the default context information that the Thread Manager automatically saves and restores when it switches contexts. You must actually define two context-switching functions, one for leaving a thread and another for entering a thread. When leaving a thread, you call the outer context-switching function to save additional context information. When reentering a thread, you call the inner context-switching function to restore the extra information that was saved on exit.

The default context information consists of the CPU registers, the FPU registers (if any), and the location of the thread's context.

When the `SetThreadSwitcher` function installs the custom switching function, it automatically disables scheduling to avoid any reentrancy problems. Therefore, in the custom switching function, you should make no yield calls or other calls that would cause scheduling to occur.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`Threads.h`

ThreadTerminationProcPtr

Defines a pointer to a thread termination callback function. Your thread termination callback function does additional cleanup when the code in a thread finishes executing.

```
typedef void (*ThreadTerminationProcPtr)
(
    ThreadID threadTerminated,
    void * terminationProcParam
);
```

If you name your function `MyThreadTerminationProc`, you would declare it like this:

```
void MyThreadTerminationProcPtr (
    ThreadID threadTerminated,
    void * terminationProcParam
);
```

Parameters

threadTerminated

The thread ID of the thread being disposed of.

terminationProcParam

A pointer to a `void` data structure that the `SetThreadTerminator` function passes to `MyThreadTerminationCallback`.

Return Value

Discussion
You use the `SetThreadTerminator` function to install the `MyThreadTerminationCallback` custom termination function. The custom termination function allows you to do additional cleanup when the code in a thread finishes executing or when you call the `DisposeThread` (page 13) function to manually dispose of a thread.

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

Declared In
`Threads.h`

Data Types

DebuggerDisposeThreadUPP

```
typedef DebuggerDisposeThreadProcPtr DebuggerDisposeThreadUPP;
```

Discussion
For more information, see the description of the DebuggerDisposeThreadUPP () callback function.

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

Declared In
`Threads.h`

DebuggerDisposeThreadTPP

```
typedef DebuggerDisposeThreadUPP DebuggerDisposeThreadTPP;
```

Discussion

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

Declared In
`Threads.h`

DebuggerNewThreadTPP

```
typedef DebuggerNewThreadUPP DebuggerNewThreadTPP;
```

Discussion

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

Declared In
Threads.h

DebuggerNewThreadUPP

```
typedef DebuggerNewThreadProcPtr DebuggerNewThreadUPP;
```

Discussion
For more information, see the description of the DebuggerNewThreadUPP () callback function.

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

Declared In
Threads.h

DebuggerThreadSchedulerUPP

```
typedef DebuggerThreadSchedulerProcPtr DebuggerThreadSchedulerUPP;
```

Discussion
For more information, see the description of the DebuggerThreadSchedulerUPP () callback function.

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

Declared In
Threads.h

DebuggerThreadSchedulerTPP

```
typedef DebuggerThreadSchedulerUPP DebuggerThreadSchedulerTPP;
```

Discussion

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

Declared In
Threads.h

SchedulerInfoRec

```
struct SchedulerInfoRec {
    UInt32 InfoRecSize;
    ThreadID CurrentThreadID;
    ThreadID SuggestedThreadID;
    ThreadID InterruptedCoopThreadID;
};
typedef struct SchedulerInfoRec SchedulerInfoRec;
typedef SchedulerInfoRec * SchedulerInfoRecPtr;
```

Fields

`InfoRecSize`
The size of the structure.

`CurrentThreadID`
The thread ID of the current thread.

`SuggestedThreadID`
The thread ID of the thread that the application has suggested to run.

`InterruptedCoopThreadID`
Historically, the thread ID of a preempted cooperative thread if a cooperative thread has been interrupted and has not yet resumed execution. Because it no longer supports preemptive threads, the Thread Manager always passes the constant `kNoThreadID` to indicate that there is no thread that has been interrupted.

Discussion

You can, if you wish, use the `SetThreadScheduler` (page 30) function to install a custom scheduling function to work in conjunction with the default Thread Manager scheduling mechanism. The Thread Manager uses the scheduler information structure to pass information to the custom scheduling function that allows it to decide which thread, if any, to schedule next.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`Threads.h`

ThreadEntryTPP

```
typedef ThreadEntryUPP ThreadEntryTPP;
```

Discussion

Availability

Available in Mac OS X v10.0 and later.

Declared In

`Threads.h`

ThreadEntryUPP

```
typedef ThreadEntryProcPtr ThreadEntryUPP;
```

Discussion
For more information, see the description of the ThreadEntryUPP () callback function.

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

Declared In
`Threads.h`

ThreadSchedulerTPP

```
typedef ThreadSchedulerUPP ThreadSchedulerTPP;
```

Discussion

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

Declared In
`Threads.h`

ThreadSchedulerUPP

```
typedef ThreadSchedulerProcPtr ThreadSchedulerUPP;
```

Discussion
For more information, see the description of the ThreadSchedulerUPP () callback function.

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

Declared In
`Threads.h`

ThreadSwitchTPP

```
typedef ThreadSwitchUPP ThreadSwitchTPP;
```

Discussion

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

Declared In
`Threads.h`

ThreadSwitchUPP

```
typedef ThreadSwitchProcPtr ThreadSwitchUPP;
```

Discussion
For more information, see the description of the ThreadSwitchUPP () callback function.

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

Declared In
`Threads.h`

ThreadTaskRef

Represents a thread task reference.

```
typedef void* ThreadTaskRef;
```

Discussion
In certain cases, such as during execution of an interrupt function, your application is not guaranteed to be the current process. Since threads are defined within an application context, it follows that in cases such as these, you cannot get or set information about any particular threads in your application unless you have a way of identifying the application context. The thread task reference gives you a way of doing this.

You can obtain the thread task reference by calling `GetThreadCurrentTaskRef` (page 17) at a time when you know your application is the current context. Later, during execution of an interrupt function, you can use the thread task reference to identify your application. For example, you can pass the thread task reference to functions such as `GetThreadStateGivenTaskRef` (page 19) and `SetThreadReadyGivenTaskRef` (page 30) in an interrupt function to get and set information about the state of particular threads in your application.

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

Declared In
`Threads.h`

ThreadTerminationTPP

```
typedef ThreadTerminationUPP ThreadTerminationTPP;
```

Discussion

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

Declared In
`Threads.h`

ThreadTerminationUPP

```
typedef ThreadTerminationProcPtr ThreadTerminationUPP;
```

Discussion
For more information, see the description of the ThreadTerminationUPP () callback function.

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

Declared In
`Threads.h`

Constants

Thread ID Constants

The `ThreadID` data type defines the thread ID.

```
typedef UInt32 ThreadID;
enum {
    kNoThreadID = 0,
    kCurrentThreadID = 1,
    kApplicationThreadID = 2
};
```

Constants

`kNoThreadID`
Indicates no thread; for example, you can use a function such as `SetThreadState` (page 32) to put the current thread in the stopped state and pass `kNoThreadID` to indicate that you don't care which thread runs next.

Available in Mac OS X v10.0 and later.

Declared in `Threads.h`.

`kCurrentThreadID`
Identifies the currently executing thread.

Available in Mac OS X v10.0 and later.

Declared in `Threads.h`.

`kApplicationThreadID`
Identifies the main application thread this is the cooperative thread that the Thread Manager creates at launch time. You cannot dispose of this thread. All applications—even those that are not aware of the Thread Manager—have one main application thread. The Thread Manager assumes that the main application thread is responsible for event gathering when an operating-system event occurs, the Thread Manager schedules the main application thread as the next thread to execute.

Available in Mac OS X v10.0 and later.

Declared in `Threads.h`.

Discussion
The Thread Manager assigns a thread ID to each thread that you create or allocate with the `NewThread` (page 24) function. The thread ID uniquely identifies a thread within an application context. You can use the thread ID in functions that schedule execution of a particular thread, dispose of a thread, and get and set information about a thread; for example, you pass the thread ID to functions such as `YieldToThread` (page 39) , `DisposeThread` (page 13) , and `GetThreadState` (page 18) .

In addition to the specific thread IDs that the `NewThread` function returns, you can use the three Thread Manager constants described here.

Thread Option Constants

```
typedef UInt32 ThreadOptions;
enum {
    kNewSuspend = (1 << 0),
    kUsePremadeThread = (1 << 1),
    kCreateIfNeeded = (1 << 2),
    kFPUNotNeeded = (1 << 3),
    kExactMatchThread = (1 << 4)
};
```

Constants

`kNewSuspend`
Begin a new thread in the stopped state.
Available in Mac OS X v10.0 and later.
Declared in `Threads.h`.

`kUsePremadeThread`
Use a thread from the existing supply.
Available in Mac OS X v10.0 and later.
Declared in `Threads.h`.

`kCreateIfNeeded`
Create a new thread if one with the proper style and stack size requirements does not exist.
Available in Mac OS X v10.0 and later.
Declared in `Threads.h`.

`kFPUNotNeeded`
Do not save the FPU context. This saves time when switching contexts. Note, however, that for PowerPC threads, the Thread Manager always saves the FPU registers regardless of how you set this option. Because the PowerPC microprocessor uses the FPU registers for optimizations, they could contain necessary information.
Available in Mac OS X v10.0 and later.
Declared in `Threads.h`.

`kExactMatchThread`
Allocate a thread from the pool only if it exactly matches the stack-size request. Without this option, a thread is allocated that best fits the request—that is, a thread whose stack is greater than or equal to the requested size.
Available in Mac OS X v10.0 and later.
Declared in `Threads.h`.

Discussion
When you create or allocate a new thread with the `NewThread` (page 24) function, you can specify thread options that define certain characteristics of the thread, using the values described here. To specify more than one option, you sum them together and pass them as a single parameter to the `NewThread` function.

The `ThreadOptions` data type defines the thread options.

Thread State Constants

```
typedef UInt16 ThreadState;
enum {
    kReadyThreadState = 0,
    kStoppedThreadState = 1,
    kRunningThreadState = 2
};
```

Constants
`kReadyThreadState`
The thread is ready to run.
Available in Mac OS X v10.0 and later.
Declared in `Threads.h`.

`kStoppedThreadState`
The thread is stopped and not ready to run.
Available in Mac OS X v10.0 and later.
Declared in `Threads.h`.

`kRunningThreadState`
The thread is running.
Available in Mac OS X v10.0 and later.
Declared in `Threads.h`.

Discussion
The Thread Manager functions which get and set information about the state of a thread, such as `GetThreadState` (page 18) and `SetThreadState` (page 32) , use these values.

Thread Style Constants

```
typedef UInt32 ThreadStyle;
enum {
    kCooperativeThread = 1L << 0,
    kPreemptiveThread = 1L << 1
};
```

Constants
`kCooperativeThread`
Available in Mac OS X v10.0 and later.
Declared in `Threads.h`.

`kPreemptiveThread`
Available in Mac OS X v10.0 and later.
Declared in `Threads.h`.

Discussion
Historically, the Thread Manager defined two types of threads to run in an application context: cooperative and preemptive, but now it supports only cooperative threads.

Although the Thread Manager only supports a single type of thread, many Thread Manager functions (for historical reasons) require you to use the thread type to specify the type of the thread.

The `ThreadStyle` data type specifies the type of a thread.

Because there is only one type of thread (cooperative) the thread type accepts a single value, `kCooperativeThread`.

Result Codes

The most common result codes returned by Thread Manager are listed below.

Result Code	Value	Description
`threadTooManyReqsErr`	-617	Available in Mac OS X v10.0 and later.
`threadNotFoundErr`	-618	Available in Mac OS X v10.0 and later.
`threadProtocolErr`	-619	Available in Mac OS X v10.0 and later.

Gestalt Constants

You can check for version and feature availability information by using the Thread Manager selectors defined in the Gestalt Manager. For more information see Inside Mac OS X: Gestalt Manager Reference.

Deprecated Thread Manager Functions

A function identified as deprecated has been superseded and may become unsupported in the future.

Deprecated in Mac OS X v10.3

GetFreeThreadCount

Determines how many threads are available to be allocated in a thread pool. (Deprecated in Mac OS X v10.3. There is no replacement.)

```
OSErr GetFreeThreadCount (
   ThreadStyle threadStyle,
   SInt16 *freeCount
);
```

Parameters

threadStyle

The type of thread to get information about. Cooperative is the only type that you can specify. Historically, the Thread Manger supported two types of threads, preemptive and cooperative, but the Thread Manager no longer supports preemptive threads.

freeCount

On return, a pointer to the number of threads available to be allocated.

Return Value

A result code. See "Thread Manager Result Codes" (page 54).

Discussion

The number of threads in the pool varies throughout execution of your application. Calls to `CreateThreadPool` add threads to the pool and calls to the function `NewThread` (page 24) , when an existing thread is allocated, reduce the number of threads. You also add threads to the pool when you dispose of a thread with the `DisposeThread` (page 13) function and specify that the thread be recycled.

Use the `GetSpecificFreeThreadCount` (page 56) function to determine how many threads of a particular stack size are available.

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.3.

Not available to 64-bit applications.

Declared In
`Threads.h`

GetSpecificFreeThreadCount

Determines how many threads with a stack size equal to or greater than the specified size are available to be allocated in a thread pool. (Deprecated in Mac OS X v10.3. There is no replacement.)

```
OSErr GetSpecificFreeThreadCount (
   ThreadStyle threadStyle,
   Size stackSize,
   SInt16 *freeCount
);
```

Parameters

threadStyle
The type of thread to get information about. Cooperative is the only type that you can specify. Historically, the Thread Manger supported two types of threads, preemptive and cooperative, but the Thread Manager no longer supports preemptive threads.

stackSize
The stack size of the threads to get information about.

freeCount
On return, a pointer to the number of threads of the specified stack size available to be allocated.

Return Value
A result code. See "Thread Manager Result Codes" (page 54).

Discussion
The `GetSpecificFreeThreadCount` function determines how many threads with a stack size equal to or greater than the specified size are available to be allocated. Use this function instead of `GetFreeThreadCount` (page 55) when you are interested not simply in the total number of available threads but when you want to know the number of available threads of a specified stack size as well.

The number of threads in the pool varies throughout execution of your application. Calls to the function `CreateThreadPool` (page 10) add threads to the pool and calls to the function `NewThread` (page 24) , when an existing thread is allocated, reduce the number of threads. You also add threads to the pool when you dispose of a thread with the `DisposeThread` (page 13) function and specify that the thread be recycled.

To determine how many threads of any stack size are available, use the `GetFreeThreadCount` function.

Special Considerations

Active development with the Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead. See *Threading Programming Guide* for more information.

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.3.
Not available to 64-bit applications.

Declared In
`Threads.h`

Document Revision History

This table describes the changes to *Thread Manager Reference*.

Date	Notes
2007-04-04	Updated for Mac OS X v10.5.
2006-07-24	Deprecated GetFreeThreadCount and GetSpecificFreeThreadCount. Added notes recommending against the use of the Thread Manager in new application development.
2003-01-01	Updated formatting and linking.

Document Revision History

Index

S

T

Y