Grand Central Dispatch (GCD) Reference

Developer

Contents

Grand Central Dispatch (GCD) Reference

Companion guide	Concurrency Programming Guide
Declared in	clock_types.h

Overview

Grand Central Dispatch (GCD) comprises language features, runtime libraries, and system enhancements that provide systemic, comprehensive improvements to the support for concurrent code execution on multicore hardware in iOS and OS X.

The BSD subsystem, CoreFoundation, and Cocoa APIs have all been extended to use these enhancements to help both the system and your application to run faster, more efficiently, and with improved responsiveness. Consider how difficult it is for a single application to use multiple cores effectively, let alone doing it on different computers with different numbers of computing cores or in an environment with multiple applications competing for those cores. GCD, operating at the system level, can better accommodate the needs of all running applications, matching them to the available system resources in a balanced fashion.

This document describes the GCD API, which supports the asynchronous execution of operations at the Unix level of the system. You can use this API to manage interactions with file descriptors, Mach ports, signals, or timers. In OS X v10.7 and later, you can also use GCD to handle general purpose asynchronous I/O operations on file descriptors.

GCD is not restricted to system-level applications, but before you use it for higher-level applications, you should consider whether similar functionality provided in Cocoa (via `NSOperation` and block objects) would be easier to use or more appropriate for your needs. See *Concurrency Programming Guide* for more information.

Functions by Task

Creating and Managing Queues

`dispatch_get_global_queue` (page 30)
Returns a well-known global concurrent queue of a given priority level.

`dispatch_get_main_queue` (page 31)
: Returns the serial dispatch queue associated with the application's main thread.

`dispatch_queue_create` (page 48)
: Creates a new dispatch queue to which blocks can be submitted.

`dispatch_get_current_queue` (page 29)
: Returns the queue on which the currently executing block is running.

`dispatch_queue_get_label` (page 49)
: Returns the label specified for the queue when the queue was created.

`dispatch_set_target_queue` (page 57)
: Sets the target queue for the given object.

`dispatch_main` (page 47)
: Executes blocks submitted to the main queue.

Queuing Tasks for Dispatch

GCD provides and manages FIFO queues to which your application can submit tasks in the form of block objects. Blocks submitted to dispatch queues are executed on a pool of threads fully managed by the system. No guarantee is made as to the thread on which a task executes. GCD offers three kinds of queues:

- **Main:** tasks execute serially on your application's main thread
- **Concurrent:** tasks are dequeued in FIFO order, but run concurrently and can finish in any order.
- **Serial:** tasks execute one at a time in FIFO order

The main queue is automatically created by the system and associated with your application's main thread. Your application uses one (and only one) of the following three approaches to invoke blocks submitted to the main queue:

- Calling `dispatch_main` (page 47)
- Calling `UIApplicationMain` (iOS) or `NSApplicationMain` (OS X)
- Using a `CFRunLoopRef` on the main thread

Use concurrent queues to execute large numbers of tasks concurrently. GCD automatically creates three concurrent dispatch queues that are global to your application and are differentiated only by their priority level. Your application requests these queues using the `dispatch_get_global_queue` (page 30) function. Because these concurrent queues are global to your application, you do not need to retain and release them; retain and release calls for them are ignored. In OS X v10.7 and later, you can also create additional concurrent queues for use in your own code modules.

Use serial queues to ensure that tasks to execute in a predictable order. It's a good practice to identify a specific purpose for each serial queue, such as protecting a resource or synchronizing key processes. Your application must explicitly create and manage serial queues. It can create as many of them as necessary, but should avoid using them instead of concurrent queues just to execute many tasks simultaneously.

Important: GCD is a C level API; it does not catch exceptions generated by higher level languages. Your application must catch all exceptions before returning from a block submitted to a dispatch queue.

`dispatch_async` (page 17)
Submits a block for asynchronous execution on a dispatch queue and returns immediately.

`dispatch_async_f` (page 18)
Submits an application-defined function for asynchronous execution on a dispatch queue and returns immediately.

`dispatch_sync` (page 70)
Submits a block object for execution on a dispatch queue and waits until that block completes.

`dispatch_sync_f` (page 71)
Submits an application-defined function for synchronous execution on a dispatch queue.

`dispatch_after` (page 14)
Enqueue a block for execution at the specified time.

`dispatch_after_f` (page 15)
Enqueues an application-defined function for execution at a specified time.

`dispatch_apply` (page 15)
Submits a block to a dispatch queue for multiple invocations.

`dispatch_apply_f` (page 16)
Submits an application-defined function to a dispatch queue for multiple invocations.

`dispatch_once` (page 47)
Executes a block object once and only once for the lifetime of an application.

Using Dispatch Groups

Grouping blocks allows for aggregate synchronization. Your application can submit multiple blocks and track when they all complete, even though they might run on different queues. This behavior can be helpful when progress can't be made until all of the specified tasks are complete.

`dispatch_group_async` (page 32)
Submits a block to a dispatch queue and associates the block with the specified dispatch group.

`dispatch_group_async_f` (page 33)
: Submits an application-defined function to a dispatch queue and associates it with the specified dispatch group.

`dispatch_group_create` (page 33)
: Creates a new group with which block objects can be associated.

`dispatch_group_enter` (page 34)
: Explicitly indicates that a block has entered the group.

`dispatch_group_leave` (page 35)
: Explicitly indicates that a block in the group has completed.

`dispatch_group_notify` (page 35)
: Schedules a block object to be submitted to a queue when a group of previously submitted block objects have completed.

`dispatch_group_notify_f` (page 36)
: Schedules an application-defined function to be submitted to a queue when a group of previously submitted block objects have completed.

`dispatch_group_wait` (page 37)
: Waits synchronously for the previously submitted block objects to complete; returns if the blocks do not complete before the specified timeout period has elapsed.

Managing Dispatch Objects

GCD provides dispatch object interfaces to allow your application to manage aspects of processing such as memory management, pausing and resuming execution, defining object context, and logging task data. Dispatch objects must be manually retained and released and are not garbage collected.

`dispatch_debug` (page 28)
: Programmatically logs debug information about a dispatch object.

`dispatch_get_context` (page 29)
: Returns the application-defined context of an object.

`dispatch_release` (page 52)
: Decrements the reference (retain) count of a dispatch object.

`dispatch_resume` (page 53)
: Resume the invocation of block objects on a dispatch object.

`dispatch_retain` (page 54)
: Increments the reference (retain) count of a dispatch object.

`dispatch_set_context` (page 56)
: Associates an application-defined context with the object.

`dispatch_set_finalizer_f` (page 57)
: Sets the finalizer function for a dispatch object.

`dispatch_suspend` (page 69)
: Suspends the invocation of block objects on a dispatch object.

Using Semaphores

A dispatch semaphore is an efficient implementation of a traditional counting semaphore. Dispatch semaphores call down to the kernel only when the calling thread needs to be blocked. If the calling semaphore does not need to block, no kernel call is made.

`dispatch_semaphore_create` (page 54)
: Creates new counting semaphore with an initial value.

`dispatch_semaphore_signal` (page 55)
: Signals (increments) a semaphore.

`dispatch_semaphore_wait` (page 55)
: Waits for (decrements) a semaphore.

Using Barriers

A dispatch barrier allows you to create a synchronization point within a concurrent dispatch queue. When it encounters a barrier, a concurrent queue delays the execution of the barrier block (or any further blocks) until all blocks submitted before the barrier finish executing. At that point, the barrier block executes by itself. Upon completion, the queue resumes its normal execution behavior.

`dispatch_barrier_async` (page 19)
: Submits a barrier block for asynchronous execution and returns immediately.

`dispatch_barrier_async_f` (page 20)
: Submits a barrier function for asynchronous execution and returns immediately.

`dispatch_barrier_sync` (page 20)
: Submits a barrier block object for execution and waits until that block completes.

`dispatch_barrier_sync_f` (page 21)
: Submits a barrier function for execution and waits until that function completes.

Managing Dispatch Sources

GCD provides a suite of dispatch sources—interfaces for monitoring (low-level system objects such as Unix descriptors, Mach ports, Unix signals, VFS nodes, and so forth) for activity. and submitting event handlers to dispatch queues when such activity occurs. When an event occurs, the dispatch source submits your task code asynchronously to the specified dispatch queue for processing.

`dispatch_source_cancel` (page 58)
: Asynchronously cancels the dispatch source, preventing any further invocation of its event handler block.

`dispatch_source_create` (page 59)
: Creates a new dispatch source to monitor low-level system objects and automatically submit a handler block to a dispatch queue in response to events.

`dispatch_source_get_data` (page 60)
: Returns pending data for the dispatch source.

`dispatch_source_get_handle` (page 61)
: Returns the underlying system handle associated with the specified dispatch source.

`dispatch_source_get_mask` (page 62)
: Returns the mask of events monitored by the dispatch source.

`dispatch_source_merge_data` (page 63)
: Merges data into a dispatch source of type `DISPATCH_SOURCE_TYPE_DATA_ADD` or `DISPATCH_SOURCE_TYPE_DATA_OR` and submits its event handler block to its target queue.

`dispatch_source_set_registration_handler` (page 66)
: Sets the registration handler block for the given dispatch source.

`dispatch_source_set_registration_handler_f` (page 67)
: Sets the registration handler function for the given dispatch source.

`dispatch_source_set_cancel_handler` (page 63)
: Sets the cancellation handler block for the given dispatch source.

`dispatch_source_set_cancel_handler_f` (page 64)
: Sets the cancellation handler function for the given dispatch source.

`dispatch_source_set_event_handler` (page 65)
: Sets the event handler block for the given dispatch source.

`dispatch_source_set_event_handler_f` (page 66)
: Sets the event handler function for the given dispatch source.

`dispatch_source_set_timer` (page 68)
: Sets a start time, interval, and leeway value for a timer source.

`dispatch_source_testcancel` (page 69)
: Tests whether the given dispatch source has been canceled.

Using the Dispatch I/O Convenience API

The dispatch I/O convenience API lets you perform asynchronous read and write operations on file descriptors. This API supports stream-based semantics for accessing the contents of the file-descriptor.

`dispatch_read` (page 51)
: Schedule an asynchronous read operation using the specified file descriptor.

`dispatch_write` (page 73)
: Schedule an asynchronous write operation using the specified file descriptor.

Using the Dispatch I/O Channel API

The dispatch I/O channel API lets you manage file descriptor–based operations. This API supports both stream-based and random-access semantics for accessing the contents of the file descriptor.

`dispatch_io_create` (page 39)
: Creates a dispatch I/O channel and associates it with the specified file descriptor.

`dispatch_io_create_with_path` (page 40)
: Creates a dispatch I/O channel with the associated path name.

`dispatch_io_read` (page 42)
: Schedules an asynchronous read operation on the specified channel.

`dispatch_io_write` (page 45)
: Schedules an asynchronous write operation for the specified channel.

`dispatch_io_close` (page 38)
: Closes the specified channel to new read and write operations.

`dispatch_io_set_high_water` (page 43)
: Sets the maximum number of bytes to process before enqueueing a handler block.

`dispatch_io_set_low_water` (page 44)
: Sets the minimum number of bytes to process before enqueueing a handler block.

`dispatch_io_set_interval` (page 44)
: Sets the interval (in nanoseconds) at which to invoke the I/O handlers for the channel.

Managing Dispatch Data Objects

Dispatch data objects present an interface for managing a memory-based data buffer. Clients accessing the data buffer see it as a contiguous block of memory, but internally the buffer may be comprised of multiple discontiguous blocks of memory.

`dispatch_data_create` (page 24)
: Creates a new dispatch data object with the specified memory buffer.

`dispatch_data_get_size` (page 27)
: Returns the logical size of the memory managed by a dispatch data object

`dispatch_data_create_map` (page 26)
: Returns a new dispatch data object containing a contiguous representation of the specified object's memory.

`dispatch_data_create_concat` (page 25)
: Returns a new dispatch data object consisting of the concatenated data from two other data objects.

`dispatch_data_create_subrange` (page 27)
: Returns a new dispatch data object whose contents consist of a portion of another object's memory region.

`dispatch_data_apply` (page 22)
: Traverses the memory of a dispatch data object and executes custom code on each region.

`dispatch_data_copy_region` (page 23)
: Returns a data object containing a portion of the data in another data object.

Managing Time

`dispatch_time` (page 71)
: Creates a `dispatch_time_t` relative to the default clock or modifies an existing `dispatch_time_t`.

`dispatch_walltime` (page 72)
: Creates a `dispatch_time_t` using an absolute time according to the wall clock.

Managing Queue-Specific Context Data

`dispatch_queue_set_specific` (page 50)
: Sets the key/value data for the specified dispatch queue.

`dispatch_queue_get_specific` (page 49)
: Gets the value for the key associated with the specified dispatch queue.

`dispatch_get_specific` (page 31)
Returns the value for the key associated with the current dispatch queue.

Functions

dispatch_after

Enqueue a block for execution at the specified time.

```
void dispatch_after(
dispatch_time_t when,
dispatch_queue_t queue,
dispatch_block_t block);
```

Parameters

`when`
The temporal milestone returned by `dispatch_time` or `dispatch_walltime`.

`queue`
The queue on which to submit the block. The queue is retained by the system until the block has run to completion. This parameter cannot be `NULL`.

`block`
The block to submit. This function performs a `Block_copy` and `Block_release` on behalf of the caller. This parameter cannot be `NULL`.

Discussion

This function waits until the specified time and then asynchronously adds `block` to the specified `queue`.

Passing `DISPATCH_TIME_NOW` (page 87) as the `when` parameter is supported, but is not as optimal as calling `dispatch_async` (page 17) instead. Passing `DISPATCH_TIME_FOREVER` (page 87) is undefined.

Availability

Available in OS X v10.6 and later.

Declared in

`dispatch/queue.h`

dispatch_after_f

Enqueues an application-defined function for execution at a specified time.

```
void dispatch_after_f(
dispatch_time_t when,
dispatch_queue_t queue,
void *context,
dispatch_function_t work);
```

Parameters

`when`

The temporal milestone returned by `dispatch_time` or `dispatch_walltime`.

`queue`

The queue on which to submit the function. The queue is retained by the system until the application-defined function has run to completion. This parameter cannot be `NULL`.

`context`

The application-defined context parameter to pass to the function.

`work`

The application-defined function to invoke on the target queue. The first parameter passed to this function is the value in the `context` parameter. This parameter cannot be `NULL`.

Discussion

This function waits until the specified time and then asynchronously adds the `work` function to the specified `queue`.

Passing `DISPATCH_TIME_NOW` (page 87) as the `when` parameter is supported, but is not as optimal as calling `dispatch_async` (page 17) instead. Passing `DISPATCH_TIME_FOREVER` (page 87) is undefined.

Availability

Available in OS X v10.6 and later.

Declared in

`dispatch/queue.h`

dispatch_apply

Submits a block to a dispatch queue for multiple invocations.

```
void dispatch_apply(
size_t iterations,
dispatch_queue_t queue,
void (^block)(
size_t));
```

Parameters

`iterations`

The number of iterations to perform.

`queue`

The queue on which to submit the block. This parameter cannot be `NULL`.

`block`

The application-defined function to be submitted. This parameter cannot be `NULL`.

Discussion

This function submits a block to a dispatch queue for multiple invocations and waits for all iterations of the task block to complete before returning. If the target queue is a concurrent queue returned by `dispatch_get_global_queue` (page 30), the block can be invoked concurrently, and it must therefore be reentrant-safe. Using this function with a concurrent queue can be useful as an efficient parallel `for` loop.

The current index of iteration is passed to each invocation of the block.

Availability

Available in OS X v10.6 and later.

Declared in

`dispatch/queue.h`

dispatch_apply_f

Submits an application-defined function to a dispatch queue for multiple invocations.

```
void dispatch_apply_f(
size_t iterations,
dispatch_queue_t queue,
void *context,
void (*work)(void *, size_t));
```

Parameters

`iterations`

The number of iterations to perform.

`queue`

The queue on which to submit the function. This parameter cannot be `NULL`.

`context`

The application-defined context parameter to pass to the function.

`work`

The application-defined function to invoke on the target queue. The first parameter passed to this function is the value in the `context` parameter. The second parameter is the current index of iteration. This parameter cannot be `NULL`.

Discussion

This function submits an application-defined function to a dispatch queue for multiple invocations and waits for all iterations of the function to complete before returning. If the target queue is a concurrent queue returned by `dispatch_get_global_queue` (page 30), the function can be invoked concurrently, and it must therefore be reentrant-safe. Using this function with a concurrent queue can be useful as an efficient parallel `for` loop.

The current index of iteration is passed to each invocation of the function.

Availability

Available in OS X v10.6 and later.

Declared in

`dispatch/queue.h`

dispatch_async

Submits a block for asynchronous execution on a dispatch queue and returns immediately.

```
void dispatch_async(
dispatch_queue_t queue,
dispatch_block_t block);
```

Parameters

`queue`

The queue on which to submit the block. The queue is retained by the system until the block has run to completion. This parameter cannot be `NULL`.

`block`

The block to submit to the target dispatch queue. This function performs `Block_copy` and `Block_release` on behalf of callers. This parameter cannot be `NULL`.

Discussion

This function is the fundamental mechanism for submitting blocks to a dispatch queue. Calls to this function always return immediately after the block has been submitted and never wait for the block to be invoked. The target queue determines whether the block is invoked serially or concurrently with respect to other blocks submitted to that same queue. Independent serial queues are processed concurrently with respect to each other.

Availability

Available in OS X v10.6 and later.

Declared in

`dispatch/queue.h`

dispatch_async_f

Submits an application-defined function for asynchronous execution on a dispatch queue and returns immediately.

```
void dispatch_async_f(
dispatch_queue_t queue,
void *context,
dispatch_function_t work);
```

Parameters

`queue`

The queue on which to submit the function. The queue is retained by the system until the function has run to completion. This parameter cannot be `NULL`.

`context`

The application-defined context parameter to pass to the function.

`work`

The application-defined function to invoke on the target queue. The first parameter passed to this function is the value of the `context` parameter. This parameter cannot be `NULL`.

Discussion

This function is the fundamental mechanism for submitting application-defined functions to a dispatch queue. Calls to this function always return immediately after the function has been submitted and never wait for it to be invoked. The target queue determines whether the function is invoked serially or concurrently with respect to other tasks submitted to that same queue. Serial queues are processed concurrently with respect to each other.

Availability
Available in OS X v10.6 and later.

Declared in
`dispatch/queue.h`

dispatch_barrier_async

Submits a barrier block for asynchronous execution and returns immediately.

```
void dispatch_barrier_async(
   dispatch_queue_t queue,
   dispatch_block_t block);
```

Parameters

`queue`
: The dispatch queue on which to execute the barrier block. The queue is retained by the system until the block has run to completion. This parameter cannot be `NULL`.

`block`
: The barrier block to submit to the target dispatch queue. This block is copied and retained until it finishes executing, at which point it is released. This parameter cannot be `NULL`.

Discussion
Calls to this function always return immediately after the block has been submitted and never wait for the block to be invoked. When the barrier block reaches the front of a private concurrent queue, it is not executed immediately. Instead, the queue waits until its currently executing blocks finish executing. At that point, the barrier block executes by itself. Any blocks submitted after the barrier block are not executed until the barrier block completes.

The queue you specify should be a concurrent queue that you create yourself using the `dispatch_queue_create` (page 48) function. If the queue you pass to this function is a serial queue or one of the global concurrent queues, this function behaves like the `dispatch_async` (page 17) function.

Availability
Available in OS X v10.7 and later.

Declared in
`dispatch/queue.h`

dispatch_barrier_async_f

Submits a barrier function for asynchronous execution and returns immediately.

```
void dispatch_barrier_async_f(
   dispatch_queue_t queue,
   void* context,
   dispatch_function_t work);
```

Parameters

`queue`
: The dispatch queue on which to execute the barrier function. The queue is retained by the system until the function has run to completion. This parameter cannot be `NULL`.

`context`
: The application-defined context parameter to pass to the function.

`work`
: The application-defined barrier function to be executed. The first parameter passed to this function is the value of the `context` parameter. This parameter cannot be `NULL`.

Discussion

Calls to this function always return immediately after the barrier function has been submitted and never wait for that function to be invoked. When the barrier function reaches the front of a private concurrent queue, it is not executed immediately. Instead, the queue waits until its currently executing blocks finish executing. At that point, the queue executes the barrier function by itself. Any blocks submitted after the barrier function are not executed until the barrier function completes.

The queue you specify should be a concurrent queue that you create yourself using the `dispatch_queue_create` (page 48) function. If the queue you pass to this function is a serial queue or one of the global concurrent queues, this function behaves like the `dispatch_async` (page 17) function.

Availability

Available in OS X v10.7 and later.

Declared in

`dispatch/queue.h`

dispatch_barrier_sync

Submits a barrier block object for execution and waits until that block completes.

```
void dispatch_barrier_sync(
   dispatch_queue_t queue,
   dispatch_block_t block);
```

Parameters

`queue`

The dispatch queue on which to execute the barrier block. This parameter cannot be `NULL`.

`block`

The barrier block to be executed. This parameter cannot be `NULL`.

Discussion

Submits a barrier block to a dispatch queue for synchronous execution. Unlike `dispatch_barrier_async` (page 19), this function does not return until the barrier block has finished. Calling this function and targeting the current queue results in deadlock.

When the barrier block reaches the front of a private concurrent queue, it is not executed immediately. Instead, the queue waits until its currently executing blocks finish executing. At that point, the queue executes the barrier block by itself. Any blocks submitted after the barrier block are not executed until the barrier block completes.

The queue you specify should be a concurrent queue that you create yourself using the `dispatch_queue_create` (page 48) function. If the queue you pass to this function is a serial queue or one of the global concurrent queues, this function behaves like the `dispatch_async_f` (page 18) function.

Unlike with `dispatch_barrier_async`, no retain is performed on the target queue. Because calls to this function are synchronous, it "borrows" the reference of the caller. Moreover, no `Block_copy` is performed on the block.

As an optimization, this function invokes the barrier block on the current thread when possible.

Availability

Available in OS X v10.7 and later.

Declared in

`dispatch/queue.h`

dispatch_barrier_sync_f

Submits a barrier function for execution and waits until that function completes.

```
void dispatch_barrier_sync_f(
   dispatch_queue_t queue,
   void* context,
   dispatch_function_t work);
```

Parameters

`queue`

The dispatch queue on which to execute the barrier function. This parameter cannot be `NULL`.

`context`

The application-defined context parameter to pass to the barrier function.

`work`

The application-defined barrier function to be executed. The first parameter passed to this function is the value in the `context` parameter. This parameter cannot be `NULL`.

Discussion

Submits a barrier function to a dispatch queue for synchronous execution. Unlike `dispatch_barrier_async_f` (page 20), this function does not return until the barrier function has finished. Calling this function and targeting the current queue results in deadlock.

When the barrier function reaches the front of a private concurrent queue, it is not executed immediately. Instead, the queue waits until its currently executing blocks finish executing. At that point, the queue executes the barrier function by itself. Any blocks submitted after the barrier function are not executed until the barrier function completes.

The queue you specify should be a concurrent queue that you create yourself using the `dispatch_queue_create` (page 48) function. If the queue you pass to this function is a serial queue or one of the global concurrent queues, this function behaves like the `dispatch_async_f` (page 18) function.

Unlike with `dispatch_barrier_async_f`, no retain is performed on the target queue. Because calls to this function are synchronous, it "borrows" the reference of the caller.

As an optimization, this function invokes the barrier function on the current thread when possible.

Availability

Available in OS X v10.7 and later.

Declared in

`dispatch/queue.h`

dispatch_data_apply

Traverses the memory of a dispatch data object and executes custom code on each region.

```
bool dispatch_data_apply(
   dispatch_data_t data,
   dispatch_data_applier_t applier);
```

Parameters

`data`

The dispatch object whose memory you want to use.

`applier`

The block to run on each contiguous memory region of `data`.

Return Value

A Boolean indicating whether the traversal completed successfully. Typically, this value is `true` if the applier block was executed on all of the regions or there was nothing to traverse. If it is `false`, it means the block terminated the traversal early.

Discussion

For each contiguous memory region, this function creates a temporary dispatch data object and passes it to the specified applier function. This new object, plus the other parameters to the block, provide direct access to the specific memory region being examined. Once the applier block returns, the temporary dispatch data object is released. (The original object in the `data` parameter is not touched.)

Availability

Available in OS X v10.7 and later.

Declared in

`dispatch/data.h`

dispatch_data_copy_region

Returns a data object containing a portion of the data in another data object.

```
dispatch_data_t dispatch_data_copy_region(
   dispatch_data_t data,
   size_t location,
   size_t *offset_ptr);
```

Parameters

`data`

The dispatch data object to query.

`location`

The byte offset to use when determining which memory region to return.

`offset_ptr`
: On input, a pointer to a variable. On output, this variable contains the offset from the beginning of `data` of the returned memory region.

Return Value

A dispatch data object whose memory region contains the specified location.

Availability

Available in OS X v10.7 and later.

Declared in

`dispatch/data.h`

dispatch_data_create

Creates a new dispatch data object with the specified memory buffer.

```
dispatch_data_t dispatch_data_create(
   const void *buffer,
   size_t size,
   dispatch_queue_t queue,
   dispatch_block_t destructor);
```

Parameters

`buffer`
: A contiguous buffer of memory containing the desired data.

`size`
: The size of `buffer`, measured in bytes.

`queue`
: The queue on which to call `destructor` when it is time to release the data object. The queue is retained by the data object.

`destructor`
: The block responsible for releasing the memory associated with the data object. For a list of constants representing the system-provided destructors, see "Data Destructor Constants" (page 84).

Return Value

A new data object containing the desired data. This object is retained initially. It is your responsibility to release the data object when you are done using it.

If buffer is `NULL` or size is `0`, this function returns an empty dispatch object.

Discussion

If you specify the default destructor using the `DISPATCH_DATA_DESTRUCTOR_DEFAULT` (page 85) constant, this function creates a copy of the data in `buffer` and manages that data internally. If you specify any other value, this function stores a pointer to your buffer and leaves the responsibility of releasing that buffer to the destructor you provide.

When you release the last reference to the object, the system typically enqueues the block in `destructor` on the provided queue. However, if you specify the `DISPATCH_DATA_DESTRUCTOR_FREE` (page 85) constant for the destructor, the system simply frees the associated memory inline.

Availability

Available in OS X v10.7 and later.

Declared in

`dispatch/data.h`

dispatch_data_create_concat

Returns a new dispatch data object consisting of the concatenated data from two other data objects.

```
dispatch_data_t dispatch_data_create_concat(
   dispatch_data_t data1,
   dispatch_data_t data2);
```

Parameters

`data1`

The first data object to include. The memory from this object is placed at the beginning of the new data object's memory region.

`data2`

The second data object to include. The memory from this object is added to the end of the memory from `data1`.

Return Value

A new dispatch data object containing the concatenated memory.

Discussion

After calling this function, it is safe to release either of the objects in `data1` or `data2`. However, be aware that the memory from those objects may not be deallocated if the newly created dispatch data object references it, as opposed to copies it.

Availability

Available in OS X v10.7 and later.

Declared in
`dispatch/data.h`

dispatch_data_create_map

Returns a new dispatch data object containing a contiguous representation of the specified object's memory.

```
dispatch_data_t dispatch_data_create_map(
   dispatch_data_t data,
   const void **buffer_ptr,
   size_t *size_ptr);
```

Parameters

`data`
A dispatch data object containing the memory to map. If the object contains multiple noncontiguous memory regions, those regions are copied to a single, contiguous memory region for the new object.

`buffer_ptr`
On input, a pointer to a variable in which to store the pointer to the memory region for the newly created dispatch data object. You may specify `NULL` for this parameter if you do not need the information.

`size_ptr`
On input, a pointer to a variable in which to store the size of the contiguous memory region in the newly created dispatch data object. You may specify `NULL` for this parameter if you do not need the information.

Return Value
A new dispatch data object containing the contiguous version of the memory managed by the object in the `data` parameter.

Discussion
If you specify non-`NULL` values for `buffer_ptr` or `size_ptr`, the values returned in those variables are valid only until you release the newly created dispatch data object. You can use these values as a quick way to access the data of the new data object.

Availability
Available in OS X v10.7 and later.

Declared in
`dispatch/data.h`

dispatch_data_create_subrange

Returns a new dispatch data object whose contents consist of a portion of another object's memory region.

```
dispatch_data_t dispatch_data_create_subrange(
   dispatch_data_t data,
   size_t offset,
   size_t length);
```

Parameters

`data`

The dispatch data object containing the original memory to use for the new object.

`offset`

A byte offset into the memory of `data`. This offset marks the starting point of the memory for the new object.

`length`

The number of bytes from `offset` to include in the new object.

Return Value

A new dispatch data object whose memory is a subrange of the memory associated with the object in the `data` parameter.

Discussion

After calling this function, it is safe to release the object in `data`. However, be aware that the memory from that object may not be deallocated immediately if the newly created dispatch data object references it, as opposed to copies it.

Availability

Available in OS X v10.7 and later.

Declared in

`dispatch/data.h`

dispatch_data_get_size

Returns the logical size of the memory managed by a dispatch data object

```
size_t dispatch_data_get_size(
   dispatch_data_t data);
```

Parameters

`data`

The dispatch data object to query

Return Value

The number of bytes represented by the data object.

Discussion

For data objects that represent multiple noncontiguous memory regions, the size reported by this function is the sum of the sizes of the individual regions.

Availability

Available in OS X v10.7 and later.

Declared in

`dispatch/data.h`

dispatch_debug

Programmatically logs debug information about a dispatch object.

```
void dispatch_debug(
   dispatch_object_t object,
   const char *message,
   ...);
```

Parameters

`object`

The object to introspect.

`message`

The message to log above and beyond the introspection, in the form of a printf-style format string. The content of this message is appended to the log message separated by a colon, like this: "{*dispatch_object_information*}: *message*".

Discussion

Debug information is logged to the Console log. This information can be useful as a debugging tool to view the internal state (current reference count, suspension count, etc.) of a dispatch object at the time the `dispatch_debug` (page 28) function is called.

Availability

Available in OS X v10.6 and later.

Declared in
`dispatch/object.h`

dispatch_get_context

Returns the application-defined context of an object.

```
   void * dispatch_get_context(
   dispatch_object_t object);
```

Parameters
`object`

This parameter cannot be `NULL`.

Return Value
The context of the object; can be `NULL`.

Discussion
Your application can associate custom context data with the object, to be used only by your application. Your application must allocate and deallocate the data as appropriate.

Availability
Available in OS X v10.6 and later.

Declared in
`dispatch/object.h`

dispatch_get_current_queue

Returns the queue on which the currently executing block is running.

```
dispatch_queue_t dispatch_get_current_queue(
   void);
```

Return Value
Returns the current queue.

Discussion
This function is defined to never return `NULL`.

When called from outside of the context of a submitted block, this function returns the main queue if the call is executed from the main thread. If the call is made from any other thread, this function returns the default concurrent queue.

Availability
Available in OS X v10.6 and later.

Declared in
`dispatch/queue.h`

dispatch_get_global_queue

Returns a well-known global concurrent queue of a given priority level.

```
dispatch_queue_t dispatch_get_global_queue(
   long priority,
   unsigned long flags);
```

Parameters

`priority`
  The priority of the queue being retrieved. For a list of possible values, see "`dispatch_queue_priority_t`" (page 81).

`flags`
  This value is reserved for future use. You should always pass 0.

Return Value
Returns the requested global queue.

Discussion
The well-known global concurrent queues cannot be modified. Calls to `dispatch_suspend` (page 69), `dispatch_resume` (page 53), `dispatch_set_context` (page 56), and the like have no effect when used with queues returned by this function.

Blocks submitted to these global concurrent queues may be executed concurrently with respect to each other.

Availability
Available in OS X v10.6 and later.

Declared in
`dispatch/queue.h`

dispatch_get_main_queue

Returns the serial dispatch queue associated with the application's main thread.

```
dispatch_queue_t dispatch_get_main_queue(void);
```

Return Value
Returns the main queue. This queue is created automatically on behalf of the main thread before `main` is called.

Discussion
The main queue is automatically created by the system and associated with your application's main thread. Your application uses one (and only one) of the following three approaches to invoke blocks submitted to the main queue:

- Calling `dispatch_main` (page 47)
- Calling `UIApplicationMain` (iOS) or `NSApplicationMain` (OS X)
- Using a `CFRunLoopRef` on the main thread

As with the global concurrent queues, calls to `dispatch_suspend` (page 69), `dispatch_resume` (page 53), `dispatch_set_context` (page 56), and the like have no effect when used with queues returned by this function.

Availability
Available in OS X v10.6 and later.

Declared in
`dispatch/queue.h`

dispatch_get_specific

Returns the value for the key associated with the current dispatch queue.

```
void* dispatch_get_specific(const void *key);
```

Parameters
`key`
The key associated with the dispatch queue on which the current block is executing. Keys are only compared as pointers and never dereferenced. Passing a string constant directly is not recommended.

Return Value
The context value for the specified key; otherwise `NULL` if the key was not set for the queue (or its target queue) or the queue is a global concurrent queue.

Discussion

This function is intended to be called from a block executing in a dispatch queue. You use it to obtain context data associated with the queue. Calling this method from code not running in a dispatch queue returns `NULL` because there is no queue to provide context.

Availability

Available in OS X v10.7 and later.

Declared in

`dispatch/queue.h`

dispatch_group_async

Submits a block to a dispatch queue and associates the block with the specified dispatch group.

```
void dispatch_group_async(
dispatch_group_t group,
dispatch_queue_t queue,
dispatch_block_t block);
```

Parameters

`group`

A dispatch group to associate the submitted block object with. The group is retained by the system until the block has run to completion. This parameter cannot be `NULL`.

`queue`

The dispatch queue to which the block object is submitted for asynchronous invocation. The queue is retained by the system until the block has run to completion. This parameter cannot be `NULL`.

`block`

The block object to perform asynchronously. This function performs a `Block_copy` and `Block_release` on behalf of the caller.

Discussion

Submits a block to a dispatch queue and associates the block object with the given dispatch group. The dispatch group can be used to wait for the completion of the block objects it references.

Availability

Available in OS X v10.6 and later.

Declared in

`dispatch/group.h`

dispatch_group_async_f

Submits an application-defined function to a dispatch queue and associates it with the specified dispatch group.

```
void dispatch_group_async_f(
dispatch_group_t group,
dispatch_queue_t queue,
void *context,
dispatch_function_t work);
```

Parameters

`group`
: A dispatch group to associate the submitted function with. The group is retained by the system until the application-defined function has run to completion. This parameter cannot be `NULL`.

`queue`
: The dispatch queue to which the function is submitted for asynchronous invocation. The queue is retained by the system until the application-defined function has run to completion. This parameter cannot be `NULL`.

`context`
: The application-defined context parameter to pass to the application-defined function.

`work`
: The application-defined function to invoke on the target queue. The first parameter passed to this function is the value in the `context` parameter.

Discussion

Submits an application-defined function to a dispatch queue and associates it with the given dispatch group. The dispatch group can be used to wait for the completion of the application-defined functions it references.

Availability

Available in OS X v10.6 and later.

Declared in

`dispatch/group.h`

dispatch_group_create

Creates a new group with which block objects can be associated.

```
dispatch_group_t dispatch_group_create(
void);
```

Return Value
The newly created group, or `NULL` on failure.

Discussion
This function creates a new group with which block objects can be associated (by using the `dispatch_group_async` (page 32) function). The dispatch group maintains a count of its outstanding associated tasks, incrementing the count when a new task is associated and decrementing it when a task completes. Functions such as `dispatch_group_notify` (page 35) and `dispatch_group_wait` (page 37) use that count to allow your application to determine when all tasks associated with the group have completed. At that time, your application can take any appropriate action.

When your application no longer needs the dispatch group, it should call `dispatch_release` (page 52) to release its reference to the group object and ultimately free its memory.

Availability
Available in OS X v10.6 and later.

Declared in
`dispatch/group.h`

dispatch_group_enter

Explicitly indicates that a block has entered the group.

```
void dispatch_group_enter(
dispatch_group_t group);
```

Parameters
`group`
    The dispatch group to update. This parameter cannot be `NULL`.

Discussion
Calling this function increments the current count of outstanding tasks in the group. Using this function (with `dispatch_group_leave` (page 35)) allows your application to properly manage the task reference count if it explicitly adds and removes tasks from the group by a means other than using the `dispatch_group_async` (page 32) function. A call to this function must be balanced with a call to `dispatch_group_leave`. You can use this function to associate a block with more than one group at the same time.

Availability
Available in OS X v10.6 and later.

Declared in
`dispatch/group.h`

dispatch_group_leave

Explicitly indicates that a block in the group has completed.

```
void dispatch_group_leave(
dispatch_group_t group);
```

Parameters
`group`

The dispatch group to update. This parameter cannot be `NULL`.

Discussion
Calling this function decrements the current count of outstanding tasks in the group. Using this function (with `dispatch_group_enter` (page 34)) allows your application to properly manage the task reference count if it explicitly adds and removes tasks from the group by a means other than using the `dispatch_group_async` (page 32) function.

A call to this function must balance a call to `dispatch_group_enter`. It is invalid to call it more times than `dispatch_group_enter`, which would result in a negative count.

Availability
Available in OS X v10.6 and later.

Declared in
`dispatch/group.h`

dispatch_group_notify

Schedules a block object to be submitted to a queue when a group of previously submitted block objects have completed.

```
void dispatch_group_notify(
dispatch_group_t group,
dispatch_queue_t queue,
dispatch_block_t block);
```

Parameters

`group`
: The dispatch group to observe. The group is retained by the system until the block has run to completion. This parameter cannot be `NULL`.

`queue`
: The queue to which the supplied block is submitted when the group completes. The queue is retained by the system until the block has run to completion. This parameter cannot be `NULL`.

`block`
: The block to submit when the group completes. This function performs a `Block_copy` and `Block_release` on behalf of the caller. This parameter cannot be `NULL`.

Discussion

This function schedules a notification block to be submitted to the specified queue when all blocks associated with the dispatch group have completed. If the group is empty (no block objects are associated with the dispatch group), the notification block object is submitted immediately.

When the notification block is submitted, the group is empty. The group can either be released with `dispatch_release` (page 52) or be reused for additional block objects. See `dispatch_group_async` (page 32) for more information.

Availability

Available in OS X v10.6 and later.

Declared in

`dispatch/group.h`

dispatch_group_notify_f

Schedules an application-defined function to be submitted to a queue when a group of previously submitted block objects have completed.

```
void dispatch_group_notify_f(
dispatch_group_t group,
dispatch_queue_t queue,
void *context,
dispatch_function_t work);
```

Parameters

`group`
: The dispatch group to observe. The group is retained by the system until the application-defined function has run to completion. This parameter cannot be `NULL`.

`queue`
: The queue to which the supplied block is submitted when the group completes. The queue is retained by the system until the application-defined function has run to completion. This parameter cannot be `NULL`.

`context`
: The application-defined context parameter to pass to the application-defined function.

`work`
: The application-defined function to invoke on the target queue. The first parameter passed to this function is the value in the `context` parameter.

Discussion

This function schedules a notification block to be submitted to the specified queue when all blocks associated with the dispatch group have completed. If the group is empty (no block objects are associated with the dispatch group), the notification block object is submitted immediately.

When the notification block is submitted, the group is empty. The group can either be released with `dispatch_release` (page 52) or be reused for additional blocks. See `dispatch_group_async` (page 32) for more information.

Availability

Available in OS X v10.6 and later.

Declared in

`dispatch/group.h`

dispatch_group_wait

Waits synchronously for the previously submitted block objects to complete; returns if the blocks do not complete before the specified timeout period has elapsed.

```
long dispatch_group_wait(
dispatch_group_t group,
dispatch_time_t timeout);
```

Parameters

`group`

: The dispatch group to wait on. This parameter cannot be `NULL`.

`timeout`

: When to timeout (see `dispatch_time` (page 71)). The `DISPATCH_TIME_NOW` (page 87) and `DISPATCH_TIME_FOREVER` (page 87) constants are provided as a convenience.

Return Value

Returns zero on success (all blocks associated with the group completed before the specified timeout) or non-zero on error (timeout occurred).

Discussion

This function waits for the completion of the blocks associated with the given dispatch group and returns when either all blocks have completed or the specified timeout has elapsed. When a timeout occurs, the group is restored to its original state.

This function returns immediately if the dispatch group is empty (there are no blocks associated with the group).

After the successful return of this function, the dispatch group is empty. It can either be released with `dispatch_release` (page 52) or be reused for additional blocks. See `dispatch_group_async` (page 32) for more information.

Availability

Available in OS X v10.6 and later.

Declared in

`dispatch/group.h`

dispatch_io_close

Closes the specified channel to new read and write operations.

```
void dispatch_io_close(
   dispatch_io_t channel,
   dispatch_io_close_flags_t flags);
```

Parameters

`channel`

: The channel to close.

`flags`
: The options to use when closing the channel. For a list of possible values, see "Channel Closing Options" (page 89).

Discussion

After calling this function, you should not schedule any more read or write operations on the channel. Doing so will cause an error to be sent to your handler.

If the `DISPATCH_IO_STOP` (page 89) option is specified in the `flags` parameter, the system attempts to interrupt any outstanding read and write operations on the I/O channel. Even if you specify this flag, the corresponding handlers may be invoked with partial results. In addition, the final invocation of the handler will be passed the `ECANCELED` error code to indicate that the operation was interrupted. If you do not specify the `DISPATCH_IO_STOP` flag, read and write operations on the channel run to completion as normal.

Availability

Available in OS X v10.7 and later.

Declared in

`dispatch/io.h`

dispatch_io_create

Creates a dispatch I/O channel and associates it with the specified file descriptor.

```
dispatch_io_t dispatch_io_create(
   dispatch_io_type_t type,
   dispatch_fd_t fd,
   dispatch_queue_t queue,
   void (^cleanup_handler)(int error));
```

Parameters

`type`
: The type of channel to create. For a list of possible options, see "Dispatch I/O Channel Types" (page 88).

`fd`
: The file descriptor to associate with the channel.

`queue`
: The dispatch queue to associate with the channel. This queue is used to execute the channel's clean up handler. The channel retains this queue.

`cleanup_handler`
: The block to enqueue when the system relinquishes control of the channel's file descriptor. This channel takes a single parameter that indicates the reason why control was relinquished. If the `error` parameter contains a non zero value, control was relinquished because there was an error creating the channel; otherwise, this value should be `0`.

Return Value

The dispatch I/O channel or `NULL` if an error occurred. The returned object is retained before it is returned; it is your responsibility to close the channel and then release this object when you are done using it.

Discussion

You use this function to create a dispatch I/O channel for an already open file descriptor. After calling this function, the system takes control of the specified file descriptor until one of the following occurs:

- You close the channel by calling the `dispatch_io_close` (page 38) function.
- An unrecoverable error occurs on the file descriptor.
- All references to the channel are released.

While it controls the file descriptor, the system may modify it on behalf of the application. For example, the system typically adds the `O_NONBLOCK` flag to ensure that any operations on the file descriptor are non-blocking. During that time, it is an error for your application to modify the file descriptor directly. However, you may create additional channels using the same file descriptor.

Availability

Available in OS X v10.7 and later.

Declared in

`dispatch/io.h`

dispatch_io_create_with_path

Creates a dispatch I/O channel with the associated path name.

```
dispatch_io_t dispatch_io_create_with_path(
   dispatch_io_type_t type,
   const char* path,
   int oflag,
   mode_t mode,
   dispatch_queue_t queue,
   void (^cleanup_handler)(int error));
```

Parameters

`type`

The type of channel to create. For a list of possible options, see "Dispatch I/O Channel Types" (page 88).

`path`

The file system path to open and use for the channel I/O. This path is opened using the `open` system call.

`oflag`

The flags to pass to the `open` function when opening the path.

`mode`

The mode to pass to the `open` function when creating a file at the specified path. If you are not creating a file, specify `0`.

`queue`

The dispatch queue to associate with the channel. This queue is used to execute the channel's clean up handler. The channel retains this queue.

`cleanup_handler`

The block to enqueue when the system relinquishes control of the channel's file descriptor. This channel takes a single parameter that indicates the reason why control was relinquished. If the `error` parameter contains a non zero value, control was relinquished because there was an error creating the channel; otherwise, this value should be `0`.

Return Value

The dispatch I/O channel or `NULL` if an error occurred. The returned object is retained before it is returned; it is your responsibility to close the channel and then release this object when you are done using it.

Discussion

This function associates the specified path with the channel but does not open a file descriptor for that path until you perform the first I/O operation. While it is open, the channel owns the file descriptor. The channel closes the file descriptor and calls its cleanup handler when one of the following occurs:

- You close the channel by calling the `dispatch_io_close` (page 38) function.
- An unrecoverable error occurs on the file descriptor.
- All references to the channel are released.

Availability

Available in OS X v10.7 and later.

Declared in

`dispatch/io.h`

dispatch_io_read

Schedules an asynchronous read operation on the specified channel.

```
void dispatch_io_read(
   dispatch_io_t channel,
   off_t offset,
   size_t length,
   dispatch_queue_t queue,
   dispatch_io_handler_t io_handler);
```

Parameters

`channel`

The channel to use when reading the data.

`offset`

For random-access channels, this parameter specifies the offset into the channel from which to read. The offset is specified relative to the initial file pointer of the channel's file descriptor at the time the channel was created.

For stream-based channels, this parameter is ignored and data is read from the current position.

`length`

The number of bytes to read from the channel. Specify `SIZE_MAX` to continue reading data until an EOF is reached.

`queue`

The dispatch queue on which to submit the `io_handler` block.

`io_handler`

The block to use to process the data read from the channel. This block may be queued multiple times to process a given data request. Each time the block is queued, the `data` parameter passed to the handler contains the most recently read chunk of data.

Your block need not be reentrant. The system guarantees that only one instance of this block will be executed at any given time.

Discussion

This function reads the specified data and submits the `io_handler` block to the `queue` to process the data. If the `done` parameter of the handler is set to `NO`, it means that only part of the data was read. If the `done` parameter is set to `YES`, it means the read operation is complete and the handler will not be submitted again. If an unrecoverable error occurs on the channel's file descriptor, the `done` parameter is set to `YES` and an appropriate error value is reported in the handler's `error` parameter.

If the handler is submitted with the `done` parameter set to `YES`, an empty data object, and an error code of `0`, it means that the channel reached the end of the file.

Availability
Available in OS X v10.7 and later.

Declared in
`dispatch/io.h`

dispatch_io_set_high_water

Sets the maximum number of bytes to process before enqueueing a handler block.

```
void dispatch_io_set_high_water(
   dispatch_io_t channel,
   size_t high_water);
```

Parameters

`channel`
The channel whose high-water mark you want to configure.

`high_water`
The maximum number of bytes to read or write before enqueueing the corresponding I/O handler block.

Discussion
During a read or write operation, the channel uses the high- and low-water mark values to determine how often to enqueue the associated handler block. It enqueues the block when the number of bytes read or written is between these two values.

The default high-water mark for channels is set to `SIZE_MAX`.

Availability
Available in OS X v10.7 and later.

Declared in
`dispatch/io.h`

See Also
`dispatch_io_set_low_water` (page 44)

dispatch_io_set_interval

Sets the interval (in nanoseconds) at which to invoke the I/O handlers for the channel.

```
void dispatch_io_set_interval(
   dispatch_io_t channel,
   uint64_t interval,
   dispatch_io_interval_flags_t flags);
```

Parameters

`channel`

The channel whose interval you want to configure.

`interval`

The number of nanoseconds that must elapse before the scheduling of any I/O handlers is desired.

`flags`

Flags indicating the desired delivery behavior at the interval time. For a list of flags, see "Channel Configuration Options" (page 89).

Discussion

A channel interval is a way for you to receive periodic progress reports on the state of a read or write operation. You can use this feedback to update progress bars or other parts of your application.

If you set an interval on a channel, the handlers for any read or write operations are enqueued at the given interval only if the amount of data that has been processed exceeds the current low-water mark for the channel. Passing the `DISPATCH_IO_STRICT_INTERVAL` (page 90) constant in the `flags` parameter forces the enqueueing of the handlers even if the low-water mark is not exceeded.

The system may add a small amount of leeway to the specified interval in order to align the delivery of handlers with other system activity. The purpose of this behavior is to improve overall performance or power consumption for the system.

Availability

Available in OS X v10.7 and later.

Declared in

`dispatch/io.h`

dispatch_io_set_low_water

Sets the minimum number of bytes to process before enqueueing a handler block.

```
void dispatch_io_set_low_water(
   dispatch_io_t channel,
   size_t low_water);
```

Parameters

`channel`

The channel whose low-water mark you want to configure.

`low_water`

The minimum number of bytes to read or write before enqueueing the corresponding I/O handler block.

Discussion

During a read or write operation, the channel uses the high- and low-water mark values to determine how often to enqueue the associated handler block. It enqueues the block when the number of bytes read or written is between these two values. The only times when the number of bytes may be less than the low-water mark are when an EOF is reached or the channel interval has the `DISPATCH_IO_STRICT_INTERVAL` (page 90) flag set.

In practice, your handlers should be designed to handle data blocks that are significantly larger than the current low-water mark. If you always want to process the same amount of data in your handler, set the low- and high-water marks to the same value.

The default low-water mark for channels is unspecified. However, you should assume that partial results can be returned even with this default value. If you want to prevent the return of partial results, set the low-water mark to `SIZE_MAX`.

Availability

Available in OS X v10.7 and later.

Declared in

`dispatch/io.h`

See Also

`dispatch_io_set_high_water` (page 43)

dispatch_io_write

Schedules an asynchronous write operation for the specified channel.

```
void dispatch_io_write(
   dispatch_io_t channel,
   off_t offset,
   dispatch_data_t data,
   dispatch_queue_t queue,
   dispatch_io_handler_t io_handler);
```

Parameters

`channel`

The channel to use when writing the data.

`offset`

For random-access channels, this parameter specifies the offset into the channel at which to write. The offset is specified relative to the initial file pointer of the channel's file descriptor at the time the channel was created.

For stream-based channels, this parameter is ignored and data is written to the current position.

`data`

The data to write to the channel.

`queue`

The dispatch queue on which to submit the `io_handler` block.

`io_handler`

The block to use to report any progress. This block may be queued multiple times to process a given data request. Each time the block is queued, the `data` parameter passed to the handler contains the data that remains to be written.

Your block need not be reentrant. The system guarantees that only one instance of this block will be executed at any given time.

Discussion

This function writes the specified data and submits the `io_handler` block to the `queue` to report on the progress of the operation. If the `done` parameter of the handler is set to `NO`, it means that only part of the data was written. If the `done` parameter is set to `YES`, it means the write operation is complete and the handler will not be submitted again. If the operation was successful, the handler's `error` parameter is set to `0`. However, if an unrecoverable error occurs on the channel's file descriptor, the `done` parameter is set to `YES` and an appropriate error value is reported in the handler's `error` parameter.

Availability

Available in OS X v10.7 and later.

Declared in

`dispatch/io.h`

dispatch_main

Executes blocks submitted to the main queue.

```
void dispatch_main(
   void);
```

Return Value

This function never returns.

Discussion

This function "parks" the main thread and waits for blocks to be submitted to the main queue. Applications that call `UIApplicationMain` (iOS), `NSApplicationMain` (OS X), or `CFRunLoopRun` on the main thread must not call `dispatch_main` (page 47).

Availability

Available in OS X v10.6 and later.

Declared in

`dispatch/queue.h`

dispatch_once

Executes a block object once and only once for the lifetime of an application.

```
   void dispatch_once(
   dispatch_once_t *predicate,
   dispatch_block_t block);
```

Parameters

`predicate`

A pointer to a `dispatch_once_t` (page 77) structure that is used to test whether the block has completed or not.

`block`

The block object to execute once.

Discussion

This function is useful for initialization of global data (singletons) in an application. Always call this function before using or testing any variables that are initialized by the block.

If called simultaneously from multiple threads, this function waits synchronously until the block has completed.

The predicate must point to a variable stored in global or static scope. The result of using a predicate with automatic or dynamic storage is undefined.

Availability
Available in OS X v10.6 and later.

Declared in
`dispatch/once.h`

dispatch_queue_create

Creates a new dispatch queue to which blocks can be submitted.

```
dispatch_queue_t dispatch_queue_create(
   const char *label
   dispatch_queue_attr_t attr);
```

Parameters

`label`

A string label to attach to the queue to uniquely identify it in debugging tools such as Instruments, `sample`, stackshots, and crash reports. Because applications, libraries, and frameworks can all create their own dispatch queues, a reverse-DNS naming style (*com.example.myqueue*) is recommended. This parameter is optional and can be `NULL`.

`attr`

In OS X v10.7 and later, specify `DISPATCH_QUEUE_SERIAL` (or `NULL`) to create a serial queue or specify `DISPATCH_QUEUE_CONCURRENT` to create a concurrent queue. In earlier versions of OS X, you must specify `NULL` for this parameter.

Return Value
The newly created dispatch queue.

Discussion
Blocks submitted to a serial queue are executed one at a time in FIFO order. Note, however, that blocks submitted to independent queues may be executed concurrently with respect to each other. Blocks submitted to a concurrent queue are dequeued in FIFO order but may run concurrently if resources are available to do so.

When your application no longer needs the dispatch queue, it should release it with the `dispatch_release` (page 52) function. Any pending blocks submitted to a queue hold a reference to that queue, so the queue is not deallocated until all pending blocks have completed.

Availability
Available in OS X v10.6 and later.

Declared in
`dispatch/queue.h`

dispatch_queue_get_label

Returns the label specified for the queue when the queue was created.

```
   const char * dispatch_queue_get_label(dispatch_queue_t queue);
```

Parameters
`queue`
    This parameter cannot be `NULL`.

Return Value
The label of the queue. The result can be `NULL` if the application does not provide a label when it creates the queue.

Availability
Available in OS X v10.6 and later.

Declared in
`dispatch/queue.h`

dispatch_queue_get_specific

Gets the value for the key associated with the specified dispatch queue.

```
void* dispatch_queue_get_specific(dispatch_queue_t queue, const void *key);
```

Parameters
`queue`
    The queue containing the desired context data. This parameter must not be `NULL`.

`key`
    The key that identifies the associated context data. Keys are only compared as pointers and are never dereferenced. Thus, you can use a pointer to a static variable for a specific subsystem or any other value that allows you to identify the value uniquely. Specifying a pointer to a string constant is not recommended.

Return Value

The context data associated with `key` or `NULL` if no context was found.

Discussion

You can use this method to get the context data associated with a specific dispatch queue. Blocks executing on a queue can use the `dispatch_get_specific` (page 31) function to retrieve the context associated with that specific queue instead.

Availability

Available in OS X v10.7 and later.

Declared in

`dispatch/queue.h`

See Also

`dispatch_queue_set_specific` (page 50)

dispatch_queue_set_specific

Sets the key/value data for the specified dispatch queue.

```
void dispatch_queue_set_specific(dispatch_queue_t queue, const void* key, void *context,
dispatch_function_t destructor);
```

Parameters

`queue`

The queue on which to set the specified key/value data. This parameter must not be `NULL`.

`key`

The key you want to use to identify the associated context data. Keys are only compared as pointers and are never dereferenced. Thus, you can use a pointer to a static variable for a specific subsystem or any other value that allows you to identify the value uniquely. Specifying a pointer to a string constant is not recommended. `NULL` is not a valid value for the key and attempts to set context data with a `NULL` key are ignored.

`context`

The context data to associate with `key`. This parameter may be `NULL`.

`destructor`

A destructor function that you can use to release your context data. This parameter may be `NULL`. If `context` is `NULL`, your destructor function is ignored.

Discussion

Use this method to associate custom context data with a dispatch queue. Blocks executing on the queue can use the `dispatch_get_specific` (page 31) function to retrieve this data while they are running.

Availability

Available in OS X v10.7 and later.

Declared in

`dispatch/queue.h`

See Also

`dispatch_queue_get_specific` (page 49)

dispatch_read

Schedule an asynchronous read operation using the specified file descriptor.

```
void dispatch_read(
   dispatch_fd_t fd,
   size_t length,
   dispatch_queue_t queue,
   void (^handler)(dispatch_data_t data, int error));
```

Parameters

`fd`

The file descriptor from which to read the data.

`length`

The maximum amount of data to read from the file descriptor.

`queue`

The queue on which to perform the specified handler block.

`handler`

The block to schedule for execution when the specified amount of data has been read or an error occurred. The parameters of the handler are as follows:

`data` - The data that was read from the file descriptor. This object contains as much data as was currently available from the file descriptor, up to the specified length. This object is owned by the system and released when the handler returns. If you wish to continue using the data, your handler must retain this object or copy the data to another location prior to returning.

`error` - This value is `0` if the data was read successfully or an EOF was reached. If an error occurred, this parameter contains the error number.

Discussion

This is a convenience function for initiating a single, asynchronous read operation from the current position of the specified file descriptor. This method is intended for simple operations where you do not need the overhead of creating a channel and do not plan on issuing more than a few calls to read or write data. Once submitted, there is no way to cancel the read operation.

After calling this function, the system takes control of the specified file descriptor until the handler block is enqueued. While it controls the file descriptor, the system may modify it on behalf of the application. For example, the system typically adds the `O_NONBLOCK` flag to ensure that any operations are non-blocking. During that time, it is an error for your application to modify the file descriptor directly. However, you may pass the file descriptor to this function or the `dispatch_write` (page 73) function to perform additional reads or writes. You may also use the file descriptor to create a new dispatch I/O channel.

The `handler` you provide is not queued for execution until the read operation finishes. In addition, if you issue multiple read or write calls for the same file descriptor using the convenience APIs, all of those operations must complete before any of the associated handlers are queued. If you are already using the file descriptor with another channel, you should use the `dispatch_io_read` (page 42) function to read data from the channel rather than use this function.

If you attempt to read past the end of file, your handler is passed an empty data object and an error code of 0. For other types of unrecoverable errors, an appropriate error code is returned along with whatever data was read before the error occurred.

Availability

Available in OS X v10.7 and later.

Declared in

`dispatch/io.h`

dispatch_release

Decrements the reference (retain) count of a dispatch object.

```
void dispatch_release(
dispatch_object_t object);
```

Parameters

`object`

The object to release. This parameter cannot be `NULL`.

Discussion

A dispatch object is asynchronously deallocated once all references to it are released (the reference count becomes zero). When your application no longer needs a dispatch object that it has created, it should call this function to release its interest in the object and allow its memory to be deallocated when appropriate. Note that GCD does not guarantee that a given client has the last or only reference to a given object.

Your application does not need to retain or release the global (main and concurrent) dispatch queues; calling this function on global dispatch queues has no effect.

Availability

Available in OS X v10.6 and later.

Declared in

`dispatch/object.h`

dispatch_resume

Resume the invocation of block objects on a dispatch object.

```
void dispatch_resume(
dispatch_object_t object);
```

Parameters

`object`
: The object to be resumed. This parameter cannot be `NULL`.

Discussion

Calling this function decrements the suspension count of a suspended dispatch queue or dispatch event source object. While the count is greater than zero, the object remains suspended. When the suspension count returns to zero, any blocks submitted to the dispatch queue or any events observed by the dispatch source while suspended are delivered.

With one exception, each call to `dispatch_resume` must balance a call to `dispatch_suspend` (page 69). New dispatch event source objects returned by `dispatch_source_create` (page 59) have a suspension count of 1 and must be resumed before any events are delivered. This approach allows your application to fully configure the dispatch event source object prior to delivery of the first event. In all other cases, it is undefined to call `dispatch_resume` more times than `dispatch_suspend`, which would result in a negative suspension count.

Availability

Available in OS X v10.6 and later.

Declared in
`dispatch/object.h`

dispatch_retain

Increments the reference (retain) count of a dispatch object.

```
void dispatch_retain(
dispatch_object_t object);
```

Parameters

`object`
The object to retain. This parameter cannot be `NULL`.

Discussion

Calls to this function must be balanced with calls to `dispatch_release` (page 52). If multiple subsystems of your application share a dispatch object, each subsystem should call `dispatch_retain` to register its interest in the object. The object is deallocated only when all subsystems have released their interest in the dispatch source.

Note that your application does not need to retain or release the global (main and concurrent) dispatch queues.

Availability

Available in OS X v10.6 and later.

Declared in
`dispatch/object.h`

dispatch_semaphore_create

Creates new counting semaphore with an initial value.

```
dispatch_semaphore_t dispatch_semaphore_create(
long value);
```

Parameters

`value`
The starting value for the semaphore. Passing a value less than zero causes `NULL` to be returned.

Return Value

The newly created semaphore, or `NULL` on failure.

Discussion

Passing zero for the value is useful for when two threads need to reconcile the completion of a particular event. Passing a value greater than zero is useful for managing a finite pool of resources, where the pool size is equal to the value.

When your application no longer needs the semaphore, it should call `dispatch_release` (page 52) to release its reference to the semaphore object and ultimately free its memory.

Availability

Available in OS X v10.6 and later.

Declared in

`dispatch/semaphore.h`

dispatch_semaphore_signal

Signals (increments) a semaphore.

```
long dispatch_semaphore_signal(
dispatch_semaphore_t dsema);
```

Parameters

`dsema`

The counting semaphore. This parameter cannot be `NULL`.

Return Value

This function returns non-zero if a thread is woken. Otherwise, zero is returned.

Discussion

Increment the counting semaphore. If the previous value was less than zero, this function wakes a thread currently waiting in `dispatch_semaphore_wait` (page 55).

Availability

Available in OS X v10.6 and later.

Declared in

`dispatch/semaphore.h`

dispatch_semaphore_wait

Waits for (decrements) a semaphore.

```
long dispatch_semaphore_wait(
dispatch_semaphore_t dsema,
dispatch_time_t timeout);
```

Parameters

`dsema`

The semaphore. This parameter cannot be `NULL`.

`timeout`

When to timeout (see `dispatch_time` (page 71)). The constants `DISPATCH_TIME_NOW` (page 87) and `DISPATCH_TIME_FOREVER` (page 87) are available as a convenience.

Return Value

Returns zero on success, or non-zero if the timeout occurred.

Discussion

Decrement the counting semaphore. If the resulting value is less than zero, this function waits in FIFO order for a signal to occur before returning.

Availability

Available in OS X v10.6 and later.

Declared in

`dispatch/semaphore.h`

dispatch_set_context

Associates an application-defined context with the object.

```
void dispatch_set_context(
dispatch_object_t object,
void *context);
```

Parameters

`object`

This parameter cannot be `NULL`.

`context`

The new application-defined context for the object. This can be `NULL`.

Discussion
Your application can associate custom context data with the object, to be used only by your application. Your application must allocate and deallocate the data as appropriate.

Availability
Available in OS X v10.6 and later.

Declared in
`dispatch/object.h`

dispatch_set_finalizer_f

Sets the finalizer function for a dispatch object.

```
void dispatch_set_finalizer_f(
dispatch_object_t object,
dispatch_function_t finalizer);
```

Parameters

`object`
The dispatch object to modify. This parameter cannot be `NULL`.

`finalizer`
The finalizer function pointer.

Discussion
The finalizer for a dispatch object is invoked on that object's target queue after all references to the object are released by calls to `dispatch_release` (page 52). The application can use the finalizer to release any resources associated with the object, such as the object's application-defined context. The context parameter passed to the finalizer function is the current context of the dispatch object at the time the finalizer call is made. The finalizer is not called if the application-defined context is `NULL`.

Availability
Available in OS X v10.6 and later.

Declared in
`dispatch/object.h`

dispatch_set_target_queue

Sets the target queue for the given object.

```
void dispatch_set_target_queue(
   dispatch_object_t object,
   dispatch_queue_t queue);
```

Parameters

`object`

The object to modify. This parameter cannot be `NULL`.

`queue`

The new target queue for the object. The queue is retained, and the previous one, if any, is released. This parameter cannot be `NULL`.

Discussion

An object's target queue is responsible for processing the object.

A dispatch queue's priority is inherited by its target queue. Use the `dispatch_get_global_queue` (page 30) function to obtain a suitable target queue of the desired priority.

A dispatch source's target queue specifies where its event handler and cancellation handler blocks are submitted.

For all dispatch objects, the target queue specifies the queue on which the object's finalizer is invoked.

Availability

Available in OS X v10.6 and later.

Declared in

`dispatch/queue.h`

dispatch_source_cancel

Asynchronously cancels the dispatch source, preventing any further invocation of its event handler block.

```
void dispatch_source_cancel(
   dispatch_source_t source);
```

Parameters

`source`

The dispatch source to be canceled. This parameter cannot be `NULL`.

Discussion

Cancellation prevents any further invocation of the event handler block for the specified dispatch source, but does not interrupt an event handler block that is already in progress. The optional cancellation handler is submitted to the target queue once the event handler block has been completed.

The cancellation handler is submitted to the source's target queue when the source's event handler has finished, indicating that it is safe to close the source's handle (file descriptor or mach port).

The optional cancellation handler is submitted to the dispatch source object's target queue only after the system has released all of its references to any underlying system objects (file descriptors or mach ports). Thus, the cancellation handler is a convenient place to close or deallocate such system objects. Note that it is invalid to close a file descriptor or deallocate a mach port currently being tracked by a dispatch source object before the cancellation handler is invoked.

Availability

Available in OS X v10.6 and later.

Declared in

`dispatch/source.h`

See Also

`dispatch_source_set_cancel_handler` (page 63)

dispatch_source_create

Creates a new dispatch source to monitor low-level system objects and automatically submit a handler block to a dispatch queue in response to events.

```
dispatch_source_t dispatch_source_create(
dispatch_source_type_t type,
uintptr_t handle,
unsigned long mask,
dispatch_queue_t queue);
```

Parameters

`type`

The type of the dispatch source. Must be one of the constants listed in "Dispatch Source Type Constants" (page 86).

`handle`

The underlying system handle to monitor. The interpretation of this argument is determined by the constant provided in the type parameter.

`mask`
: A mask of flags specifying which events are desired. The interpretation of this argument is determined by the constant provided in the type parameter.

`queue`
: The dispatch queue to which the event handler block is submitted.

Return Value

A new dispatch source object or `NULL` if the dispatch source could not be created.

Discussion

Dispatch sources are not reentrant. Any events received while the dispatch source is suspended or while the event handler block is currently executing are coalesced and delivered after the dispatch source is resumed or the event handler block has returned.

Dispatch sources are created in a suspended state. After creating the source and setting any desired attributes (for example, the handler or the context), your application must call `dispatch_resume` (page 53) to begin event delivery.

When your application no longer needs the event source, it should call `dispatch_release` (page 52) to release its reference to the source object and ultimately free its memory.

Availability

Available in OS X v10.6 and later.

Declared in

`dispatch/source.h`

dispatch_source_get_data

Returns pending data for the dispatch source.

```
unsigned long dispatch_source_get_data(
   dispatch_source_t source);
```

Parameters

`source`
: This parameter cannot be `NULL`.

Return Value

The return value should be interpreted according to the type of the dispatch source, and can be one of the following:

- `DISPATCH_SOURCE_TYPE_DATA_ADD` (page 86): application-defined data
- `DISPATCH_SOURCE_TYPE_DATA_OR` (page 86): application-defined data
- `DISPATCH_SOURCE_TYPE_MACH_SEND` (page 86): "`dispatch_source_mach_send_flags_t`" (page 82)
- `DISPATCH_SOURCE_TYPE_PROC` (page 86): "`dispatch_source_proc_flags_t`" (page 82)
- `DISPATCH_SOURCE_TYPE_READ` (page 86): estimated bytes available to read
- `DISPATCH_SOURCE_TYPE_SIGNAL` (page 87): number of signals delivered since the last handler invocation
- `DISPATCH_SOURCE_TYPE_TIMER` (page 87): number of times the timer has fired since the last handler invocation

Discussion

Call this function from within the event handler block. The result of calling this function outside of the event handler callback is undefined.

Availability

Available in OS X v10.6 and later.

Declared in

`dispatch/source.h`

dispatch_source_get_handle

Returns the underlying system handle associated with the specified dispatch source.

```
uintptr_t dispatch_source_get_handle(
dispatch_source_t source);
```

Parameters

`source`

This parameter cannot be NULL.

Return Value

The return value should be interpreted according to the type of the dispatch source, and can be one of the following handles:

- `DISPATCH_SOURCE_TYPE_MACH_SEND` (page 86): mach port (`mach_port_t`)
- `DISPATCH_SOURCE_TYPE_MACH_RECV` (page 86): mach port (`mach_port_t`)
- `DISPATCH_SOURCE_TYPE_PROC` (page 86): process identifier (`pid_t`)
- `DISPATCH_SOURCE_TYPE_READ` (page 86): file descriptor (`int`)

- `DISPATCH_SOURCE_TYPE_SIGNAL` (page 87): signal number (`int`)
- `DISPATCH_SOURCE_TYPE_VNODE` (page 87): file descriptor (`int`)
- `DISPATCH_SOURCE_TYPE_WRITE` (page 87): file descriptor (`int`)

Discussion

The handle returned is a reference to the underlying system object being monitored by the dispatch source.

Availability

Available in OS X v10.6 and later.

Declared in

`dispatch/source.h`

dispatch_source_get_mask

Returns the mask of events monitored by the dispatch source.

```
unsigned long dispatch_source_get_mask(
   dispatch_source_t source);
```

Parameters

`source`

This parameter cannot be `NULL`.

Return Value

The return value should be interpreted according to the type of the dispatch source, and can be one of the following flag sets:

- `DISPATCH_SOURCE_TYPE_MACH_SEND` (page 86): “`dispatch_source_mach_send_flags_t`” (page 82)
- `DISPATCH_SOURCE_TYPE_PROC` (page 86): “`dispatch_source_proc_flags_t`” (page 82)
- `DISPATCH_SOURCE_TYPE_VNODE` (page 87): “`dispatch_source_vnode_flags_t`” (page 83)

Discussion

The mask is a bitmask of relevant events being monitored by the dispatch event source. Any events that are not specified in the event mask are ignored and no event handler block is submitted for them.

For details, see the flag descriptions in “Constants” (page 81).

Availability

Available in OS X v10.6 and later.

Declared in
`dispatch/source.h`

dispatch_source_merge_data

Merges data into a dispatch source of type `DISPATCH_SOURCE_TYPE_DATA_ADD` or `DISPATCH_SOURCE_TYPE_DATA_OR` and submits its event handler block to its target queue.

```
void dispatch_source_merge_data(
   dispatch_source_t source,
   unsigned long value);
```

Parameters

`source`
This parameter cannot be `NULL`.

`value`
The value to coalesce with the pending data using a logical OR or an ADD as specified by the dispatch source type. A value of zero has no effect and does not result in the submission of the event handler block.

Discussion
Your application can use this function to indicate that an event has occurred on one of the application-defined dispatch event sources of type `DISPATCH_SOURCE_TYPE_DATA_ADD` (page 86) or `DISPATCH_SOURCE_TYPE_DATA_OR` (page 86).

Availability
Available in OS X v10.6 and later.

Declared in
`dispatch/source.h`

dispatch_source_set_cancel_handler

Sets the cancellation handler block for the given dispatch source.

```
void dispatch_source_set_cancel_handler(
   dispatch_source_t source,
   dispatch_block_t cancel_handler);
```

Parameters

`source`

The dispatch source to modify. This parameter cannot be `NULL`.

`handler`

The cancellation handler block to submit to the source's target queue. This function performs a `Block_copy` on behalf of the caller, and `Block_release` on the previous handler (if any). This parameter can be `NULL`.

Discussion

The cancellation handler (if specified) is submitted to the source's target queue in response to a call to `dispatch_source_cancel` (page 58) when the system has released all references to the source's underlying handle and the source's event handler block has returned.

Important: To safely close a file descriptor or destroy a Mach port, a cancellation handler is required for the source for that descriptor or port. Closing the descriptor or port before the cancellation handler runs can result in a race condition. If a new descriptor is allocated with the same value as the recently closed descriptor while the source's event handler is still running, the event handler may read/write data using the wrong descriptor.

Availability

Available in OS X v10.6 and later.

Declared in

`dispatch/source.h`

dispatch_source_set_cancel_handler_f

Sets the cancellation handler function for the given dispatch source.

```
void dispatch_source_set_cancel_handler_f(
   dispatch_source_t source,
   dispatch_function_t cancel_handler);
```

Parameters

`source`

The dispatch source to modify. This parameter cannot be `NULL`.

`handler`
: The cancellation handler function to submit to the source's target queue. The context parameter passed to the event handler function is the current context of the dispatch source at the time the handler call is made.

Discussion

The cancellation handler (if specified) is submitted to the source's target queue in response to a call to `dispatch_source_cancel` (page 58) when the system has released all references to the source's underlying handle and the source's event handler block has returned.

> **Important:** To safely close a file descriptor or destroy a Mach port, a cancellation handler is required for the source for that descriptor or port. Closing the descriptor or port before the cancellation handler runs can result in a race condition. If a new descriptor is allocated with the same value as the recently closed descriptor while the source's event handler is still running, the event handler may read/write data using the wrong descriptor.

Availability

Available in OS X v10.6 and later.

Declared in

`dispatch/source.h`

dispatch_source_set_event_handler

Sets the event handler block for the given dispatch source.

```
void dispatch_source_set_event_handler(
   dispatch_source_t source,
   dispatch_block_t handler);
```

Parameters

`source`
: The dispatch source to modify. This parameter cannot be `NULL`.

`handler`
: The event handler block to submit to the source's target queue. This function performs a `Block_copy` on behalf of the caller, and `Block_release` on the previous handler (if any). This parameter cannot be `NULL`.

Discussion

The event handler (if specified) is submitted to the source's target queue in response to the arrival of an event.

Availability
Available in OS X v10.6 and later.

Declared in
`dispatch/source.h`

dispatch_source_set_event_handler_f

Sets the event handler function for the given dispatch source.

```
void dispatch_source_set_event_handler_f(
   dispatch_source_t source,
   dispatch_function_t handler);
```

Parameters

`source`
: The dispatch source to modify. This parameter cannot be `NULL`.

`handler`
: The event handler function to submit to the source's target queue. The context parameter passed to the event handler function is the current context of the dispatch source at the time the handler call is made. This parameter cannot be `NULL`.

Discussion
The event handler (if specified) is submitted to the source's target queue in response to the arrival of an event.

Availability
Available in OS X v10.6 and later.

Declared in
`dispatch/source.h`

dispatch_source_set_registration_handler

Sets the registration handler block for the given dispatch source.

```
void dispatch_source_set_registration_handler(
   dispatch_source_t source,
   dispatch_block_t registration_handler);
```

Parameters

`source`

The dispatch source to modify. This parameter cannot be `NULL`.

`registration_handler`

The registration handler block to install. The previous registration handler (if any) is released before the new one is installed. This function uses a `Block_copy` call to store a copy of the new registration handler. This parameter can be `NULL`.

Discussion

The registration handler (if specified) is submitted to the source's target queue as soon as the source has been fully set up and is ready to start delivering events. The set up of a dispatch source's underlying event-delivery mechanism occurs asynchronously. Installing a registration handler is a way to be notified when that set up is complete and the dispatch source is ready to start delivering events.

After your operation handler is executed, the dispatch source uninstalls it. Thus, registration handlers are executed only once after you resume the dispatch source.

If you install a registration handler on a dispatch source that is already set up and running, your handler is invoked immediately.

dispatch_source_set_registration_handler_f

Sets the registration handler function for the given dispatch source.

```
void dispatch_source_set_registration_handler_f(
   dispatch_source_t source,
   dispatch_function_t registration_handler);
```

Parameters

`source`

The dispatch source to modify. This parameter cannot be `NULL`.

`registration_handler`

The registration handler function to install. The previous registration handler (if any) is released before the new one is installed. The context parameter passed to the event handler function is the current context of the dispatch source at the time the handler call is made. This parameter can be `NULL`.

Discussion

The registration handler (if specified) is submitted to the source's target queue as soon as the source has been fully set up and is ready to start delivering events. The set up of a dispatch source's underlying event-delivery mechanism occurs asynchronously. Installing a registration handler is a way to be notified when that set up is complete and the dispatch source is ready to start delivering events.

After your operation handler is executed, the dispatch source uninstalls it. Thus, registration handlers are executed only once after you resume the dispatch source.

If you install a registration handler on a dispatch source that is already set up and running, your handler is invoked immediately.

dispatch_source_set_timer

Sets a start time, interval, and leeway value for a timer source.

```
void dispatch_source_set_timer(
dispatch_source_t source,
dispatch_time_t start,
uint64_t interval,
uint64_t leeway);
```

Parameters

`start`

The start time of the timer. See `dispatch_time` (page 71) and `dispatch_walltime` (page 72) for more information.

`interval`

The nanosecond interval for the timer.

`leeway`

The amount of time, in nanoseconds, that the system can defer the timer.

Discussion

Your application can call this function multiple times on the same dispatch timer source object to reset the time interval for the timer source as necessary.

The `start` time parameter also determines which clock is used for the timer. If the start time is `DISPATCH_TIME_NOW` (page 87) or is created with `dispatch_time` (page 71), the timer is based on `mach_absolute_time`. Otherwise, if the start time of the timer is created with `dispatch_walltime` (page 72), the timer is based on `gettimeofday`(3).

The `leeway` parameter is a hint from the application as to the amount of time, in nanoseconds, up to which the system can defer the timer to align with other system activity for improved system performance or power consumption. For example, an application might perform a periodic task every 5 minutes, with a leeway of up to 30 seconds. Note that some latency is to be expected for all timers, even when a leeway value of zero is specified.

Calling this function has no effect if the timer source has already been canceled.

Availability
Available in OS X v10.6 and later.

Declared in
`dispatch/source.h`

dispatch_source_testcancel

Tests whether the given dispatch source has been canceled.

```
long dispatch_source_testcancel(
   dispatch_source_t source);
```

Parameters

`source`
    The dispatch source to be tested. This parameter cannot be `NULL`.

Return Value
Non-zero if canceled and zero if not canceled.

Discussion
Your application can use this function to test whether a dispatch source object has been canceled by a call to `dispatch_source_cancel` (page 58). The result of this function is non-zero immediately after `dispatch_source_cancel` has been called.

Availability
Available in OS X v10.6 and later.

Declared in
`dispatch/source.h`

dispatch_suspend

Suspends the invocation of block objects on a dispatch object.

```
   void dispatch_suspend(
   dispatch_object_t object);
```

Parameters

`object`

The dispatch queue or dispatch source to suspend. (You cannot suspend other types of dispatch objects.) This parameter cannot be `NULL`.

Discussion

By suspending a dispatch object, your application can temporarily prevent the execution of any blocks associated with that object. The suspension occurs after completion of any blocks running at the time of the call. Calling this function increments the suspension count of the object, and calling `dispatch_resume` (page 53) decrements it. While the count is greater than zero, the object remains suspended, so you must balance each `dispatch_suspend` call with a matching `dispatch_resume` call.

Any blocks submitted to a dispatch queue or events observed by a dispatch source are delivered once the object is resumed.

Availability

Available in OS X v10.6 and later.

Declared in

`dispatch/object.h`

dispatch_sync

Submits a block object for execution on a dispatch queue and waits until that block completes.

```
void dispatch_sync(
dispatch_queue_t queue,
dispatch_block_t block);
```

Parameters

`queue`

The queue on which to submit the block. This parameter cannot be `NULL`.

`block`

The block to be invoked on the target dispatch queue. This parameter cannot be `NULL`.

Discussion

Submits a block to a dispatch queue for synchronous execution. Unlike `dispatch_async` (page 17), this function does not return until the block has finished. Calling this function and targeting the current queue results in deadlock.

Unlike with `dispatch_async`, no retain is performed on the target queue. Because calls to this function are synchronous, it "borrows" the reference of the caller. Moreover, no `Block_copy` is performed on the block.

As an optimization, this function invokes the block on the current thread when possible.

Availability
Available in OS X v10.6 and later.

Declared in
`dispatch/queue.h`

dispatch_sync_f

Submits an application-defined function for synchronous execution on a dispatch queue.

```
void dispatch_sync_f(
dispatch_queue_t queue,
void *context,
dispatch_function_t work);
```

Parameters

`queue`
    The queue on which to submit the function. This parameter cannot be `NULL`.

`context`
    The application-defined context parameter to pass to the function.

`work`
    The application-defined function to invoke on the target queue. The first parameter passed to this function is the value in the `context` parameter. This parameter cannot be `NULL`.

Availability
Available in OS X v10.6 and later.

Declared in
`dispatch/queue.h`

See Also
`dispatch_sync` (page 70)

dispatch_time

Creates a `dispatch_time_t` relative to the default clock or modifies an existing `dispatch_time_t`.

```
dispatch_time_t dispatch_time(
dispatch_time_t when,
int64_t delta);
```

Parameters

`when`

The `dispatch_time_t` (page 78) value to use as the basis for a new value. Pass `DISPATCH_TIME_NOW` (page 87) to create a new time value relative to now.

`delta`

The number of nanoseconds to add to the time in the `when` parameter.

Return Value

A new `dispatch_time_t`.

Discussion

The default clock is based on `mach_absolute_time`.

Availability

Available in OS X v10.6 and later.

Declared in

`dispatch/time.h`

dispatch_walltime

Creates a `dispatch_time_t` using an absolute time according to the wall clock.

```
dispatch_time_t dispatch_walltime(
const struct timespec *when,
int64_t delta);
```

Parameters

`when`

A `struct timespec` to add time to. If `NULL` is passed, then this function uses the result of `gettimeofday`.

`delta`

Nanoseconds to add.

Return Value

A new `dispatch_time_t`.

Discussion

The wall clock is based on `gettimeofday`.

Availability

Available in OS X v10.6 and later.

Declared in

`dispatch/time.h`

dispatch_write

Schedule an asynchronous write operation using the specified file descriptor.

```
void dispatch_write(
   dispatch_fd_t fd,
   dispatch_data_t data,
   dispatch_queue_t queue,
   void (^handler)(dispatch_data_t data, int error));
```

Parameters

`fd`

The file descriptor to use when writing the data.

`data`

The data to write to the file descriptor.

`queue`

The queue on which to execute the specified handler block.

`handler`

The block to schedule for execution once the specified data has been written to the file descriptor. The parameters of the handler are as follows:

`data` - The data that could not be written to the file descriptor. If the data was written successfully, this parameter is `NULL`.

`error` - This value is `0` if the data was written successfully. If an error occurred, this parameter contains the error number.

Discussion

This is a convenience function for initiating a single, asynchronous write operation at the current position of the specified file descriptor. This method is intended for simple operations where you do not need the overhead of creating a channel and do not plan on issuing more than a few calls to read or write data. Once submitted, there is no way to cancel the write operation.

After calling this function, the system takes control of the specified file descriptor until the handler block is enqueued. While it controls the file descriptor, the system may modify it on behalf of the application. For example, the system typically adds the `O_NONBLOCK` flag to ensure that any operations are non-blocking. During that time, it is an error for your application to modify the file descriptor directly. However, you may pass the file descriptor to this function or the `dispatch_read` (page 51) function. You may also use the file descriptor to create a new dispatch I/O channel. The system relinquishes control of the file descriptor before calling your handler, so it is safe to modify the file descriptor again from your handler code.

The `handler` you provide is not queued for execution until the write operation finishes. In addition, if you issue multiple read or write calls for the same file descriptor using the convenience APIs, all of those operations must complete before any of the associated handlers are queued. If you are already using the file descriptor with another channel, you should use the `dispatch_io_write` (page 45) function to write data to the channel rather than use this function.

Availability
Available in OS X v10.7 and later.

Declared in
`dispatch/io.h`

Data Types

dispatch_block_t

The prototype of blocks submitted to dispatch queues, which take no arguments and have no return value.

```
typedef void (^dispatch_block_t)(
   void);
```

Discussion
The declaration of a block allocates storage on the stack. Therefore, this example demonstrates an invalid construct:

```
dispatch_block_t block;

if (x) {
    block = ^{printf("true\n"); };
} else {
    block = ^{printf("false\n"); };
}
block();  // unsafe!!
```

What is happening behind the scenes:

```
if (x) {
    struct Block __tmp_1 = ...;  // setup details
    block = &__tmp_1;
}  else {
    struct Block __tmp_2 = ...; // setup details
    block = &__tmp_2;
}
```

As the example demonstrates, the address of a stack variable is escaping the scope in which it is allocated.

Availability
Available in OS X v10.6 and later.

Declared in
`dispatch/queue.h`

dispatch_group_t

A group of block objects submitted to a queue for asynchronous invocation.

```
typedef struct dispatch_group_s *dispatch_group_t;
```

Discussion
A dispatch group is a mechanism for monitoring a set of blocks. Your application can monitor the blocks in the group synchronously or asynchronously depending on your needs. By extension, a group can be useful for synchronizing for code that depends on the completion of other tasks.

Note that the blocks in a group may be run on different queues, and each individual block can add more blocks to the group.

The dispatch group keeps track of how many blocks are outstanding, and GCD retains the group until all its associated blocks complete execution.

Availability
Available in OS X v10.6 and later.

Declared in
`dispatch/group.h`

dispatch_object_t

A polymorphic object type for use with for use with all GCD dispatch object functions.

```
typedef union {
   struct dispatch_object_s *_do;
   struct dispatch_continuation_s *_dc;
   struct dispatch_queue_s *_dq;
   struct dispatch_queue_attr_s *_dqa;
   struct dispatch_group_s *_dg;
   struct dispatch_source_s *_ds;
   struct dispatch_source_attr_s *_dsa;
   struct dispatch_semaphore_s *_dsema;
   struct dispatch_data_s *_ddata;
   struct dispatch_io_s *_dchannel;
   struct dispatch_operation_s *_doperation;
   struct dispatch_fld_s *_dfld;
} dispatch_object_t __attribute__((transparent_union));
```

Discussion
Dispatch objects share functions for coordinating memory management, suspension, cancellation and context pointers. Objects returned by creation functions in the dispatch framework may be uniformly retained and released with the `dispatch_retain` and `dispatch_release` functions, respectively. GCD does not guarantee that any given client has the last or only reference to a given object. Objects may be retained internally by the system.

Note: The complex declaration above is from the header file, but for the sake of simplicity, you might consider it as follows:

typedef void *dispatch_object_t;

Availability
Available in OS X v10.6 and later.

Declared in
`dispatch/base.h`

dispatch_once_t

A predicate for use with the `dispatch_once` function.

```
typedef long dispatch_once_t;
```

Discussion

Variables of this type must have global or static scope. The result of using this type with automatic or dynamic allocation is undefined. See `dispatch_once` (page 47) for details.

Availability

Available in OS X v10.6 and later.

Declared in

`dispatch/once.h`

dispatch_queue_t

A dispatch queue is a lightweight object to which your application submits blocks for subsequent execution.

```
typedef struct dispatch_queue_s *dispatch_queue_t;
```

Discussion

A dispatch queue invokes blocks submitted to it serially in FIFO order. A serial queue invokes only one block at a time, but independent queues may each invoke their blocks concurrently with respect to each other.

The global concurrent queues invoke blocks in FIFO order but do not wait for their completion, allowing multiple blocks to be invoked concurrently.

The system manages a pool of threads that process dispatch queues and invoke blocks submitted to them. Conceptually, a dispatch queue may have its own thread of execution, and interaction between queues is highly asynchronous.

Dispatch queues are reference counted via calls to `dispatch_retain` (page 54) and `dispatch_release` (page 52). Pending blocks submitted to a queue also hold a reference to the queue until they have finished. Once all references to a queue have been released, the queue will be deallocated by the system.

Availability

Available in OS X v10.6 and later.

Declared in

`dispatch/queue.h`

dispatch_time_t

A somewhat abstract representation of time.

```
typedef uint64_t dispatch_time_t;
```

Discussion

For a list of possible values, see "Dispatch Time Constants" (page 87).

Availability

Available in OS X v10.6 and later.

Declared in

`dispatch/time.h`

dispatch_source_type_t

An identifier for the type system object being monitored by a dispatch source.

```
typedef const struct dispatch_source_type_s *dispatch_source_type_t;
```

Discussion

Constants of this type represent the class of low-level system object that is being monitored by the dispatch source. Constants of this type are passed as a parameter to `dispatch_source_create` (page 59) and determine how the handle argument is interpreted (as a file descriptor, mach port, signal number, process identifier, etc.) and how the mask argument is interpreted. See "Dispatch Source Type Constants" (page 86) for details about source type constants.

Availability

Available in OS X v10.6 and later.

dispatch_fd_t

A file descriptor used for I/O operations.

```
typedef int dispatch_fd_t;
```

Availability

Available in OS X v10.7 and later.

dispatch_data_t

An immutable object representing a contiguous or sparse region of memory.

```
typedef dispatch_data_s *dispatch_data_t;
```

Discussion

Any direct access to the memory in a data object must not modify that memory.

Availability

Available in OS X v10.7 and later.

dispatch_data_applier_t

A block to invoke for every contiguous memory region in a data object.

```
typedef bool (^dispatch_data_applier_t)(dispatch_data_t region,
   size_t offset,
   const void *buffer,
   size_t size);
```

Discussion

The parameters of a dispatch data applier block are as follows:

`region` - A data object containing the current memory region being analyzed.

`offset` - The logical offset to the current region from the start of the data object.

`buffer` - A pointer to the memory for the current region.

`size` - The size of the memory for the current region.

This handler returns a Boolean value indicating whether traversal of the region should continue.

Availability

Available in OS X v10.7 and later.

dispatch_io_t

A dispatch I/O channel.

```
typedef dispatch_io_s *dispatch_io_t;
```

Discussion
A dispatch I/O channel represents a file descriptor and the asynchronous I/O policies applied to that file descriptor. A dispatch I/O channel is a standard type of dispatch object and may be retained, released, suspended, and resumed accordingly.

Availability
Available in OS X v10.7 and later.

dispatch_io_handler_t

A handler block used to process operations on a dispatch I/O channel.

```
typedef void (^dispatch_io_handler_t)(bool done, dispatch_data_t data, int error);
```

Discussion
The parameters of a dispatch I/O handler are as follows:

`done` - A flag indicating whether the operation is complete.

`data` - The data object to be handled. This object is retained by the system for the duration of the handler's execution and is released when the handler block returns.

`error` - The error number (if any) reported for the operation. An error number of `0` typically indicates the operation was successful.

Availability
Available in OS X v10.7 and later.

dispatch_io_type_t

The type of a dispatch I/O channel

```
typedef unsigned long dispatch_io_type_t;
```

Availability
Available in OS X v10.7 and later.

dispatch_io_close_flags_t

The type for flags used to specify closing options for a channel.

```
typedef unsigned long dispatch_io_close_flags_t;
```

Availability

Available in OS X v10.7 and later.

dispatch_io_interval_flags_t

The type for flags used to specify the dispatch interval of a channel.

```
typedef unsigned long dispatch_io_interval_flags_t;
```

Availability

Available in OS X v10.7 and later.

Constants

dispatch_queue_priority_t

Used to select the appropriate global concurrent queue.

```
#define DISPATCH_QUEUE_PRIORITY_HIGH        2
#define DISPATCH_QUEUE_PRIORITY_DEFAULT     0
#define DISPATCH_QUEUE_PRIORITY_LOW         (-2)
#define DISPATCH_QUEUE_PRIORITY_BACKGROUND  INT16_MIN
```

Constants

`DISPATCH_QUEUE_PRIORITY_HIGH`

Items dispatched to the queue run at high priority; the queue is scheduled for execution before any default priority or low priority queue.

`DISPATCH_QUEUE_PRIORITY_DEFAULT`

Items dispatched to the queue run at the default priority; the queue is scheduled for execution after all high priority queues have been scheduled, but before any low priority queues have been scheduled.

`DISPATCH_QUEUE_PRIORITY_LOW`

Items dispatched to the queue run at low priority; the queue is scheduled for execution after all default priority and high priority queues have been scheduled.

`DISPATCH_QUEUE_PRIORITY_BACKGROUND`

Items dispatched to the queue run at background priority; the queue is scheduled for execution after all high priority queues have been scheduled and the system runs items on a thread whose priority is set for background status. Such a thread has the lowest priority and any disk I/O is throttled to minimize the impact on the system.

Available in OS X v10.7 and later.

Availability

Available in iOS 4.0 and later.

Available in OS X v10.6 and later.

Declared in

`dispatch/queue.h`

dispatch_source_mach_send_flags_t

Mach send event flags.

```
enum {
   DISPATCH_MACH_SEND_DEAD = 0x1,
};
```

Constants

`DISPATCH_MACH_SEND_DEAD`

The receive right corresponding to the given send right was destroyed.

Availability

Available in iOS 4.0 and later.

Available in OS X v10.6 and later.

Declared in

`dispatch/source.h`

dispatch_source_proc_flags_t

Process event flags.

```
enum {
   DISPATCH_PROC_EXIT = 0x80000000,
   DISPATCH_PROC_FORK = 0x40000000,
   DISPATCH_PROC_EXEC = 0x20000000,
   DISPATCH_PROC_SIGNAL = 0x08000000,
};
```

Constants

`DISPATCH_PROC_EXIT`

The process has exited (perhaps cleanly, perhaps not).

`DISPATCH_PROC_FORK`

The process has created one or more child processes.

`DISPATCH_PROC_EXEC`

The process has become another executable image via an `exec` or `posix_spawn` function family call.

`DISPATCH_PROC_SIGNAL`

A Unix signal was delivered to the process.

Availability

Available in iOS 4.0 and later.

Available in OS X v10.6 and later.

Declared in

`dispatch/source.h`

dispatch_source_vnode_flags_t

File-system object event flags.

```
enum {
   DISPATCH_VNODE_DELETE = 0x1,
   DISPATCH_VNODE_WRITE = 0x2,
   DISPATCH_VNODE_EXTEND = 0x4,
   DISPATCH_VNODE_ATTRIB = 0x8,
   DISPATCH_VNODE_LINK = 0x10,
   DISPATCH_VNODE_RENAME = 0x20,
   DISPATCH_VNODE_REVOKE = 0x40,
};
```

Constants

`DISPATCH_VNODE_DELETE`

The file-system object was deleted from the namespace.

`DISPATCH_VNODE_WRITE`

The file-system object data changed.

`DISPATCH_VNODE_EXTEND`
: The file-system object changed in size.

`DISPATCH_VNODE_ATTRIB`
: The file-system object metadata changed.

`DISPATCH_VNODE_LINK`
: The file-system object link count changed.

`DISPATCH_VNODE_RENAME`
: The file-system object was renamed in the namespace.

`DISPATCH_VNODE_REVOKE`
: The file-system object was revoked.

Availability

Available in iOS 4.0 and later.

Available in OS X v10.6 and later.

Declared in

`dispatch/source.h`

Data Object Constants

Constants representing data objects.

```
#define dispatch_data_empty
```

Constants

`dispatch_data_empty`
: A data object representing a zero-length memory region.

Availability

Available in iOS 5.0 and later.

Available in OS X v10.7 and later.

Data Destructor Constants

Constants representing the destructors to use for data objects.

```
#define DISPATCH_DATA_DESTRUCTOR_DEFAULT NULL
#define DISPATCH_DATA_DESTRUCTOR_FREE
```

Constants

`DISPATCH_DATA_DESTRUCTOR_DEFAULT`

The default data destructor for dispatch objects.

`DISPATCH_DATA_DESTRUCTOR_FREE`

The destructor for dispatch data objects whose memory buffer was created using the `malloc` family of allocation routines.

Availability

Available in iOS 5.0 and later.

Available in OS X v10.7 and later.

Dispatch Queue Types

Attributes to use when creating new dispatch queues.

```
#define DISPATCH_QUEUE_SERIAL
#define DISPATCH_QUEUE_CONCURRENT
```

Constants

`DISPATCH_QUEUE_SERIAL`

A dispatch queue that executes blocks serially in FIFO order.

`DISPATCH_QUEUE_CONCURRENT`

A dispatch queue that executes blocks concurrently. Although they execute blocks concurrently, you can use barrier blocks to create synchronization points within the queue.

Discussion

You use these constants with the `dispatch_queue_create` (page 48) function to specify the type of dispatch queue you want to create. Pass one of these constants for the `attr` parameter of that function.

Availability

Available in iOS 5.0 and later.

Available in OS X v10.7 and later.

Dispatch Source Type Constants

Types of dispatch sources.

```
#define DISPATCH_SOURCE_TYPE_DATA_ADD
#define DISPATCH_SOURCE_TYPE_DATA_OR
#define DISPATCH_SOURCE_TYPE_MACH_RECV
#define DISPATCH_SOURCE_TYPE_MACH_SEND
#define DISPATCH_SOURCE_TYPE_PROC
#define DISPATCH_SOURCE_TYPE_READ
#define DISPATCH_SOURCE_TYPE_SIGNAL
#define DISPATCH_SOURCE_TYPE_TIMER
#define DISPATCH_SOURCE_TYPE_VNODE
#define DISPATCH_SOURCE_TYPE_WRITE
```

Constants

`DISPATCH_SOURCE_TYPE_DATA_ADD`

A dispatch source that coalesces data obtained via calls to `dispatch_source_merge_data` (page 63). An ADD is used to coalesce the data. The handle is unused (pass zero for now). The mask is unused (pass zero for now).

`DISPATCH_SOURCE_TYPE_DATA_OR`

A dispatch source that coalesces data obtained via calls to `dispatch_source_merge_data` (page 63). A logical OR is used to coalesce the data. The handle is unused (pass zero for now). The mask is used to perform a logical AND with the value passed to `dispatch_source_merge_data`.

`DISPATCH_SOURCE_TYPE_MACH_RECV`

A dispatch source that monitors a Mach port for pending messages. The handle is a Mach port with a receive right (`mach_port_t`). The mask is unused (pass zero for now).

`DISPATCH_SOURCE_TYPE_MACH_SEND`

A dispatch source that monitors a Mach port for dead name notifications (the send right no longer has any corresponding receive right). The handle is a Mach port with a send or send-once right (`mach_port_t`). The mask is a mask of desired events from "`dispatch_source_mach_send_flags_t`" (page 82).

`DISPATCH_SOURCE_TYPE_PROC`

A dispatch source that monitors an external process for events defined by "`dispatch_source_proc_flags_t`" (page 82). The handle is a process identifier (`pid_t`). The mask is a mask of desired events from "`dispatch_source_proc_flags_t`".

`DISPATCH_SOURCE_TYPE_READ`

A dispatch source that monitors a file descriptor for pending bytes available to be read. The handle is a file descriptor (`int`). The mask is unused (pass zero for now).

`DISPATCH_SOURCE_TYPE_SIGNAL`
: A dispatch source that monitors the current process for signals. The handle is a signal number (`int`). The mask is unused (pass zero for now).

`DISPATCH_SOURCE_TYPE_TIMER`
: A dispatch source that submits the event handler block based on a timer. The handle is unused (pass zero for now). The mask is unused (pass zero for now).

`DISPATCH_SOURCE_TYPE_VNODE`
: A dispatch source that monitors a file descriptor for events defined by “`dispatch_source_vnode_flags_t`” (page 83). The handle is a file descriptor (`int`). The mask is a mask of desired events from “`dispatch_source_vnode_flags_t`”.

`DISPATCH_SOURCE_TYPE_WRITE`
: A dispatch source that monitors a file descriptor for available buffer space to write bytes. The handle is a file descriptor (int). The mask is unused (pass zero for now).

Availability

Available in iOS 4.0 and later.

Available in OS X v10.6 and later.

Declared in

`dispatch/source.h`

Dispatch Time Constants

Base time constants.

```
#define DISPATCH_TIME_NOW 0
#define DISPATCH_TIME_FOREVER (~0ull)
```

Constants

`DISPATCH_TIME_NOW`
: Indicates a time that occurs immediately.

`DISPATCH_TIME_FOREVER`
: Indicates a time that means infinity.

Availability

Available in iOS 4.0 and later.

Available in OS X v10.6 and later.

Time Multiplier Constants

Multipliers for calculating time values.

```
#define NSEC_PER_SEC 1000000000ull
#define USEC_PER_SEC 1000000ull
#define NSEC_PER_USEC 1000ull
```

Constants

`NSEC_PER_SEC`

The number of nanoseconds in one second.

Available in OS X v10.0 and later.

Declared in `clock_types.h`.

`USEC_PER_SEC`

The number of microseconds in one second.

Available in OS X v10.0 and later.

Declared in `clock_types.h`.

`NSEC_PER_USEC`

The number of nanoseconds in one microsecond.

Available in OS X v10.0 and later.

Declared in `clock_types.h`.

Availability

Available in iOS 4.0 and later.

Available in OS X v10.6 and later.

Dispatch I/O Channel Types

The types of dispatch I/O channels that may be created.

```
#define DISPATCH_IO_STREAM 0
#define DISPATCH_IO_RANDOM 1
```

Constants

`DISPATCH_IO_STREAM`

The channel represents a linear stream of bytes. Read and write operations are performed serially in the order they were started. Operations always read or write data at the file pointer position that is current when the read or write begins. Read and write operations may be performed simultaneously on the same channel.

Offset values are ignored for channels of this type.

`DISPATCH_IO_RANDOM`

The channel represents a random access file. Read and write operations may be performed concurrently with a channel of this type. Offsets are interpreted relative to the file pointer position that is current at the time the channel is created. After channel creation, the file pointer position of the file descriptor is indeterminate until channel relinquishes control of the file descriptor, at which time the position is reset to its initial value.

The file descriptor for a channel of this type must be seekable. If it is not, attempting to create a channel of this type for the descriptor will result in an error.

Availability

Available in iOS 5.0 and later.

Available in OS X v10.7 and later.

Channel Closing Options

The options to use when closing a dispatch I/O channel.

```
#define DISPATCH_IO_STOP 0x1
```

Constants

`DISPATCH_IO_STOP`

Stop any in-progress read and write operations.

Availability

Available in iOS 5.0 and later.

Available in OS X v10.7 and later.

Channel Configuration Options

The options to use when configuring a channel.

```
#define DISPATCH_IO_STRICT_INTERVAL 0x1
```

Constants

`DISPATCH_IO_STRICT_INTERVAL`

Enqueue handlers for a channel at strict intervals regardless of how much data has been read or written. Setting this flag can lead to the handler being called even if the amount of data does not exceed the channel's low-water mark.

Availability

Available in iOS 5.0 and later.

Available in OS X v10.7 and later.

Document Revision History

This table describes the changes to *Grand Central Dispatch (GCD) Reference* .

Date	Notes
2011-11-18	Added iOS availability information.
2011-10-12	Added new symbols introduced in OS X v10.7 and iOS 5.
2010-05-11	Updated to reflect support for iOS.
2010-02-01	Fixed several typographical errors and added some missing constants.
2009-08-19	New document that describes the Grand Central Dispatch API for efficient use of multicore systems.