Grand Central Dispatch (GCD) Reference

Performance

2009-08-19

Contents

Grand Central Dispatch (GCD) Reference

Declared in	dispatch/dispatch.h
Companion guide	Concurrency Programming Guide

Overview

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

The BSD subsystem, CoreFoundation, and Cocoa APIs have all been extended to use these enhancements to help both Mac OS X 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. You should read this document if your application needs to operate at the Unix level of Mac OS X—for example, if it needs to manipulate file descriptors, Mach ports, signals, or timers. 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

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 will execute. GCD offers three kinds of queues:

- **Main:** tasks execute serially on your application’s main thread
- **Concurrent:** tasks start executing in FIFO order, but can run concurrently.
- **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`
- calling `NSApplicationMain`
- using a `CFRunLoop` 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` 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.

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_after` (page 9)
  Schedule a block for execution on a specified queue at a specified time.

`dispatch_after_f` (page 10)
  Schedules an application-defined function for execution on a specified queue at a specified time.

`dispatch_apply` (page 10)
  Submits a block to a dispatch queue for multiple invocations.

`dispatch_apply_f` (page 11)
  Submits an application-defined function to a dispatch queue for multiple invocations.

`dispatch_async` (page 12)
  Submits a block for asynchronous execution on a dispatch queue and returns immediately.

`dispatch_async_f` (page 12)
  Submits an application-defined function for asynchronous execution on a dispatch queue and returns immediately.

`dispatch_get_current_queue` (page 14)
  Returns the queue on which the currently executing block is running.

`dispatch_get_global_queue` (page 14)
  Returns a well-known global concurrent queue of a given priority level.

`dispatch_get_main_queue` (page 15)
  Returns the serial dispatch queue associated with the application's main thread.

`dispatch_main` (page 20)
  Executes blocks submitted to the main queue.

`dispatch_once` (page 21)
  Executes a block object once and only once for the lifetime of an application.

`dispatch_queue_create` (page 21)
: Creates a new dispatch queue to which blocks can be submitted.

`dispatch_queue_get_label` (page 22)
: Returns the label specified for the queue when the queue was created.

`dispatch_set_target_queue` (page 26)
: Sets the target queue for the given object.

`dispatch_sync` (page 35)
: Submits a block object for execution on a dispatch queue and waits synchronously until that block completes.

`dispatch_sync_f` (page 36)
: Submits an application-defined function for synchronous execution on a dispatch queue.

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 16)
: Submits a block to a dispatch queue and associates the block with the specified dispatch group.

`dispatch_group_async_f` (page 16)
: Submits an application-defined function to a dispatch queue and associates it with the specified dispatch group.

`dispatch_group_create` (page 17)
: Creates a new group with which block objects can be associated.

`dispatch_group_enter` (page 17)
: Explicitly indicates that a block has entered the group.

`dispatch_group_leave` (page 18)
: Explicitly indicates that a block in the group has completed.

`dispatch_group_notify` (page 18)
: 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 19)
: 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 20)
: 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_debug` (page 13)
: Programmatically logs debug information about a dispatch object.

`dispatch_get_context` (page 14)
: Returns the application-defined context of an object.

`dispatch_release` (page 23)
: Decrements the reference (retain) count of a dispatch object.

`dispatch_resume` (page 23)
: Resume the invocation of block objects on a dispatch object.

`dispatch_retain` (page 24)
: Increments the reference (retain) count of a dispatch object.

`dispatch_set_context` (page 25)
: Associates an application-defined context with the object.

`dispatch_set_finalizer_f` (page 26)
: Sets the finalizer function for a dispatch object.

`dispatch_suspend` (page 34)
: 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 24)
: Creates new counting semaphore with an initial value.

`dispatch_semaphore_signal` (page 25)
: Signals (increments) a semaphore.

`dispatch_semaphore_wait` (page 25)
: Waits for (decrements) a semaphore.

Handling Events

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 27)
: Asynchronously cancels the dispatch source, preventing any further invocation of its event handler block.

`dispatch_source_create` (page 28)
: 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 28)
: Returns pending data for the dispatch source.

`dispatch_source_get_handle` (page 29)
: Returns the underlying system handle associated with the specified dispatch source.

`dispatch_source_get_mask` (page 30)
: Returns the mask of events monitored by the dispatch source.

`dispatch_source_merge_data` (page 30)
: 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_cancel_handler` (page 31)
: Sets the cancellation handler block for the given dispatch source.

`dispatch_source_set_cancel_handler_f` (page 32)
: Sets the cancellation handler function for the given dispatch source.

`dispatch_source_set_event_handler` (page 32)
: Sets the event handler block for the given dispatch source.

`dispatch_source_set_event_handler_f` (page 33)
: Sets the event handler function for the given dispatch source.

`dispatch_source_set_timer` (page 33)
: Sets a start time, interval, and leeway value for a timer source.

`dispatch_source_testcancel` (page 34)
: Tests whether the given dispatch source has been canceled.

Managing Time

`dispatch_time` (page 36)
: Creates a `dispatch_time_t` relative to the default clock or modifies an existing `dispatch_time_t`.

`dispatch_walltime` (page 37)
: Creates a `dispatch_time_t` using an absolute time according to the wall clock.

Functions

dispatch_after

Schedule a block for execution on a specified queue at a 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 to which the block is submitted at the specified time. 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
Passing `DISPATCH_TIME_NOW` as the `when` parameter is supported, but is not as optimal as calling `dispatch_async` instead. Passing `DISPATCH_TIME_FOREVER` is undefined.

Availability
Available in Mac OS X v10.6.

Declared In
`dispatch/queue.h`

dispatch_after_f

Schedules an application-defined function for execution on a specified queue at a specified time.

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

Parameters

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

queue
: The queue to which the block is submitted at the specified time. 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 context provided to `dispatch_after_f`. This parameter cannot be NULL.

Discussion
Passing `DISPATCH_TIME_NOW` as the `when` parameter is supported, but is not as optimal as calling `dispatch_async` instead. Passing `DISPATCH_TIME_FOREVER` is undefined.

Availability
Available in Mac OS X v10.6.

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 target dispatch queue to which the block is submitted. 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_concurrent_queue`, the block can be invoked concurrently, and it must therefore be reentrant-safe. Using `dispatch_apply` 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 Mac OS X v10.6.

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 target dispatch queue to which the block is submitted. 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 context provided to `dispatch_apply_f`. The second parameter passed to this function 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 task block to complete before returning. If the target queue is a concurrent queue returned by `dispatch_get_concurrent_queue`, the function can be invoked concurrently, and it must therefore be reentrant-safe. Using `dispatch_apply` 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 Mac OS X v10.6.

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 target dispatch queue to which the block is submitted. 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
The `dispatch_async` function is the fundamental mechanism for submitting blocks to a dispatch queue. Calls to `dispatch_async` 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 will be 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 Mac OS X v10.6.

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 target dispatch queue to which the function is submitted. The queue is retained by the system until the block 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 context provided to `dispatch_async_f`. This parameter cannot be NULL.

Discussion

The `dispatch_async_f` function is the fundamental mechanism for submitting application-defined functions to a dispatch queue. Calls to `dispatch_async_f` always return immediately after the function has been submitted and never wait for it to be invoked. The target queue determines whether the function will be 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 Mac OS X v10.6.

Declared In

`dispatch/queue.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` function is called.

Availability

Available in Mac OS X v10.6.

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 Mac OS X v10.6.

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 `dispatch_get_current_queue` is called outside of the context of a submitted block, it returns the default concurrent queue.

Availability
Available in Mac OS X v10.6.

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_concurrent_queue(
    long priority,
    unsigned long flags);
```

Parameters

priority

A priority defined in `dispatch_queue_priority_t`. See "dispatch_queue_priority_t" (page 41) for more information.

Return Value

Returns the requested global queue.

Discussion

The well-known global concurrent queues cannot be modified. Calls to `dispatch_suspend`, `dispatch_resume`, `dispatch_set_context`, 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 Mac OS X v10.6.

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`
- calling `NSApplicationMain`
- using a `CFRunLoop` on the main thread

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

Availability

Available in Mac OS X v10.6.

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 will be 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 Mac OS X v10.6.

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 will be 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 context provided to `dispatch_group_async_f`.

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 Mac OS X v10.6.

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` 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` and `dispatch_group_wait` 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` to release its reference to the group object and ultimately free its memory.

Availability

Available in Mac OS X v10.6.

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`) 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` function. A call to this function must be balanced with a call to `dispatch_group_leave`.

Availability
Available in Mac OS X v10.6.

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`) 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` 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 Mac OS X v10.6.

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` or be reused for additional block objects. See `dispatch_group_async` for more information.

Availability

Available in Mac OS X v10.6.

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 context provided to `dispatch_group_notify_f`.

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` or be reused for additional blocks. See `dispatch_group_async` for more information.

Availability
Available in Mac OS X v10.6.

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`). The `DISPATCH_TIME_NOW` and `DISPATCH_TIME_FOREVER` 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` or be reused for additional blocks. See `dispatch_group_async` for more information.

Availability
Available in Mac OS X v10.6.

Declared In
`dispatch/group.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 `NSApplicationMain` or `CFRunLoopRun` on the main thread must not call `dispatch_main`.

Availability
Available in Mac OS X v10.6.

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` 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 `dispatch_once` 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 Mac OS X v10.6.

Declared In
`dispatch/once.h`

dispatch_queue_create

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

```
dispatch_queue_attr_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

Currently unused; pass NULL.

Return Value

The newly created dispatch queue.

Returns a newly allocated serial queue.

Discussion

Blocks submitted to the 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.

When your application no longer needs the dispatch queue, it should release it with the `dispatch_release` 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 Mac OS X v10.6.

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 Mac OS X v10.6.

Declared In

`dispatch/queue.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 Mac OS X v10.6.

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 `dispatch_resume` 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`. New dispatch event source objects returned by `dispatch_source_create` 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 Mac OS X v10.6.

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 `dispatch_retain` must be balanced with calls to `dispatch_release`. 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 Mac OS X v10.6.

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 will cause 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` to release its reference to the semaphore object and ultimately free its memory.

Availability

Available in Mac OS X v10.6.

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`.

Availability
Available in Mac OS X v10.6.

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`). The constants `DISPATCH_TIME_NOW` and `DISPATCH_TIME_FOREVER` 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 Mac OS X v10.6.

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 Mac OS X v10.6.

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`. 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 will not be called if the application-defined context is NULL.

Availability

Available in Mac OS X v10.6.

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` 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 will be submitted.

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

Availability

Available in Mac OS X v10.6.

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.

See `dispatch_source_set_cancel_handler` for more information.

Availability
Available in Mac OS X v10.6.

Declared In
`dispatch/source.h`

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,
  dispatch_source_attr_t attr,
  unsigned long mask,
  dispatch_queue_t queue);
```

Parameters

type
: The type of the dispatch source. Must be one of the defined `dispatch_source_type_t` constants.

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.

Discussion
Dispatch sources are not reentrant. Any events received while the dispatch source is suspended or while the event handler block is currently executing will be 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` to begin event delivery.

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

Availability
Available in Mac OS X v10.6.

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`: application-defined data
- `DISPATCH_SOURCE_TYPE_DATA_OR`: application-defined data
- `DISPATCH_SOURCE_TYPE_MACH_SEND`: `dispatch_source_mach_send_flags_t`
- `DISPATCH_SOURCE_TYPE_PROC`: `dispatch_source_proc_flags_t`
- `DISPATCH_SOURCE_TYPE_READ`: estimated bytes available to read
- `DISPATCH_SOURCE_TYPE_SIGNAL`: number of signals delivered since the last handler invocation
- `DISPATCH_SOURCE_TYPE_TIMER`: 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 Mac OS X v10.6.

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`: mach port (`mach_port_t`)
- `DISPATCH_SOURCE_TYPE_MACH_RECV`: mach port (`mach_port_t`)
- `DISPATCH_SOURCE_TYPE_PROC`: process identifier (`pid_t`)
- `DISPATCH_SOURCE_TYPE_READ`: file descriptor (`int`)
- `DISPATCH_SOURCE_TYPE_SIGNAL`: signal number (`int`)

- `DISPATCH_SOURCE_TYPE_VNODE`: file descriptor (`int`)
- `DISPATCH_SOURCE_TYPE_WRITE`: file descriptor (`int`)

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

Availability
Available in Mac OS X v10.6.

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`: `dispatch_source_mach_send_flags_t`
- `DISPATCH_SOURCE_TYPE_PROC`: `dispatch_source_proc_flags_t`
- `DISPATCH_SOURCE_TYPE_VNODE`: `dispatch_source_vnode_flags_t`

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 will be ignored and no event handler block will be submitted for them.

For details, see the flag descriptions in "Constants" (page 41).

Availability
Available in Mac OS X v10.6.

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 will 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` or `DISPATCH_SOURCE_TYPE_DATA_OR`.

Availability

Available in Mac OS X v10.6.

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` 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 Mac OS X v10.6.

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` 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 Mac OS X v10.6.

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 Mac OS X v10.6.

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 Mac OS X v10.6.

Declared In
`dispatch/source.h`

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` and `dispatch_walltime` 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 will be used for the timer. If the start time is `DISPATCH_TIME_NOW` or is created with `dispatch_time`, the timer is based on `mach_absolute_time`. Otherwise, if the start time of the timer is created with `dispatch_walltime`, 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 Mac OS X v10.6.

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`. The result of this function is non-zero immediately after `dispatch_source_cancel` has been called.

Availability

Available in Mac OS X v10.6.

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 object to be suspended. 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 `dispatch_suspend` increments the suspension count of the object, and calling `dispatch_resume` 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 will be delivered once the object is resumed.

Availability

Available in Mac OS X v10.6.

Declared In

`dispatch/object.h`

dispatch_sync

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

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

Parameters

queue

The target dispatch queue to which the block is submitted. 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`, `dispatch_sync` does not return until the block has finished.

Calls to `dispatch_sync` targeting the current queue result in deadlock because the submitted block is not invoked until the current block has completed, but `dispatch_sync` does not return until the submitted block has completed.

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

As an optimization, `dispatch_sync` invokes the block on the current thread when possible.

Availability

Available in Mac OS X v10.6.

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 target dispatch queue to which the block is submitted. This parameter cannot be NULL.

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

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

Discussion
See `dispatch_sync`.

Availability
Available in Mac OS X v10.6.

Declared In
`dispatch/queue.h`

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`
An optional `dispatch_time_t` to add nanoseconds to. If you pass `DISPATCH_TIME_NOW`, `dispatch_time` will use the result of `mach_absolute_time`.

`delta`
Nanoseconds to add.

Return Value
A new `dispatch_time_t`.

Discussion
On Mac OS X the default clock is based on `mach_absolute_time`.

Availability
Available in Mac OS X v10.6.

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 `dispatch_walltime` will use the result of `gettimeofday`.

`delta`
: Nanoseconds to add.

Return Value
A new `dispatch_time_t`.

Discussion
On Mac OS X the wall clock is based on `gettimeofday`.

Availability
Available in Mac OS X v10.6.

Declared In
`dispatch/time.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 Mac OS X v10.6.

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 Mac OS X v10.6.

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_event_s *_de;
    struct dispatch_apply_s *_da;
    struct dispatch_semaphore_s *_dsema;
} 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 Mac OS X v10.6.

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 21) for details.

Availability
Available in Mac OS X v10.6.

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` and `dispatch_release`. 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 Mac OS X v10.6.

Declared In
`dispatch/queue.h`

dispatch_time_t

A somewhat abstract representation of time, wherein DISPATCH_TIME_NOW means "now" and DISPATCH_TIME_FOREVER means "infinity" and every value in between is of an opaque encoding.

```
typedef uint64_t dispatch_time_t;
```

Availability
Available in Mac OS X v10.6.

Declared In
`dispatch/time.h`

dispatch_source_type_t

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` 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.

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

Discussion
See "Dispatch Source Type Constants" (page 43) for details about source type constants.

Constants

dispatch_queue_priority_t

Used to select the appropriate global concurrent queue.

```
enum {
    DISPATCH_QUEUE_PRIORITY_HIGH = 2,
    DISPATCH_QUEUE_PRIORITY_DEFAULT = 0,
    DISPATCH_QUEUE_PRIORITY_LOW = -2,
};
```

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.

Availability
Available in Mac OS X v10.6.

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 Mac OS X v10.6.

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 Mac OS X v10.6.

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 Mac OS X v10.6.

Declared In
`dispatch/source.h`

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`. 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`. 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`.

`DISPATCH_SOURCE_TYPE_PROC`
: A dispatch source that monitors an external process for events defined by `dispatch_source_proc_flags_t`. 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`. 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 Mac OS X v10.6.

Declared In
`dispatch/source.h`

Document Revision History

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

Date	Notes
2009-08-19	New document that describes the Grand Central Dispatch API for efficient use of multicore systems.

Index