Dispatch Sources

Whenever you interact with the underlying system, you must be prepared for that task to take a nontrivial amount of time. Calling down to the kernel or other system layers involves a change in context that is reasonably expensive compared to calls that occur within your own process. As a result, many system libraries provide asynchronous interfaces to allow your code to submit a request to the system and continue to do other work while that request is processed. Grand Central Dispatch builds on this general behavior by allowing you to submit your request and have the results reported back to your code using blocks and dispatch queues.

About Dispatch Sources

A *dispatch source* is a fundamental data type that coordinates the processing of specific low-level system events. Grand Central Dispatch supports the following types of dispatch sources:

- *Timer dispatch sources* generate periodic notifications.
- *Signal dispatch sources* notify you when a UNIX signal arrives.
- *Descriptor sources* notify you of various file- and socket-based operations, such as:
    - When data is available for reading
    - When it is possible to write data
    - When files are deleted, moved, or renamed in the file system
    - When file meta information changes
- *Process dispatch sources* notify you of process-related events, such as:
    - When a process exits
    - When a process issues a `fork` or `exec` type of call
    - When a signal is delivered to the process
- *Mach port dispatch sources* notify you of Mach-related events.
- *Custom dispatch sources* are ones you define and trigger yourself.

Dispatch sources replace the asynchronous callback functions that are typically used to process system-related events. When you configure a dispatch source, you specify the events you want to monitor and the dispatch queue and code to use to process those events. You can specify your code using block objects or functions. When an event of interest arrives, the dispatch source submits your block or function to the specified dispatch queue for execution.

Unlike tasks that you submit to a queue manually, dispatch sources provide a continuous source of events for your application. A dispatch source remains attached to its dispatch queue until you cancel it explicitly. While attached, it submits its associated task code to the dispatch queue whenever the corresponding event occurs. Some events, such as timer events, occur at regular intervals but most occur only sporadically as specific conditions arise. For this reason, dispatch sources retain their associated dispatch queue to prevent it from being released prematurely while events may still be pending.

To prevent events from becoming backlogged in a dispatch queue, dispatch sources implement an event coalescing scheme. If a new event arrives before the event handler for a previous event has been dequeued and executed, the dispatch source coalesces the data from the new event data with data from the old event. Depending on the type of event, coalescing may replace the old event or update the information it holds. For example, a signal-based dispatch source provides information about only the most recent signal but also reports how many total signals have been delivered since the last invocation of the event handler.

Creating Dispatch Sources

Creating a dispatch source involves creating both the source of the events and the dispatch source itself. The source of the events is whatever native data structures are required to process the events. For example, for a descriptor-based dispatch source you would need to open the descriptor and for a process-based source you would need to obtain the process ID of the target program. When you have your event source, you can then create the corresponding dispatch source as follows:

1. Create the dispatch source using the `dispatch_source_create` function.
2. Configure the dispatch source:
   - Assign an event handler to the dispatch source; see Writing and Installing an Event Handler.
   - For timer sources, set the timer information using the `dispatch_source_set_timer` function; see Creating a Timer.
3. Optionally assign a cancellation handler to the dispatch source; see Installing a Cancellation Handler.
4. Call the `dispatch_resume` function to start processing events; see Suspending and Resuming Dispatch Sources.

Because dispatch sources require some additional configuration before they can be used, the `dispatch_source_create` function returns dispatch sources in a suspended state. While suspended, a dispatch source receives events but does not process them. This gives you time to install an event handler and perform any additional configuration needed to process the actual events.

The following sections show you how to configure various aspects of a dispatch source. For detailed examples showing you how to configure specific types of dispatch sources, see Dispatch Source Examples. For additional information about the functions you use to create and configure dispatch sources, see *Grand Central Dispatch (GCD) Reference*.

Writing and Installing an Event Handler

To handle the events generated by a dispatch source, you must define an event handler to process those events. An event handler is a function or block object that you install on your dispatch source using the `dispatch_source_set_event_handler` or `dispatch_source_set_event_handler_f` function. When an event arrives, the dispatch source submits your event handler to the designated dispatch queue for processing.

The body of your event handler is responsible for processing any events that arrive. If your event handler is already queued and waiting to process an event when a new event arrives, the dispatch source coalesces the two events. An event handler generally sees information only for the most recent event, but depending on the type of the dispatch source it may also be able to get information about other events that occurred and were coalesced. If one or more new events arrive after the event handler has begun executing, the dispatch source holds onto those events until the current event handler has finished executing. At that point, it submits the event handler to the queue again with the new events.

Function-based event handlers take a single context pointer, containing the dispatch source object, and return no value. Block-based event handlers take no parameters and have no return value.

```
// Block-based event handler
void (^dispatch_block_t)(void)


// Function-based event handler
void (*dispatch_function_t)(void *)
```

Inside your event handler, you can get information about the given event from the dispatch source itself. Although function-based event handlers are passed a pointer to the dispatch source as a parameter, block-based event handlers must capture that pointer themselves. You can do this for your blocks by referencing the variable containing the dispatch source normally. For example, the following code snippet captures the `source` variable, which is declared outside the scope of the block.

```
dispatch_source_t source = dispatch_source_create(DISPATCH_SOURCE_TYPE_READ,
                                 myDescriptor, 0, myQueue);
dispatch_source_set_event_handler(source, ^{
   // Get some data from the source variable, which is captured
   // from the parent context.
   size_t estimated = dispatch_source_get_data(source);

   // Continue reading the descriptor...
});
dispatch_resume(source);
```

Capturing variables inside of a block is commonly done to allow for greater flexibility and dynamism. Of course, captured variables are read-only within the block by default. Although the blocks feature provides support for modifying captured variables under specific circumstances, you should not attempt to do so in the event handlers associated with a dispatch source. Dispatch sources always execute their event handlers asynchronously, so the defining scope of any variables you captured is likely gone by the time your event handler executes. For more information about how to capture and use variables inside of blocks, see *Blocks Programming Topics*.

Table 4-1 lists the functions you can call from your event handler code to obtain information about an event.

Table 4-1 Getting data from a dispatch source

Function	Description
`dispatch_source_get_handle`	This function returns the underlying system data type that the dispatch source manages. For a descriptor dispatch source, this function returns an `int` type containing the descriptor associated with the dispatch source. For a signal dispatch source, this function returns an `int` type containing the signal number for the most recent event. For a process dispatch source, this function returns a `pid_t` data structure for the process being monitored. For a Mach port dispatch source, this function returns a `mach_port_t` data structure. For other dispatch sources, the value returned by this function is undefined.
`dispatch_source_get_data`	This function returns any pending data associated with the event. For a descriptor dispatch source that reads data from a file, this function returns the number of bytes available for reading. For a descriptor dispatch source that writes data to a file, this function returns a positive integer if space is available for writing. For a descriptor dispatch source that monitors file system activity, this function returns a constant indicating the type of event that occurred. For a list of constants, see the `dispatch_source_vnode_flags_t` enumerated type. For a process dispatch source, this function returns a constant indicating the type of event that occurred. For a list of

	constants, see the `dispatch_source_proc_flags_t` enumerated type. For a Mach port dispatch source, this function returns a constant indicating the type of event that occurred. For a list of constants, see the `dispatch_source_machport_flags_t` enumerated type. For a custom dispatch source, this function returns the new data value created from the existing data and the new data passed to the `dispatch_source_merge_data` function.
`dispatch_source_get_mask`	This function returns the event flags that were used to create the dispatch source. For a process dispatch source, this function returns a mask of the events that the dispatch source receives. For a list of constants, see the `dispatch_source_proc_flags_t` enumerated type. For a Mach port dispatch source with send rights, this function returns a mask of the desired events. For a list of constants, see the `dispatch_source_mach_send_flags_t` enumerated type. For a custom OR dispatch source, this function returns the mask used to merge the data values.

For some examples of how to write and install event handlers for specific types of dispatch sources, see Dispatch Source Examples.

Installing a Cancellation Handler

Cancellation handlers are used to clean up a dispatch source before it is released. For most types of dispatch sources, cancellation handlers are optional and only necessary if you have some custom behaviors tied to the dispatch source that also need to be updated. For dispatch sources that use a descriptor or Mach port, however, you must provide a cancellation handler to close the descriptor or release the Mach port. Failure to do so can lead to subtle bugs in your code resulting from those structures being reused unintentionally by your code or other parts of the system.

You can install a cancellation handler at any time but usually you would do so when creating the dispatch source. You install a cancellation handler using the `dispatch_source_set_cancel_handler` or `dispatch_source_set_cancel_handler_f` function, depending on whether you want to use a block object or a function in your implementation. The following example shows a simple cancellation handler that closes a descriptor that was opened for a dispatch source. The `fd` variable is a captured variable containing the descriptor.

```
dispatch_source_set_cancel_handler(mySource, ^{
   close(fd); // Close a file descriptor opened earlier.
});
```

To see a complete code example for a dispatch source that uses a cancellation handler, see Reading Data from a Descriptor.

Changing the Target Queue

Although you specify the queue on which to run your event and cancellation handlers when you create a dispatch source, you can change that queue at any time using the `dispatch_set_target_queue` function. You might do this to change the priority at which the dispatch source's events are processed.

Changing a dispatch source's queue is an asynchronous operation and the dispatch source does its best to make the change as quickly as possible. If an event handler is already queued and waiting to be processed, it executes on the previous queue. However, other events arriving around the time you make the change could be processed on either queue.

Associating Custom Data with a Dispatch Source

Like many other data types in Grand Central Dispatch, you can use the `dispatch_set_context` function to associate custom data with a dispatch source. You can use the context pointer to store any data your event handler needs to process events. If you do store any custom data in the context pointer, you should also install a cancellation handler (as described in Installing a Cancellation Handler) to release that data when the dispatch source is no longer needed.

If you implement your event handler using blocks, you can also capture local variables and use them within your block-based code. Although this might alleviate the need to store data in the context pointer of the dispatch source, you should always use this feature judiciously. Because dispatch sources may be long-lived in your application, you should be careful when capturing variables containing pointers. If the data pointed to by a pointer could be deallocated at any time, you should either copy the data or retain it to prevent that from happening. In either case, you would then need to provide a cancellation handler to release the data later.

Memory Management for Dispatch Sources

Like other dispatch objects, dispatch sources are reference counted data types. A dispatch source has an initial reference count of 1 and can be retained and released using the `dispatch_retain` and `dispatch_release` functions. When the reference count of a queue reaches zero, the system automatically deallocates the dispatch source data structures.

Because of the way they are used, the ownership of dispatch sources can be managed either internally or externally to the dispatch source itself. With external ownership, another object or piece of code takes ownership of the dispatch source and is responsible for releasing it when it is no longer needed. With internal ownership, the dispatch source owns itself and is responsible for releasing itself at the appropriate time. Although external ownership is very common, you might use internal ownership in cases where you want to create an autonomous dispatch source and let it manage some behavior of your code without any further interactions. For example, if a dispatch source is designed to respond to a single global event, you might have it handle that event and then exit immediately.

Dispatch Source Examples

The following sections show you how to create and configure some of the more commonly used dispatch sources. For more information about configuring specific types of dispatch sources, see *Grand Central Dispatch (GCD) Reference*.

Creating a Timer

Timer dispatch sources generate events at regular, time-based intervals. You can use timers to initiate specific tasks that need to be performed regularly. For example, games and other graphics-intensive applications might use timers to initiate screen or animation updates. You could also set up a timer and use the resulting events to check for new information on a frequently updated server.

All timer dispatch sources are interval timers—that is, once created, they deliver regular events at the interval you specify. When you create a timer dispatch source, one of the values you must specify is a leeway value to give the system some idea of the desired accuracy for timer events. Leeway values give the system some flexibility in how it manages power and wakes up cores. For example, the system might use the leeway value to advance or delay the fire time and align it better with other system events. You should therefore specify a leeway value whenever possible for your own timers.

Note: Even if you specify a leeway value of 0, you should never expect a timer to fire at the exact nanosecond you requested. The system does its best to accommodate your needs but cannot guarantee exact firing times.

When a computer goes to sleep, all timer dispatch sources are suspended. When the computer wakes up, those timer dispatch sources are automatically woken up as well. Depending on the configuration of the timer, pauses of this nature may affect when your timer fires next. If you set up your timer dispatch source using the `dispatch_time` function or the `DISPATCH_TIME_NOW` constant, the timer dispatch source uses the default system clock to determine when to fire. However, the default clock does not advance while the computer is asleep. By contrast, when you set up your timer dispatch source using the `dispatch_walltime` function, the timer dispatch source tracks its firing time to the wall clock time. This latter option is typically appropriate for timers whose firing interval is relatively large because it prevents there from being too much drift between event times.

Listing 4-1 shows an example of a timer that fires once every 30 seconds and has a leeway value of 1 second. Because the timer interval is relatively large, the dispatch source is created using the `dispatch_walltime` function. The first firing of the timer occurs immediately and subsequent events arrive every 30 seconds. The `MyPeriodicTask` and `MyStoreTimer` symbols represent custom functions that you would write to implement the timer behavior and to store the timer somewhere in your application's data structures.

Listing 4-1 Creating a timer dispatch source

```
dispatch_source_t CreateDispatchTimer(uint64_t interval,
              uint64_t leeway,
              dispatch_queue_t queue,
              dispatch_block_t block)
{
   dispatch_source_t timer = dispatch_source_create(DISPATCH_SOURCE_TYPE_TIMER,
                                                     0, 0, queue);
   if (timer)
   {
      dispatch_source_set_timer(timer, dispatch_walltime(NULL, 0), interval,
leeway);
      dispatch_source_set_event_handler(timer, block);
      dispatch_resume(timer);
   }
   return timer;
}

void MyCreateTimer()
{
   dispatch_source_t aTimer = CreateDispatchTimer(30ull * NSEC_PER_SEC,
                               1ull * NSEC_PER_SEC,
                               dispatch_get_main_queue(),
                               ^{ MyPeriodicTask(); });

   // Store it somewhere for later use.
    if (aTimer)
    {
        MyStoreTimer(aTimer);
    }
}
```

Although creating a timer dispatch source is the main way to receive time-based events, there are other options available as well. If you want to perform a block once after a specified time interval, you can use the `dispatch_after` or `dispatch_after_f` function. This function behaves much like the `dispatch_async` function except that it allows you to specify a time value at which to submit the block to a queue. The time value can be specified as a relative or absolute time value depending on your needs.

Reading Data from a Descriptor

To read data from a file or socket, you must open the file or socket and create a dispatch source of type `DISPATCH_SOURCE_TYPE_READ`. The event handler you specify should be capable of reading and processing the contents of the file descriptor. In the case of a file, this amounts to reading the file data (or a subset of that data) and creating the appropriate data structures for your application. For a network socket, this involves processing newly received network data.

Whenever reading data, you should always configure your descriptor to use non-blocking operations. Although you can use the `dispatch_source_get_data` function to see how much data is available for reading, the number returned by that function could change between the time you make the call and the time you actually read the data. If the underlying file is truncated or a network error occurs, reading from a descriptor that blocks the current thread could stall your event handler in mid execution and prevent the dispatch queue from dispatching other tasks. For a serial queue, this could deadlock your queue, and even for a concurrent queue this reduces the number of new tasks that can be started.

Listing 4-2 shows an example that configures a dispatch source to read data from a file. In this example, the event handler reads the entire contents of the specified file into a buffer and calls a custom function (that you would define in your own code) to process the data. (The caller of this function would use the returned dispatch source to cancel it once the read operation was completed.) To ensure that the dispatch queue does not block unnecessarily when there is no data to read, this example uses the `fcntl` function to configure the file descriptor to perform nonblocking operations. The cancellation handler installed on the dispatch source ensures that the file descriptor is closed after the data is read.

Listing 4-2 Reading data from a file

```
dispatch_source_t ProcessContentsOfFile(const char* filename)
{
   // Prepare the file for reading.
   int fd = open(filename, O_RDONLY);
   if (fd == -1)
      return NULL;
   fcntl(fd, F_SETFL, O_NONBLOCK);  // Avoid blocking the read operation

   dispatch_queue_t queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
   dispatch_source_t readSource = dispatch_source_create(DISPATCH_SOURCE_TYPE_READ,
                                  fd, 0, queue);
   if (!readSource)
   {
      close(fd);
      return NULL;
   }

   // Install the event handler
   dispatch_source_set_event_handler(readSource, ^{
      size_t estimated = dispatch_source_get_data(readSource) + 1;
      // Read the data into a text buffer.
      char* buffer = (char*)malloc(estimated);
      if (buffer)
      {
         ssize_t actual = read(fd, buffer, (estimated));
         Boolean done = MyProcessFileData(buffer, actual);  // Process the data.

         // Release the buffer when done.
         free(buffer);

         // If there is no more data, cancel the source.
         if (done)
            dispatch_source_cancel(readSource);
      }
    });

   // Install the cancellation handler
   dispatch_source_set_cancel_handler(readSource, ^{close(fd);});

   // Start reading the file.
   dispatch_resume(readSource);
   return readSource;
}
```

In the preceding example, the custom `MyProcessFileData` function determines when enough file data has been read and the dispatch source can be canceled. By default, a dispatch source configured for reading from a descriptor schedules its event handler repeatedly while there is still data to read. If the socket connection closes or you reach the end of a file, the dispatch source automatically stops scheduling the event handler. If you know you do not need a dispatch source though, you can cancel it directly yourself.

Writing Data to a Descriptor

The process for writing data to a file or socket is very similar to the process for reading data. After configuring a descriptor for write operations, you create a dispatch source of type `DISPATCH_SOURCE_TYPE_WRITE`. Once that dispatch source is created, the system calls your event handler to give it a chance to start writing data to the file or socket. When you are finished writing data, use the `dispatch_source_cancel` function to cancel the dispatch source.

Whenever writing data, you should always configure your file descriptor to use non-blocking operations. Although you can use the `dispatch_source_get_data` function to see how much space is available for writing, the value returned by that function is advisory only and could change between the time you make the call and the time you actually write the data. If an error occurs, writing data to a blocking file descriptor could stall your event handler in mid execution and prevent the dispatch queue from dispatching other tasks. For a serial queue, this could deadlock your queue, and even for a concurrent queue this reduces the number of new tasks that can be started.

Listing 4-3 shows the basic approach for writing data to a file using a dispatch source. After creating the new file, this function passes the resulting file descriptor to its event handler. The data being put into the file is provided by the `MyGetData` function, which you would replace with whatever code you needed to generate the data for the file. After writing the data to the file, the event handler cancels the dispatch source to prevent it from being called again. The owner of the dispatch source would then be responsible for releasing it.

Listing 4-3 Writing data to a file

```
dispatch_source_t WriteDataToFile(const char* filename)
{
    int fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC,
                      (S_IRUSR | S_IWUSR | S_ISUID | S_ISGID));
    if (fd == -1)
        return NULL;
    fcntl(fd, F_SETFL); // Block during the write.

    dispatch_queue_t queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
    dispatch_source_t writeSource =
dispatch_source_create(DISPATCH_SOURCE_TYPE_WRITE,
                            fd, 0, queue);
    if (!writeSource)
    {
        close(fd);
        return NULL;
    }

    dispatch_source_set_event_handler(writeSource, ^{
        size_t bufferSize = MyGetDataSize();
        void* buffer = malloc(bufferSize);

        size_t actual = MyGetData(buffer, bufferSize);
        write(fd, buffer, actual);

        free(buffer);

        // Cancel and release the dispatch source when done.
        dispatch_source_cancel(writeSource);
    });

    dispatch_source_set_cancel_handler(writeSource, ^{close(fd);});
    dispatch_resume(writeSource);
    return (writeSource);
}
```

Monitoring a File-System Object

If you want to monitor a file system object for changes, you can set up a dispatch source of type `DISPATCH_SOURCE_TYPE_VNODE`. You can use this type of dispatch source to receive notifications when a file is deleted, written to, or renamed. You can also use it to be alerted when specific types of meta information for a file (such as its size and link count) change.

Note: The file descriptor you specify for your dispatch source must remain open while the source itself is processing events.

Listing 4-4 shows an example that monitors a file for name changes and performs some custom behavior when it does. (You would provide the actual behavior in place of the `MyUpdateFileName` function called in the example.) Because a descriptor is opened specifically for the dispatch source, the dispatch source includes a cancellation handler that closes the descriptor. Because the file descriptor created by the example is associated with the underlying file-system object, this same dispatch source can be used to detect any number of filename changes.

Listing 4-4 Watching for filename changes

```
dispatch_source_t MonitorNameChangesToFile(const char* filename)
{
   int fd = open(filename, O_EVTONLY);
   if (fd == -1)
      return NULL;

   dispatch_queue_t queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
   dispatch_source_t source = dispatch_source_create(DISPATCH_SOURCE_TYPE_VNODE,
                fd, DISPATCH_VNODE_RENAME, queue);
   if (source)
   {
      // Copy the filename for later use.
      int length = strlen(filename);
      char* newString = (char*)malloc(length + 1);
      newString = strcpy(newString, filename);
      dispatch_set_context(source, newString);

      // Install the event handler to process the name change
      dispatch_source_set_event_handler(source, ^{
            const char*  oldFilename = (char*)dispatch_get_context(source);
            MyUpdateFileName(oldFilename, fd);
      });

      // Install a cancellation handler to free the descriptor
      // and the stored string.
      dispatch_source_set_cancel_handler(source, ^{
          char* fileStr = (char*)dispatch_get_context(source);
          free(fileStr);
          close(fd);
      });

      // Start processing events.
      dispatch_resume(source);
   }
   else
      close(fd);

   return source;
}
```

Monitoring Signals

UNIX signals allow the manipulation of an application from outside of its domain. An application can receive many different types of signals ranging from unrecoverable errors (such as illegal instructions) to notifications about important information (such as when a child process exits). Traditionally, applications use the `sigaction` function to install a signal handler function, which processes signals synchronously as soon as they arrive. If you just want to be notified of a signal's arrival and do not actually want to handle the signal, you can use a signal dispatch source to process the signals asynchronously.

Signal dispatch sources are not a replacement for the synchronous signal handlers you install using the `sigaction` function. Synchronous signal handlers can actually catch a signal and prevent it from terminating your application. Signal dispatch sources allow you to monitor only the arrival of the signal. In addition, you cannot use signal dispatch sources to retrieve all types of signals. Specifically, you cannot use them to monitor the `SIGILL`, `SIGBUS`, and `SIGSEGV` signals.

Because signal dispatch sources are executed asynchronously on a dispatch queue, they do not suffer from some of the same limitations as synchronous signal handlers. For example, there are no restrictions on the functions you can call from your signal dispatch source's event handler. The tradeoff for this increased flexibility is the fact that there may be some increased latency between the time a signal arrives and the time your dispatch source's event handler is called.

Listing 4-5 shows how you configure a signal dispatch source to handle the `SIGHUP` signal. The event handler for the dispatch source calls the `MyProcessSIGHUP` function, which you would replace in your application with code to process the signal.

Listing 4-5 Installing a block to monitor signals

```
void InstallSignalHandler()
{
   // Make sure the signal does not terminate the application.
   signal(SIGHUP, SIG_IGN);


   dispatch_queue_t queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
   dispatch_source_t source = dispatch_source_create(DISPATCH_SOURCE_TYPE_SIGNAL,
SIGHUP, 0, queue);


   if (source)
   {
      dispatch_source_set_event_handler(source, ^{
         MyProcessSIGHUP();
      });


      // Start processing signals
      dispatch_resume(source);
   }
}
```

If you are developing code for a custom framework, an advantage of using signal dispatch sources is that your code can monitor signals independent of any applications linked to it. Signal dispatch sources do not interfere with other dispatch sources or any synchronous signal handlers the application might have installed.

For more information about implementing synchronous signal handlers, and for a list of signal names, see `signal` man page.

Monitoring a Process

A process dispatch source lets you monitor the behavior of a specific process and respond appropriately. A parent process might use this type of dispatch source to monitor any child processes it creates. For example, the parent process could use it to watch for the death of a child process. Similarly, a child process could use it to monitor its parent process and exit if the parent process exits.

Listing 4-6 shows the steps for installing a dispatch source to monitor for the termination of a parent process. When the parent process dies, the dispatch source sets some internal state information to let the child process know it should exit. (Your own application would need to implement the `MySetAppExitFlag` function to set an appropriate flag for termination.) Because the dispatch source runs autonomously, and therefore owns itself, it also cancels and releases itself in anticipation of the program shutting down.

Listing 4-6 Monitoring the death of a parent process

```
void MonitorParentProcess()
{
   pid_t parentPID = getppid();


   dispatch_queue_t queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
   dispatch_source_t source = dispatch_source_create(DISPATCH_SOURCE_TYPE_PROC,
                                                     parentPID, DISPATCH_PROC_EXIT,
queue);
   if (source)
   {
      dispatch_source_set_event_handler(source, ^{
         MySetAppExitFlag();
         dispatch_source_cancel(source);
         dispatch_release(source);
      });
      dispatch_resume(source);
   }
}
```

Canceling a Dispatch Source

Dispatch sources remain active until you cancel them explicitly using the `dispatch_source_cancel` function. Canceling a dispatch source stops the delivery of new events and cannot be undone. Therefore, you typically cancel a dispatch source and then immediately release it, as shown here:

```
void RemoveDispatchSource(dispatch_source_t mySource)
{
   dispatch_source_cancel(mySource);
   dispatch_release(mySource);
}
```

Cancellation of a dispatch source is an asynchronous operation. Although no new events are processed after you call the `dispatch_source_cancel` function, events that are already being processed by the dispatch source continue to be processed. After it finishes processing any final events, the dispatch source executes its cancellation handler if one is present.

The cancellation handler is your chance to deallocate memory or clean up any resources that were acquired on behalf of the dispatch source. If your dispatch source uses a descriptor or mach port, you must provide a cancellation handler to close the descriptor or destroy the port when cancellation occurs. Other types of dispatch sources do not require cancellation handlers, although you still should provide one if you associate any memory or data with the dispatch source. For example, you should provide one if you store data in the dispatch source's context pointer. For more information about cancellation handlers, see Installing a Cancellation Handler.

Suspending and Resuming Dispatch Sources

You can suspend and resume the delivery of dispatch source events temporarily using the `dispatch_suspend` and `dispatch_resume` methods. These methods increment and decrement the suspend count for your dispatch object. As a result, you must balance each call to `dispatch_suspend` with a matching call to `dispatch_resume` before event delivery resumes.

When you suspend a dispatch source, any events that occur while that dispatch source is suspended are accumulated until the queue is resumed. When the queue resumes, rather than deliver all of the events, the events are coalesced into a single event before delivery. For example, if you were monitoring a file for name changes, the delivered event would include only the last name change. Coalescing events in this way prevents them from building up in the queue and overwhelming your application when work resumes.