Run Loops

(Legacy)

Core Foundation > Events & Other Input

2008-10-15

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

Simultaneously published in the United States and Canada.

Contents

Listings

Introduction to Run Loops

Important: The information in this document is superseded by the information in *Threading Programming Guide*. For information about how to configure a run loop for your custom threads, see that document instead.

The run loop is the center of any event-driven application or tool. After it is fully initialized, an application normally enters its main run loop, waiting for events. Everything the application does from that point on is the result of a callout from the run loop in response to events being detected and dispatched. This document describes how run loops work and how you can use it in your applications.

Organization of This Document

This topic is under construction.

The following concepts are covered in this topic.

- "About Run Loops" (page 9)
- "Input Modes" (page 11)

The following tasks are covered in this topic.

- "Running the Run Loop" (page 13)
- "Using Run Loops With Cocoa and Carbon" (page 15)

About Run Loops

Core Foundation provides the basis for every application's event loop with the CFRunLoop opaque type. A CFRunLoop object monitors objects that represent various sources of input to a task. The run loop dispatches control when an input source becomes ready for processing. Examples of input sources might include user input devices, network connections, periodic or time-delayed events, and asynchronous callbacks. Input sources are registered with a run loop, and when a run loop is "run", callback functions associated with each input source are called when some activity occurs.

While being run, a run loop goes through a cycle of activities. Input sources are checked, timers which need firing are fired, and then the run loop blocks, waiting for something to happen (or in the case of timers, waiting for it to be time for something to happen). When something does happen, the run loop wakes up, processes the activity (usually by calling a callback function for an input source), checks other sources, fires timers, and goes back to sleep. And so on.

Run loops are strongly tied to the threads in your application. Every thread has exactly one run loop. No more and no less. Each thread's run loop monitors its own independent list of objects. (See "Input Modes" (page 11) for details on how a run loop can monitor subsets of its objects.) In a Carbon or Cocoa application, for instance, the main thread's run loop normally monitors all the events generated by the user. Additional threads may use their run loops to listen for (and then process) network activity, to receive messages from other threads or processes, or to perform periodic activities. By placing these input sources in different run loops on separate threads, the events can be processed without blocking any other thread's run loop, such as the main thread's run loop, which processes user events.

Three types of objects can be placed into and monitored by a run loop: sources, timers, and observers. Each is described in the following sections.

Sources

Run loop sources, represented by the CFRunLoopSource opaque type, are abstractions of input sources that can be put into a run loop. Input sources typically generate asynchronous events, such as messages arriving on a network port or actions performed by the user.

An input source type normally defines an API for creating and operating on objects of the type, as if it were a separate entity from the run loop, then provides a function to create a CFRunLoopSource for an object. The run loop source can then be registered with the run loop and act as an intermediary between the run loop and the actual input source type object. Examples of input sources include CFMachPort, CFMessagePort, and CFSocket.

Timers

Run loop timers, represented by the CFRunLoopTimer opaque type, are specialized run loop sources that fire at preset times in the future. Timers can fire either only once or repeatedly at fixed time intervals. Repeating timers can also have their next firing time manually adjusted.

Observers

Run loop observers, represented by the CFRunLoopObserver opaque type, provide a general means to receive callbacks at different points within a running run loop. In contrast to sources, which fire when an asynchronous event occurs, and timers, which fire when a particular time passes, observers fire at special locations within the execution of the run loop, such as before sources are processed or before the run loop goes to sleep, waiting for an event to occur. In essence, observers are specialized run loop sources that represent events inside the run loop itself.

Observers can be either one-time events or repeated every time through the run loop's loop.

Input Modes

Each run loop can have different modes, each identified with an arbitrary string name, in which it can run. Each mode has its own set of sources, timers, and observers associated with it. A run loop is run—in a named mode—to have it monitor the objects that have been registered in that mode. Examples of modes include the default mode, which a process would normally spend most of its time in, and a modal panel mode, which might be run when a modal panel is up, to restrict the set of input sources that are allowed to "fire." Modes do not provide the granularity of, for example, what type of user input events are interesting, however. That sort of finer-grained granularity is given by higher-level frameworks, such as Cocoa and Carbon, with "get next event matching mask" or similar functionality.

To receive callbacks when a run loop source, timer, or observer needs processing, you must first place the object into a run loop mode, using the appropriate `CFRunLoopAdd...` function. You can later remove an object from a run loop mode, using the appropriate `CFRunLoopRemove...` function or by invalidating the object, to stop receiving its callback.

Core Foundation defines a default mode to hold objects that should be monitored while the application (or thread) is sitting idle. For example, Carbon and Cocoa applications place run loop sources for user events into this mode of their main thread's run loop. You access this default mode using the `kCFRunLoopDefaultMode` constant for the mode name. Additional modes are created automatically when an object is added to an unrecognized mode.

Each run loop has its own independent set of modes. Multiple run loops can have modes with the same name, but they do not share the objects placed into them. In fact, every run loop has a default mode named with the `kCFRunLoopDefaultMode` constant, but each can, and usually will, have different sets of objects in them.

Core Foundation also defines a special pseudo-mode to hold objects that should be shared by a set of "common" modes. Common modes are a set of run loop modes for which you can define a set of sources, timers, and observers that are shared by these modes. Instead of registering a source, for example, to each specific run loop mode, you can register it once to the run loop's common pseudo-mode and the source will be automatically registered in each run loop mode in the common mode set. The default mode, `kCFRunLoopDefaultMode`, is always a member of the set of common modes. Additional modes are added to the set of common modes by calling the `CFRunLoopAddCommonMode` function.

Objects are added and removed from a run loop's common pseudo-mode the same as a regular mode; you merely use the `kCFRunLoopCommonModes` constant for the mode name. To monitor these objects, however, you do not then run the run loop in the common pseudo-mode. You instead run the run loop in one of modes that is a member of the set of common modes. The `kCFRunLoopCommonModes` constant is never passed to the `CFRunLoopRunInMode` function.

Each run loop has its own independent list of modes that are in the set of common modes. Adding a mode to one run loop's set of common modes does not add it to every run loop's set, even when other run loops have a mode with the same name.

Running the Run Loop

There is exactly one run loop per thread. You neither create nor destroy a thread's run loop. Core Foundation automatically creates it for you as needed. You obtain the current thread's run loop with the `CFRunLoopGetCurrent` function. Call `CFRunLoopRun` to run the current thread's run loop in the default mode until the run loop is stopped with `CFRunLoopStop`. You can also call `CFRunLoopRunInMode` to run the current thread's run loop in a specified mode for a set period of time, until the run loop is stopped, or until after the next run loop source is processed.

Listing 1 (page 13) contains a function, `RunMyTimer`, that uses the run loop with a timer to call a particular function every 5 seconds for 20 seconds. The function creates the run loop timer, `myTimer`, with the `CFRunLoopTimerCreate` function, initializing the timer's callback function to `MyTimerFunction`, its initial firing time to 1 second in the future, and its periodicity to 5 seconds. Before the timer can fire, the function has to place the timer into a run loop mode, in this case a new mode named `"MyCustomMode,"` and then run the run loop in that mode by calling `CFRunLoopRunInMode`. The run loop will run for 20 seconds before control returns to the caller. During this time, the timer will fire and `MyTimerFunction` will be called 4 times. After the run loop exits, the function invalidates the timer to render it inoperable and to remove it from all run loop modes. Finally, because the function still holds a reference to the timer from the create function, the function releases the timer object.

Listing 1 Running a run loop with a timer

```
void MyTimerFunction( CFRunLoopTimerRef timer, void *info );

void RunMyTimer ()
{
    CFStringRef myCustomMode = CFSTR("MyCustomMode");
    CFRunLoopTimerRef myTimer;

    myTimer = CFRunLoopTimerCreate( NULL, CFAbsoluteTimeGetCurrent()+1.0,
                    5.0, 0, 0, MyTimerFunction, NULL );
    CFRunLoopAddTimer( CFRunLoopGetCurrent(), myTimer, myCustomMode );
    CFRunLoopRunInMode( myCustomMode, 20.0, false );
    CFRunLoopTimerInvalidate( myTimer );
    CFRelease( myTimer );
}
```

A run loop can only run if the requested mode has at least one source or timer to monitor. If a run loop mode is empty, the `CFRunLoopRun` and `CFRunLoopRunInMode` functions return immediately without doing anything.

Run loops can be run recursively. You can call `CFRunLoopRun` or `CFRunLoopRunInMode` from within any run loop callout and create nested run loop activations on the current thread's call stack. You are not restricted in which modes you can run from within a callout. You can create another run loop activation running in any available run loop mode, including any modes already running higher in the call stack.

Using Run Loops With Cocoa and Carbon

Cocoa and Carbon each build upon CFRunLoop to implement their own higher-level event loop. Cocoa exposes its event loop through the NSApplication and NSRunLoop classes; Carbon exposes its event loop through the Carbon Event Manager.

When writing a Cocoa or Carbon application, you can add your sources, timers, and observers to their run loops and modes. Your objects will then get monitored as part of the regular application event loop. Use the NSRunLoop instance method `getCFRunLoop` to obtain the CFRunLoop corresponding to a Cocoa run loop. In Carbon applications, use the `GetCFRunLoopFromEventLoop` function.

Cocoa and Carbon automatically set up and run the run loop in the main thread of the application. If you spawn additional threads, you are responsible for managing and running their run loops.

Cocoa defines several of its own run loop modes for use with NSRunLoop objects. Cocoa defines `NSDefaultRunLoopMode` as its default run loop mode, but this is equivalent to Core Foundation's `kCFRunLoopDefaultMode` and can be used interchangeably. Cocoa also defines the modes `NSModalPanelRunLoopMode` and `NSEventTrackingRunLoopMode` in which it runs the main thread's run loop when a modal panel is up or during event-tracking operations, such as drag-and-drop operations, respectively. Both of these modes are members of the main thread's set of common modes.

Document Revision History

This table describes the changes to *Run Loops*.

Date	Notes
2008-10-15	This document is now replaced by the Threading Programming Guide.
2003-01-17	Converted existing Core Foundation documentation into topic format. Added revision history.