Run Loops

Run loops are part of the fundamental infrastructure associated with threads. A *run loop* is an event processing loop that you use to schedule work and coordinate the receipt of incoming events. The purpose of a run loop is to keep your thread busy when there is work to do and put your thread to sleep when there is none.

Run loop management is not entirely automatic. You must still design your thread's code to start the run loop at appropriate times and respond to incoming events. Both Cocoa and Core Foundation provide *run loop objects* to help you configure and manage your thread's run loop. Your application does not need to create these objects explicitly; each thread, including the application's main thread, has an associated run loop object. Only secondary threads need to run their run loop explicitly, however. The app frameworks automatically set up and run the run loop on the main thread as part of the application startup process.

The following sections provide more information about run loops and how you configure them for your application. For additional information about run loop objects, see *NSRunLoop Class Reference* and *CFRunLoop Reference*.

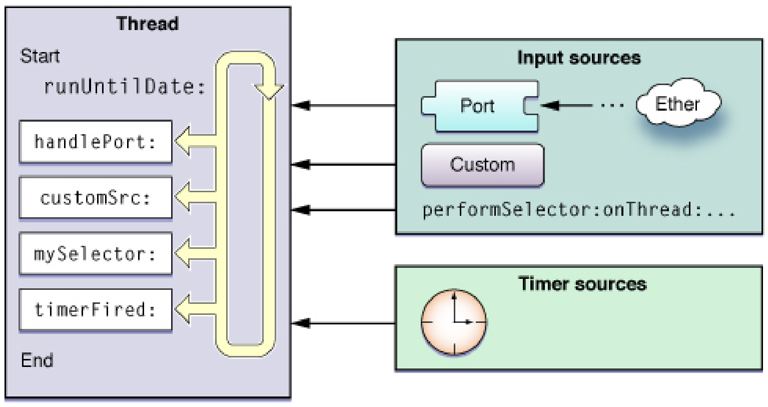
Anatomy of a Run Loop

A run loop is very much like its name sounds. It is a loop your thread enters and uses to run event handlers in response to incoming events. Your code provides the control statements used to implement the actual loop portion of the run loop—in other words, your code provides the `while` or `for` loop that drives the run loop. Within your loop, you use a run loop object to "run" the event-processing code that receives events and calls the installed handlers.

A run loop receives events from two different types of sources. *Input sources* deliver asynchronous events, usually messages from another thread or from a different application. *Timer sources* deliver synchronous events, occurring at a scheduled time or repeating interval. Both types of source use an application-specific handler routine to process the event when it arrives.

Figure 3-1 shows the conceptual structure of a run loop and a variety of sources. The input sources deliver asynchronous events to the corresponding handlers and cause the `runUntilDate:` method (called on the thread's associated `NSRunLoop` object) to exit. Timer sources deliver events to their handler routines but do not cause the run loop to exit.

Figure 3-1 Structure of a run loop and its sources



In addition to handling sources of input, run loops also generate notifications about the run loop's behavior. Registered *run-loop observers* can receive these notifications and use them to do additional processing on the thread. You use Core Foundation to install run-loop observers on your threads.

The following sections provide more information about the components of a run loop and the modes in which they operate. They also describe the notifications that are generated at different times during the handling of events.

Run Loop Modes

A *run loop mode* is a collection of input sources and timers to be monitored and a collection of run loop observers to be notified. Each time you run your run loop, you specify (either explicitly or implicitly) a particular "mode" in which to run. During that pass of the run loop, only sources associated with that mode are monitored and allowed to deliver their events. (Similarly, only observers associated with that mode are notified of the run loop's progress.) Sources associated with other modes hold on to any new events until subsequent passes through the loop in the appropriate mode.

In your code, you identify modes by name. Both Cocoa and Core Foundation define a default mode and several commonly used modes, along with strings for specifying those modes in your code. You can define custom modes by simply specifying a custom string for the mode name. Although the names you assign to custom modes are arbitrary, the contents of those modes are not. You must be sure to add one or more input sources, timers, or run-loop observers to any modes you create for them to be useful.

You use modes to filter out events from unwanted sources during a particular pass through your run loop. Most of the time, you will want to run your run loop in the system-defined "default" mode. A modal panel, however, might run in the "modal" mode. While in this mode, only sources relevant to the modal panel would deliver events to the thread. For secondary threads, you might use custom modes to prevent low-priority sources from delivering events during time-critical operations.

Note: Modes discriminate based on the source of the event, not the type of the event. For example, you would not use modes to match only mouse-down events or only keyboard events. You could use modes to listen to a different set of ports, suspend timers temporarily, or otherwise change the sources and run loop observers currently being monitored.

Table 3-1 lists the standard modes defined by Cocoa and Core Foundation along with a description of when you use that mode. The name column lists the actual constants you use to specify the mode in your code.

Table 3-1 Predefined run loop modes

Mode	Name	Description
Default	`NSDefaultRunLoopMode` (Cocoa) `kCFRunLoopDefaultMode` (Core Foundation)	The default mode is the one used for most operations. Most of the time, you should use this mode to start your run loop and configure your input sources.
Connection	`NSConnectionReplyMode` (Cocoa)	Cocoa uses this mode in conjunction with `NSConnection` objects to monitor replies. You should rarely need to use this mode yourself.
Modal	`NSModalPanelRunLoopMode` (Cocoa)	Cocoa uses this mode to identify events intended for modal panels.
Event tracking	`NSEventTrackingRunLoopMode` (Cocoa)	Cocoa uses this mode to restrict incoming events during mouse-dragging loops and other sorts of user interface tracking loops.
Common modes	`NSRunLoopCommonModes` (Cocoa) `kCFRunLoopCommonModes` (Core Foundation)	This is a configurable group of commonly used modes. Associating an input source with this mode also associates it with each of the modes in the group. For Cocoa applications, this set includes the default, modal, and event tracking modes by default. Core Foundation includes just the default mode initially. You can add custom modes to the set using the `CFRunLoopAddCommonMode` function.

Input Sources

Input sources deliver events asynchronously to your threads. The source of the event depends on the type of the input source, which is generally one of two categories. Port-based input sources monitor your application's Mach ports. Custom input sources monitor custom sources of events. As far as your run loop is concerned, it should not matter whether an input source is port-based or custom. The system typically implements input sources of both types that you can use as is. The only difference between the two sources is how they are signaled. Port-based sources are signaled automatically by the kernel, and custom sources must be signaled manually from another thread.

When you create an input source, you assign it to one or more modes of your run loop. Modes affect which input sources are monitored at any given moment. Most of the time, you run the run loop in the default mode, but you can specify custom modes too. If an input source is not in the currently monitored mode, any events it generates are held until the run loop runs in the correct mode.

The following sections describe some of the input sources.

Port-Based Sources

Cocoa and Core Foundation provide built-in support for creating port-based input sources using port-related objects and functions. For example, in Cocoa, you never have to create an input source directly at all. You simply create a port object and use the methods of `NSPort` to add that port to the run loop. The port object handles the creation and configuration of the needed input source for you.

In Core Foundation, you must manually create both the port and its run loop source. In both cases, you use the functions associated with the port opaque type (`CFMachPortRef`, `CFMessagePortRef`, or `CFSocketRef`) to create the appropriate objects.

For examples of how to set up and configure custom port-based sources, see Configuring a Port-Based Input Source.

Custom Input Sources

To create a custom input source, you must use the functions associated with the `CFRunLoopSourceRef` opaque type in Core Foundation. You configure a custom input source using several callback functions. Core Foundation calls these functions at different points to configure the source, handle any incoming events, and tear down the source when it is removed from the run loop.

In addition to defining the behavior of the custom source when an event arrives, you must also define the event delivery mechanism. This part of the source runs on a separate thread and is responsible for providing the input source with its data and for signaling it when that data is ready for processing. The event delivery mechanism is up to you but need not be overly complex.

For an example of how to create a custom input source, see Defining a Custom Input Source. For reference information for custom input sources, see also *CFRunLoopSource Reference*.

Cocoa Perform Selector Sources

In addition to port-based sources, Cocoa defines a custom input source that allows you to perform a selector on any thread. Like a port-based source, perform selector requests are serialized on the target thread, alleviating many of the synchronization problems that might occur with multiple methods being run on one thread. Unlike a port-based source, a perform selector source removes itself from the run loop after it performs its selector.

Note: Prior to OS X v10.5, perform selector sources were used mostly to send messages to the main thread, but in OS X v10.5 and later and in iOS, you can use them to send messages to any thread.

When performing a selector on another thread, the target thread must have an active run loop. For threads you create, this means waiting until your code explicitly starts the run loop. Because the main thread starts its own run loop, however, you can begin issuing calls on that thread as soon as the application calls the `applicationDidFinishLaunching:` method of the application delegate. The run loop processes all queued perform selector calls each time through the loop, rather than processing one during each loop iteration.

Table 3-2 lists the methods defined on `NSObject` that can be used to perform selectors on other threads. Because these methods are declared on `NSObject`, you can use them from any threads where you have access to Objective-C objects, including POSIX threads. These methods do not actually create a new thread to perform the selector.

Table 3-2 Performing selectors on other threads

Methods	Description

`performSelectorOnMainThread:withObject:waitUntilDone:`
`performSelectorOnMainThread:withObject:waitUntilDone:modes:`
	Performs the specified selector on the application's main thread during that thread's next run loop cycle. These methods give you the option of blocking the current thread until the selector is performed.

`performSelector:onThread:withObject:waitUntilDone:`
`performSelector:onThread:withObject:waitUntilDone:modes:`
	Performs the specified selector on any thread for which you have an `NSThread` object. These methods give you the option of blocking the current thread until the selector is performed.

`performSelector:withObject:afterDelay:`
`performSelector:withObject:afterDelay:inModes:`
	Performs the specified selector on the current thread during the next run loop cycle and after an optional delay period. Because it waits until the next run loop cycle to perform the selector, these methods provide an automatic mini delay from the currently executing code. Multiple queued selectors are performed one after another in the order they were queued.

`cancelPreviousPerformRequestsWithTarget:`
`cancelPreviousPerformRequestsWithTarget:selector:object:`
	Lets you cancel a message sent to the current thread using the `performSelector:withObject:afterDelay:` or `performSelector:withObject:afterDelay:inModes:` method.

For detailed information about each of these methods, see *NSObject Class Reference*.

Timer Sources

Timer sources deliver events synchronously to your threads at a preset time in the future. Timers are a way for a thread to notify itself to do something. For example, a search field could use a timer to initiate an automatic search once a certain amount of time has passed between successive key strokes from the user. The use of this delay time gives the user a chance to type as much of the desired search string as possible before beginning the search.

Although it generates time-based notifications, a timer is not a real-time mechanism. Like input sources, timers are associated with specific modes of your run loop. If a timer is not in the mode currently being monitored by the run loop, it does not fire until you run the run loop in one of the timer's supported modes. Similarly, if a timer fires when the run loop is in the middle of executing a handler routine, the timer waits until the next time through the run loop to invoke its handler routine. If the run loop is not running at all, the timer never fires.

You can configure timers to generate events only once or repeatedly. A repeating timer reschedules itself automatically based on the scheduled firing time, not the actual firing time. For example, if a timer is scheduled to fire at a particular time and every 5 seconds after that, the scheduled firing time will always fall on the original 5 second time intervals, even if the actual firing time gets delayed. If the firing time is delayed so much that it misses one or more of the scheduled firing times, the timer is fired only once for the missed time period. After firing for the missed period, the timer is rescheduled for the next scheduled firing time.

For more information on configuring timer sources, see Configuring Timer Sources. For reference information, see *NSTimer Class Reference* or *CFRunLoopTimer Reference*.

Run Loop Observers

In contrast to sources, which fire when an appropriate asynchronous or synchronous event occurs, run loop observers fire at special locations during the execution of the run loop itself. You might use run loop observers to prepare your thread to process a given event or to prepare the thread before it goes to sleep. You can associate run loop observers with the following events in your run loop:

- The entrance to the run loop.
- When the run loop is about to process a timer.
- When the run loop is about to process an input source.
- When the run loop is about to go to sleep.
- When the run loop has woken up, but before it has processed the event that woke it up.
- The exit from the run loop.

You can add run loop observers to apps using Core Foundation. To create a run loop observer, you create a new instance of the `CFRunLoopObserverRef` opaque type. This type keeps track of your custom callback function and the activities in which it is interested.

Similar to timers, run-loop observers can be used once or repeatedly. A one-shot observer removes itself from the run loop after it fires, while a repeating observer remains attached. You specify whether an observer runs once or repeatedly when you create it.

For an example of how to create a run-loop observer, see Configuring the Run Loop. For reference information, see *CFRunLoopObserver Reference*.

The Run Loop Sequence of Events

Each time you run it, your thread's run loop processes pending events and generates notifications for any attached observers. The order in which it does this is very specific and is as follows:

1. Notify observers that the run loop has been entered.
2. Notify observers that any ready timers are about to fire.
3. Notify observers that any input sources that are not port based are about to fire.
4. Fire any non-port-based input sources that are ready to fire.
5. If a port-based input source is ready and waiting to fire, process the event immediately. Go to step 9.
6. Notify observers that the thread is about to sleep.
7. Put the thread to sleep until one of the following events occurs:
   - An event arrives for a port-based input source.
   - A timer fires.
   - The timeout value set for the run loop expires.
   - The run loop is explicitly woken up.
8. Notify observers that the thread just woke up.
9. Process the pending event.
   - If a user-defined timer fired, process the timer event and restart the loop. Go to step 2.
   - If an input source fired, deliver the event.
   - If the run loop was explicitly woken up but has not yet timed out, restart the loop. Go to step 2.
10. Notify observers that the run loop has exited.

Because observer notifications for timer and input sources are delivered before those events actually occur, there may be a gap between the time of the notifications and the time of the actual events. If the timing between these events is critical, you can use the sleep and awake-from-sleep notifications to help you correlate the timing between the actual events.

Because timers and other periodic events are delivered when you run the run loop, circumventing that loop disrupts the delivery of those events. The typical example of this behavior occurs whenever you implement a mouse-tracking routine by entering a loop and repeatedly requesting events from the application. Because your code is grabbing events directly, rather than letting the application dispatch those events normally, active timers would be unable to fire until after your mouse-tracking routine exited and returned control to the application.

A run loop can be explicitly woken up using the run loop object. Other events may also cause the run loop to be woken up. For example, adding another non-port-based input source wakes up the run loop so that the input source can be processed immediately, rather than waiting until some other event occurs.

When Would You Use a Run Loop?

The only time you need to run a run loop explicitly is when you create secondary threads for your application. The run loop for your application's main thread is a crucial piece of infrastructure. As a result, the app frameworks provide the code for running the main application loop and start that loop automatically. The `run` method of `UIApplication` in iOS (or `NSApplication` in OS X) starts an application's main loop as part of the normal startup sequence. If you use the Xcode template projects to create your application, you should never have to call these routines explicitly.

For secondary threads, you need to decide whether a run loop is necessary, and if it is, configure and start it yourself. You do not need to start a thread's run loop in all cases. For example, if you use a thread to perform some long-running and predetermined task, you can probably avoid starting the run loop. Run loops are intended for situations where you want more interactivity with the thread. For example, you need to start a run loop if you plan to do any of the following:

- Use ports or custom input sources to communicate with other threads.
- Use timers on the thread.
- Use any of the `performSelector…` methods in a Cocoa application.
- Keep the thread around to perform periodic tasks.

If you do choose to use a run loop, the configuration and setup is straightforward. As with all threaded programming though, you should have a plan for exiting your secondary threads in appropriate situations. It is always better to end a thread cleanly by letting it exit than to force it to terminate. Information on how to configure and exit a run loop is described in Using Run Loop Objects.

Using Run Loop Objects

A run loop object provides the main interface for adding input sources, timers, and run-loop observers to your run loop and then running it. Every thread has a single run loop object associated with it. In Cocoa, this object is an instance of the `NSRunLoop` class. In a low-level application, it is a pointer to a `CFRunLoopRef` opaque type.

Getting a Run Loop Object

To get the run loop for the current thread, you use one of the following:

- In a Cocoa application, use the `currentRunLoop` class method of `NSRunLoop` to retrieve an `NSRunLoop` object.
- Use the `CFRunLoopGetCurrent` function.

Although they are not toll-free bridged types, you can get a `CFRunLoopRef` opaque type from an `NSRunLoop` object when needed. The `NSRunLoop` class defines a `getCFRunLoop` method that returns a `CFRunLoopRef` type that you can pass to Core Foundation routines. Because both objects refer to the same run loop, you can intermix calls to the `NSRunLoop` object and `CFRunLoopRef` opaque type as needed.

Configuring the Run Loop

Before you run a run loop on a secondary thread, you must add at least one input source or timer to it. If a run loop does not have any sources to monitor, it exits immediately when you try to run it. For examples of how to add sources to a run loop, see Configuring Run Loop Sources.

In addition to installing sources, you can also install run loop observers and use them to detect different execution stages of the run loop. To install a run loop observer, you create a `CFRunLoopObserverRef` opaque type and use the `CFRunLoopAddObserver` function to add it to your run loop. Run loop observers must be created using Core Foundation, even for Cocoa applications.

Listing 3-1 shows the main routine for a thread that attaches a run loop observer to its run loop. The purpose of the example is to show you how to create a run loop observer, so the code simply sets up a run loop observer to monitor all run loop activities. The basic handler routine (not shown) simply logs the run loop activity as it processes the timer requests.

Listing 3-1 Creating a run loop observer

```
- (void)threadMain
{
    // The application uses garbage collection, so no autorelease pool is needed.
    NSRunLoop* myRunLoop = [NSRunLoop currentRunLoop];

    // Create a run loop observer and attach it to the run loop.
    CFRunLoopObserverContext  context = {0, self, NULL, NULL, NULL};
    CFRunLoopObserverRef    observer = CFRunLoopObserverCreate(kCFAllocatorDefault,
            kCFRunLoopAllActivities, YES, 0, &myRunLoopObserver, &context);

    if (observer)
    {
        CFRunLoopRef    cfLoop = [myRunLoop getCFRunLoop];
        CFRunLoopAddObserver(cfLoop, observer, kCFRunLoopDefaultMode);
    }

    // Create and schedule the timer.
    [NSTimer scheduledTimerWithTimeInterval:0.1 target:self
                selector:@selector(doFireTimer:) userInfo:nil repeats:YES];

    NSInteger    loopCount = 10;
    do
    {
        // Run the run loop 10 times to let the timer fire.
        [myRunLoop runUntilDate:[NSDate dateWithTimeIntervalSinceNow:1]];
        loopCount--;
    }
    while (loopCount);
}
```

When configuring the run loop for a long-lived thread, it is better to add at least one input source to receive messages. Although you can enter the run loop with only a timer attached, once the timer fires, it is typically invalidated, which would then cause the run loop to exit. Attaching a repeating timer could keep the run loop running over a longer period of time, but would involve firing the timer periodically to wake your thread, which is effectively another form of polling. By contrast, an input source waits for an event to happen, keeping your thread asleep until it does.

Starting the Run Loop

Starting the run loop is necessary only for the secondary threads in your application. A run loop must have at least one input source or timer to monitor. If one is not attached, the run loop exits immediately.

There are several ways to start the run loop, including the following:

- Unconditionally
- With a set time limit
- In a particular mode

Entering your run loop unconditionally is the simplest option, but it is also the least desirable. Running your run loop unconditionally puts the thread into a permanent loop, which gives you very little control over the run loop itself. You can add and remove input sources and timers, but the only way to stop the run loop is to kill it. There is also no way to run the run loop in a custom mode.

Instead of running a run loop unconditionally, it is better to run the run loop with a timeout value. When you use a timeout value, the run loop runs until an event arrives or the allotted time expires. If an event arrives, that event is dispatched to a handler for processing and then the run loop exits. Your code can then restart the run loop to handle the next event. If the allotted time expires instead, you can simply restart the run loop or use the time to do any needed housekeeping.

In addition to a timeout value, you can also run your run loop using a specific mode. Modes and timeout values are not mutually exclusive and can both be used when starting a run loop. Modes limit the types of sources that deliver events to the run loop and are described in more detail in Run Loop Modes.

Listing 3-2 shows a skeleton version of a thread's main entry routine. The key portion of this example shows the basic structure of a run loop. In essence, you add your input sources and timers to the run loop and then repeatedly call one of the routines to start the run loop. Each time the run loop routine returns, you check to see if any conditions have arisen that might warrant exiting the thread. The example uses the Core Foundation run loop routines so that it can check the return result and determine why the run loop exited. You could also use the methods of the `NSRunLoop` class to run the run loop in a similar manner if you are using Cocoa and do not need to check the return value. (For an example of a run loop that calls methods of the `NSRunLoop` class, see Listing 3-14.)

Listing 3-2 Running a run loop

```
- (void)skeletonThreadMain
{
    // Set up an autorelease pool here if not using garbage collection.
    BOOL done = NO;

    // Add your sources or timers to the run loop and do any other setup.


    do
    {
        // Start the run loop but return after each source is handled.
        SInt32    result = CFRunLoopRunInMode(kCFRunLoopDefaultMode, 10, YES);

        // If a source explicitly stopped the run loop, or if there are no
        // sources or timers, go ahead and exit.
        if ((result == kCFRunLoopRunStopped) || (result == kCFRunLoopRunFinished))
            done = YES;

        // Check for any other exit conditions here and set the
        // done variable as needed.
    }
    while (!done);

    // Clean up code here. Be sure to release any allocated autorelease pools.
}
```

It is possible to run a run loop recursively. In other words, you can call `CFRunLoopRun`, `CFRunLoopRunInMode`, or any of the `NSRunLoop` methods for starting the run loop from within the handler routine of an input source or timer. When doing so, you can use any mode you want to run the nested run loop, including the mode in use by the outer run loop.

Exiting the Run Loop

There are two ways to make a run loop exit before it has processed an event:

- Configure the run loop to run with a timeout value.
- Tell the run loop to stop.

Using a timeout value is certainly preferred, if you can manage it. Specifying a timeout value lets the run loop finish all of its normal processing, including delivering notifications to run loop observers, before exiting.

Stopping the run loop explicitly with the `CFRunLoopStop` function produces a result similar to a timeout. The run loop sends out any remaining run-loop notifications and then exits. The difference is that you can use this technique on run loops you started unconditionally.

Although removing a run loop's input sources and timers may also cause the run loop to exit, this is not a reliable way to stop a run loop. Some system routines add input sources to a run loop to handle needed events. Because your code might not be aware of these input sources, it would be unable to remove them, which would prevent the run loop from exiting.

Thread Safety and Run Loop Objects

Thread safety varies depending on which API you are using to manipulate your run loop. The functions in Core Foundation are generally thread-safe and can be called from any thread. If you are performing operations that alter the configuration of the run loop, however, it is still good practice to do so from the thread that owns the run loop whenever possible.

The Cocoa `NSRunLoop` class is not as inherently thread safe as its Core Foundation counterpart. If you are using the `NSRunLoop` class to modify your run loop, you should do so only from the same thread that owns that run loop. Adding an input source or timer to a run loop belonging to a different thread could cause your code to crash or behave in an unexpected way.

Configuring Run Loop Sources

The following sections show examples of how to set up different types of input sources in both Cocoa and Core Foundation.

Defining a Custom Input Source

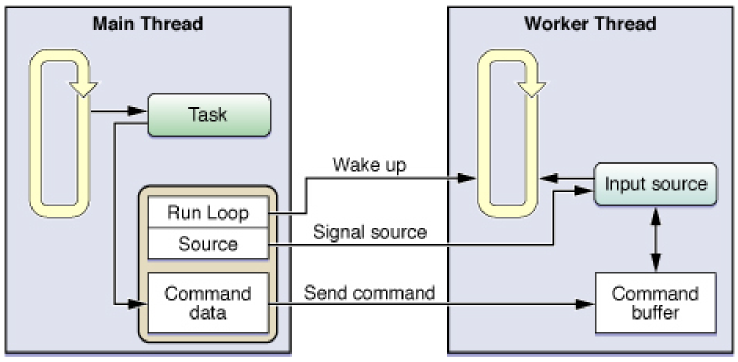
Creating a custom input source involves defining the following:

- The information you want your input source to process.
- A scheduler routine to let interested clients know how to contact your input source.
- A handler routine to perform requests sent by any clients.
- A cancellation routine to invalidate your input source.

Because you create a custom input source to process custom information, the actual configuration is designed to be flexible. The scheduler, handler, and cancellation routines are the key routines you almost always need for your custom input source. Most of the rest of the input source behavior, however, happens outside of those handler routines. For example, it is up to you to define the mechanism for passing data to your input source and for communicating the presence of your input source to other threads.

Figure 3-2 shows a sample configuration of a custom input source. In this example, the application's main thread maintains references to the input source, the custom command buffer for that input source, and the run loop on which the input source is installed. When the main thread has a task it wants to hand off to the worker thread, it posts a command to the command buffer along with any information needed by the worker thread to start the task. (Because both the main thread and the input source of the worker thread have access to the command buffer, that access must be synchronized.) Once the command is posted, the main thread signals the input source and wakes up the worker thread's run loop. Upon receiving the wake up command, the run loop calls the handler for the input source, which processes the commands found in the command buffer.

Figure 3-2 Operating a custom input source



The following sections explain the implementation of the custom input source from the preceding figure and show the key code you would need to implement.

Defining the Input Source

Defining a custom input source requires the use of Core Foundation routines to configure your run loop source and attach it to a run loop. Although the basic handlers are C-based functions, that does not preclude you from writing wrappers for those functions and using Objective-C or C++ to implement the body of your code.

The input source introduced in Figure 3-2 uses an Objective-C object to manage a command buffer and coordinate with the run loop. Listing 3-3 shows the definition of this object. The `RunLoopSource` object manages a command buffer and uses that buffer to receive messages from other threads. This listing also shows the definition of the `RunLoopContext` object, which is really just a container object used to pass a `RunLoopSource` object and a run loop reference to the application's main thread.

Listing 3-3 The custom input source object definition

```
@interface RunLoopSource : NSObject
{
    CFRunLoopSourceRef runLoopSource;
    NSMutableArray* commands;
}

- (id)init;
- (void)addToCurrentRunLoop;
- (void)invalidate;

// Handler method
- (void)sourceFired;

// Client interface for registering commands to process
- (void)addCommand:(NSInteger)command withData:(id)data;
- (void)fireAllCommandsOnRunLoop:(CFRunLoopRef)runloop;

@end

// These are the CFRunLoopSourceRef callback functions.
void RunLoopSourceScheduleRoutine (void *info, CFRunLoopRef rl, CFStringRef mode);
void RunLoopSourcePerformRoutine (void *info);
void RunLoopSourceCancelRoutine (void *info, CFRunLoopRef rl, CFStringRef mode);

// RunLoopContext is a container object used during registration of the input source.
@interface RunLoopContext : NSObject
{
    CFRunLoopRef        runLoop;
    RunLoopSource*        source;
}
@property (readonly) CFRunLoopRef runLoop;
@property (readonly) RunLoopSource* source;

- (id)initWithSource:(RunLoopSource*)src andLoop:(CFRunLoopRef)loop;
@end
```

Although the Objective-C code manages the custom data of the input source, attaching the input source to a run loop requires C-based callback functions. The first of these functions is called when you actually attach the run loop source to your run loop, and is shown in Listing 3-4. Because this input source has only one client (the main thread), it uses the scheduler function to send a message to register itself with the application delegate on that thread. When the delegate wants to communicate with the input source, it uses the information in `RunLoopContext` object to do so.

Listing 3-4 Scheduling a run loop source

```
void RunLoopSourceScheduleRoutine (void *info, CFRunLoopRef rl, CFStringRef mode)
{
    RunLoopSource* obj = (RunLoopSource*)info;
    AppDelegate*   del = [AppDelegate sharedAppDelegate];
    RunLoopContext* theContext = [[RunLoopContext alloc] initWithSource:obj andLoop:rl];

    [del performSelectorOnMainThread:@selector(registerSource:)
                            withObject:theContext waitUntilDone:NO];
}
```

One of the most important callback routines is the one used to process custom data when your input source is signaled. Listing 3-5 shows the perform callback routine associated with the `RunLoopSource` object. This function simply forwards the request to do the work to the `sourceFired` method, which then processes any commands present in the command buffer.

Listing 3-5 Performing work in the input source

```
void RunLoopSourcePerformRoutine (void *info)
{
    RunLoopSource*  obj = (RunLoopSource*)info;
    [obj sourceFired];
}
```

If you ever remove your input source from its run loop using the `CFRunLoopSourceInvalidate` function, the system calls your input source's cancellation routine. You can use this routine to notify clients that your input source is no longer valid and that they should remove any references to it. Listing 3-6 shows the cancellation callback routine registered with the `RunLoopSource` object. This function sends another `RunLoopContext` object to the application delegate, but this time asks the delegate to remove references to the run loop source.

Listing 3-6 Invalidating an input source

```
void RunLoopSourceCancelRoutine (void *info, CFRunLoopRef rl, CFStringRef mode)
{
    RunLoopSource* obj = (RunLoopSource*)info;
    AppDelegate* del = [AppDelegate sharedAppDelegate];
    RunLoopContext* theContext = [[RunLoopContext alloc] initWithSource:obj andLoop:rl];

    [del performSelectorOnMainThread:@selector(removeSource:)
                            withObject:theContext waitUntilDone:YES];
}
```

Note: The code for the application delegate's `registerSource:` and `removeSource:` methods is shown in Coordinating with Clients of the Input Source.

Installing the Input Source on the Run Loop

Listing 3-7 shows the `init` and `addToCurrentRunLoop` methods of the `RunLoopSource` class. The `init` method creates the `CFRunLoopSourceRef` opaque type that must actually be attached to the run loop. It passes the `RunLoopSource` object itself as the contextual information so that the callback routines have a pointer to the object. Installation of the input source does not occur until the worker thread invokes the `addToCurrentRunLoop` method, at which point the `RunLoopSourceScheduleRoutine` callback function is called. Once the input source is added to the run loop, the thread can run its run loop to wait on it.

Listing 3-7 Installing the run loop source

```
- (id)init
{
    CFRunLoopSourceContext    context = {0, self, NULL, NULL, NULL, NULL, NULL,
                                        &RunLoopSourceScheduleRoutine,
                                        RunLoopSourceCancelRoutine,
                                        RunLoopSourcePerformRoutine};

    runLoopSource = CFRunLoopSourceCreate(NULL, 0, &context);
    commands = [[NSMutableArray alloc] init];

    return self;
}

- (void)addToCurrentRunLoop
{
    CFRunLoopRef runLoop = CFRunLoopGetCurrent();
    CFRunLoopAddSource(runLoop, runLoopSource, kCFRunLoopDefaultMode);
}
```

Coordinating with Clients of the Input Source

For your input source to be useful, you need to manipulate it and signal it from another thread. The whole point of an input source is to put its associated thread to sleep until there is something to do. That fact necessitates having other threads in your application know about the input source and have a way to communicate with it.

One way to notify clients about your input source is to send out registration requests when your input source is first installed on its run loop. You can register your input source with as many clients as you want, or you can simply register it with some central agency that then vends your input source to interested clients. Listing 3-8 shows the registration method defined by the application delegate and invoked when the `RunLoopSource` object's scheduler function is called. This method receives the `RunLoopContext` object provided by the `RunLoopSource` object and adds it to its list of sources. This listing also shows the routine used to unregister the input source when it is removed from its run loop.

Listing 3-8 Registering and removing an input source with the application delegate

```
- (void)registerSource:(RunLoopContext*)sourceInfo;
{
    [sourcesToPing addObject:sourceInfo];
}

- (void)removeSource:(RunLoopContext*)sourceInfo
{
    id    objToRemove = nil;

    for (RunLoopContext* context in sourcesToPing)
    {
        if ([context isEqual:sourceInfo])
        {
            objToRemove = context;
            break;
        }
    }

    if (objToRemove)
        [sourcesToPing removeObject:objToRemove];
}
```

Note: The callback functions that call the methods in the preceding listing are shown in Listing 3-4 and Listing 3-6.

Signaling the Input Source

After it hands off its data to the input source, a client must signal the source and wake up its run loop. Signaling the source lets the run loop know that the source is ready to be processed. And because the thread might be asleep when the signal occurs, you should always wake up the run loop explicitly. Failing to do so might result in a delay in processing the input source.

Listing 3-9 shows the `fireCommandsOnRunLoop` method of the `RunLoopSource` object. Clients invoke this method when they are ready for the source to process the commands they added to the buffer.

Listing 3-9 Waking up the run loop

```
- (void)fireCommandsOnRunLoop:(CFRunLoopRef)runloop
{
    CFRunLoopSourceSignal(runLoopSource);
    CFRunLoopWakeUp(runloop);
}
```

Note: You should never try to handle a `SIGHUP` or other type of process-level signal by messaging a custom input source. The Core Foundation functions for waking up the run loop are not signal safe and should not be used inside your application's signal handler routines. For more information about signal handler routines, see the `sigaction` man page.

Configuring Timer Sources

To create a timer source, all you have to do is create a timer object and schedule it on your run loop. In Cocoa, you use the `NSTimer` class to create new timer objects, and in Core Foundation you use the `CFRunLoopTimerRef` opaque type. Internally, the `NSTimer` class is simply an extension of Core Foundation that provides some convenience features, like the ability to create and schedule a timer using the same method.

In Cocoa, you can create and schedule a timer all at once using either of these class methods:

- `scheduledTimerWithTimeInterval:target:selector:userInfo:repeats:`
- `scheduledTimerWithTimeInterval:invocation:repeats:`

These methods create the timer and add it to the current thread's run loop in the default mode (`NSDefaultRunLoopMode`). You can also schedule a timer manually if you want by creating your `NSTimer` object and then adding it to the run loop using the `addTimer:forMode:` method of `NSRunLoop`. Both techniques do basically the same thing but give you different levels of control over the timer's configuration. For example, if you create the timer and add it to the run loop manually, you can do so using a mode other than the default mode. Listing 3-10 shows how to create timers using both techniques. The first timer has an initial delay of 1 second but then fires regularly every 0.1 seconds after that. The second timer begins firing after an initial 0.2 second delay and then fires every 0.2 seconds after that.

Listing 3-10 Creating and scheduling timers using NSTimer

```
NSRunLoop* myRunLoop = [NSRunLoop currentRunLoop];

// Create and schedule the first timer.
NSDate* futureDate = [NSDate dateWithTimeIntervalSinceNow:1.0];
NSTimer* myTimer = [[NSTimer alloc] initWithFireDate:futureDate
                        interval:0.1
                        target:self
                        selector:@selector(myDoFireTimer1:)
                        userInfo:nil
                        repeats:YES];
[myRunLoop addTimer:myTimer forMode:NSDefaultRunLoopMode];

// Create and schedule the second timer.
[NSTimer scheduledTimerWithTimeInterval:0.2
                        target:self
                        selector:@selector(myDoFireTimer2:)
                        userInfo:nil
                        repeats:YES];
```

Listing 3-11 shows the code needed to configure a timer using Core Foundation functions. Although this example does not pass any user-defined information in the context structure, you could use this structure to pass around any custom data you needed for your timer. For more information about the contents of this structure, see its description in *CFRunLoopTimer Reference*.

Listing 3-11 Creating and scheduling a timer using Core Foundation

```
CFRunLoopRef runLoop = CFRunLoopGetCurrent();
CFRunLoopTimerContext context = {0, NULL, NULL, NULL, NULL};
CFRunLoopTimerRef timer = CFRunLoopTimerCreate(kCFAllocatorDefault, 0.1, 0.3, 0, 0,
                                        &myCFTimerCallback, &context);

CFRunLoopAddTimer(runLoop, timer, kCFRunLoopCommonModes);
```

Configuring a Port-Based Input Source

Both Cocoa and Core Foundation provide port-based objects for communicating between threads or between processes. The following sections show you how to set up port communication using several different types of ports.

Configuring an NSMachPort Object

To establish a local connection with an `NSMachPort` object, you create the port object and add it to your primary thread's run loop. When launching your secondary thread, you pass the same object to your thread's entry-point function. The secondary thread can use the same object to send messages back to your primary thread.

Implementing the Main Thread Code

Listing 3-12 shows the primary thread code for launching a secondary worker thread. Because the Cocoa framework performs many of the intervening steps for configuring the port and run loop, the `launchThread` method is noticeably shorter than its Core Foundation equivalent (Listing 3-17); however, the behavior of the two is nearly identical. One difference is that instead of sending the name of the local port to the worker thread, this method sends the `NSPort` object directly.

Listing 3-12 Main thread launch method

```
- (void)launchThread
{
    NSPort* myPort = [NSMachPort port];
    if (myPort)
    {
        // This class handles incoming port messages.
        [myPort setDelegate:self];

        // Install the port as an input source on the current run loop.
        [[NSRunLoop currentRunLoop] addPort:myPort forMode:NSDefaultRunLoopMode];

        // Detach the thread. Let the worker release the port.
        [NSThread detachNewThreadSelector:@selector(LaunchThreadWithPort:)
               toTarget:[MyWorkerClass class] withObject:myPort];
    }
}
```

In order to set up a two-way communications channel between your threads, you might want to have the worker thread send its own local port to your main thread in a check-in message. Receiving the check-in message lets your main thread know that all went well in launching the second thread and also gives you a way to send further messages to that thread.

Listing 3-13 shows the `handlePortMessage:` method for the primary thread. This method is called when data arrives on the thread's own local port. When a check-in message arrives, the method retrieves the port for the secondary thread directly from the port message and saves it for later use.

Listing 3-13 Handling Mach port messages

```
#define kCheckinMessage 100

// Handle responses from the worker thread.
- (void)handlePortMessage:(NSPortMessage *)portMessage
{
    unsigned int message = [portMessage msgid];
    NSPort* distantPort = nil;

    if (message == kCheckinMessage)
    {
        // Get the worker thread’s communications port.
        distantPort = [portMessage sendPort];

        // Retain and save the worker port for later use.
        [self storeDistantPort:distantPort];
    }
    else
    {
        // Handle other messages.
    }
}
```

Implementing the Secondary Thread Code

For the secondary worker thread, you must configure the thread and use the specified port to communicate information back to the primary thread.

Listing 3-14 shows the code for setting up the worker thread. After creating an autorelease pool for the thread, the method creates a worker object to drive the thread execution. The worker object's `sendCheckinMessage:` method (shown in Listing 3-15) creates a local port for the worker thread and sends a check-in message back to the main thread.

Listing 3-14 Launching the worker thread using Mach ports

```
+(void)LaunchThreadWithPort:(id)inData
{
    NSAutoreleasePool*  pool = [[NSAutoreleasePool alloc] init];

    // Set up the connection between this thread and the main thread.
    NSPort* distantPort = (NSPort*)inData;

    MyWorkerClass*  workerObj = [[self alloc] init];
    [workerObj sendCheckinMessage:distantPort];
    [distantPort release];

    // Let the run loop process things.
    do
    {
        [[NSRunLoop currentRunLoop] runMode:NSDefaultRunLoopMode
                            beforeDate:[NSDate distantFuture]];
    }
    while (![workerObj shouldExit]);

    [workerObj release];
    [pool release];
}
```

When using `NSMachPort`, local and remote threads can use the same port object for one-way communication between the threads. In other words, the local port object created by one thread becomes the remote port object for the other thread.

Listing 3-15 shows the check-in routine of the secondary thread. This method sets up its own local port for future communication and then sends a check-in message back to the main thread. The method uses the port object received in the `LaunchThreadWithPort:` method as the target of the message.

Listing 3-15 Sending the check-in message using Mach ports

```
// Worker thread check-in method
- (void)sendCheckinMessage:(NSPort*)outPort
{
    // Retain and save the remote port for future use.
    [self setRemotePort:outPort];

    // Create and configure the worker thread port.
    NSPort* myPort = [NSMachPort port];
    [myPort setDelegate:self];
    [[NSRunLoop currentRunLoop] addPort:myPort forMode:NSDefaultRunLoopMode];

    // Create the check-in message.
    NSPortMessage* messageObj = [[NSPortMessage alloc] initWithSendPort:outPort
                                     receivePort:myPort components:nil];

    if (messageObj)
    {
        // Finish configuring the message and send it immediately.
        [messageObj setMsgId:setMsgid:kCheckinMessage];
        [messageObj sendBeforeDate:[NSDate date]];
    }
}
```

Configuring an NSMessagePort Object

To establish a local connection with an `NSMessagePort` object, you cannot simply pass port objects between threads. Remote message ports must be acquired by name. Making this possible in Cocoa requires registering your local port with a specific name and then passing that name to the remote thread so that it can obtain an appropriate port object for communication. Listing 3-16 shows the port creation and registration process in cases where you want to use message ports.

Listing 3-16 Registering a message port

```
NSPort* localPort = [[NSMessagePort alloc] init];

// Configure the object and add it to the current run loop.
[localPort setDelegate:self];
[[NSRunLoop currentRunLoop] addPort:localPort forMode:NSDefaultRunLoopMode];

// Register the port using a specific name. The name must be unique.
NSString* localPortName = [NSString stringWithFormat:@"MyPortName"];
[[NSMessagePortNameServer sharedInstance] registerPort:localPort
                     name:localPortName];
```

Configuring a Port-Based Input Source in Core Foundation

This section shows how to set up a two-way communications channel between your application’s main thread and a worker thread using Core Foundation.

Listing 3-17 shows the code called by the application’s main thread to launch the worker thread. The first thing the code does is set up a `CFMessagePortRef` opaque type to listen for messages from worker threads. The worker thread needs the name of the port to make the connection, so that string value is delivered to the entry point function of the worker thread. Port names should generally be unique within the current user context; otherwise, you might run into conflicts.

Listing 3-17 Attaching a Core Foundation message port to a new thread

```
#define kThreadStackSize        (8 *4096)

OSStatus MySpawnThread()
{
    // Create a local port for receiving responses.
    CFStringRef myPortName;
    CFMessagePortRef myPort;
    CFRunLoopSourceRef rlSource;
    CFMessagePortContext context = {0, NULL, NULL, NULL, NULL};
    Boolean shouldFreeInfo;

    // Create a string with the port name.
    myPortName = CFStringCreateWithFormat(NULL, NULL, CFSTR("com.myapp.MainThread"));

    // Create the port.
    myPort = CFMessagePortCreateLocal(NULL,
                myPortName,
                &MainThreadResponseHandler,
                &context,
                &shouldFreeInfo);

    if (myPort != NULL)
    {
        // The port was successfully created.
        // Now create a run loop source for it.
        rlSource = CFMessagePortCreateRunLoopSource(NULL, myPort, 0);

        if (rlSource)
        {
            // Add the source to the current run loop.
            CFRunLoopAddSource(CFRunLoopGetCurrent(), rlSource, kCFRunLoopDefaultMode);
            // Once installed, these can be freed.
            CFRelease(myPort);
            CFRelease(rlSource);
        }
    }

    // Create the thread and continue processing.
    MPTaskID        taskID;
    return(MPCreateTask(&ServerThreadEntryPoint,
                    (void*)myPortName,
                    kThreadStackSize,
                    NULL,
                    NULL,
                    NULL,
                    0,
                    &taskID));
}
```

With the port installed and the thread launched, the main thread can continue its regular execution while it waits for the thread to check in. When the check-in message arrives, it is dispatched to the main thread's `MainThreadResponseHandler` function, shown in Listing 3-18. This function extracts the port name for the worker thread and creates a conduit for future communication.

Listing 3-18 Receiving the checkin message

```
#define kCheckinMessage 100

// Main thread port message handler
CFDataRef MainThreadResponseHandler(CFMessagePortRef local,
                    SInt32 msgid,
                    CFDataRef data,
                    void* info)
{
    if (msgid == kCheckinMessage)
    {
        CFMessagePortRef messagePort;
        CFStringRef threadPortName;
        CFIndex bufferLength = CFDataGetLength(data);
        UInt8* buffer = CFAllocatorAllocate(NULL, bufferLength, 0);

        CFDataGetBytes(data, CFRangeMake(0, bufferLength), buffer);
        threadPortName = CFStringCreateWithBytes (NULL, buffer, bufferLength, kCFStringEncodingASCII, FALSE);

        // You must obtain a remote message port by name.
        messagePort = CFMessagePortCreateRemote(NULL, (CFStringRef)threadPortName);

        if (messagePort)
        {
            // Retain and save the thread's comm port for future reference.
            AddPortToListOfActiveThreads(messagePort);

            // Since the port is retained by the previous function, release
            // it here.
            CFRelease(messagePort);
        }

        // Clean up.
        CFRelease(threadPortName);
        CFAllocatorDeallocate(NULL, buffer);
    }
    else
    {
        // Process other messages.
    }

    return NULL;
}
```

With the main thread configured, the only thing remaining is for the newly created worker thread to create its own port and check in. Listing 3-19 shows the entry point function for the worker thread. The function extracts the main thread's port name and uses it to create a remote connection back to the main thread. The function then creates a local port for itself, installs the port on the thread's run loop, and sends a check-in message to the main thread that includes the local port name.

Listing 3-19 Setting up the thread structures

```
OSStatus ServerThreadEntryPoint(void* param)
{
    // Create the remote port to the main thread.
    CFMessagePortRef mainThreadPort;
    CFStringRef portName = (CFStringRef)param;

    mainThreadPort = CFMessagePortCreateRemote(NULL, portName);

    // Free the string that was passed in param.
    CFRelease(portName);

    // Create a port for the worker thread.
    CFStringRef myPortName = CFStringCreateWithFormat(NULL, NULL, CFSTR("com.MyApp.Thread-%d"),
MPCurrentTaskID());

    // Store the port in this thread’s context info for later reference.
    CFMessagePortContext context = {0, mainThreadPort, NULL, NULL, NULL};
    Boolean shouldFreeInfo;
    Boolean shouldAbort = TRUE;

    CFMessagePortRef myPort = CFMessagePortCreateLocal(NULL,
                myPortName,
                &ProcessClientRequest,
                &context,
                &shouldFreeInfo);

    if (shouldFreeInfo)
    {
        // Couldn't create a local port, so kill the thread.
        MPExit(0);
    }

    CFRunLoopSourceRef rlSource = CFMessagePortCreateRunLoopSource(NULL, myPort, 0);
    if (!rlSource)
    {
        // Couldn't create a local port, so kill the thread.
        MPExit(0);
    }

    // Add the source to the current run loop.
    CFRunLoopAddSource(CFRunLoopGetCurrent(), rlSource, kCFRunLoopDefaultMode);

    // Once installed, these can be freed.
    CFRelease(myPort);
    CFRelease(rlSource);

    // Package up the port name and send the check-in message.
    CFDataRef returnData = nil;
    CFDataRef outData;
    CFIndex stringLength = CFStringGetLength(myPortName);
    UInt8* buffer = CFAllocatorAllocate(NULL, stringLength, 0);

    CFStringGetBytes(myPortName,
                CFRangeMake(0,stringLength),
                kCFStringEncodingASCII,
                0,
                FALSE,
                buffer,
                stringLength,
                NULL);

    outData = CFDataCreate(NULL, buffer, stringLength);

    CFMessagePortSendRequest(mainThreadPort, kCheckinMessage, outData, 0.1, 0.0, NULL, NULL);

    // Clean up thread data structures.
    CFRelease(outData);
    CFAllocatorDeallocate(NULL, buffer);

    // Enter the run loop.
    CFRunLoopRun();
}
```

Once it enters its run loop, all future events sent to the thread's port are handled by the `ProcessClientRequest` function. The implementation of that function depends on the type of work the thread does and is not shown here.