Carbon-Cocoa Integration Guide

Cocoa > Design Guidelines

2007-10-31

Contents

Figures, Tables, and Listings

Introduction to Carbon-Cocoa Integration Guide

No matter which development environment you choose for developing applications—Cocoa or Carbon—you may find that the other development environment offers functionality you'd like to use in your application. Choosing the Cocoa or Carbon development environment to create new applications doesn't restrict you to using the API defined for that environment. You can use the Carbon API from a Cocoa application or the Cocoa API from a Carbon application. This document shows you how.

There are a number of reasons you might want to integrate Cocoa and Carbon in an application, including the following:

- You want to use existing code while getting the benefits of technologies offered by another framework.
- You're developing a common service that you want to make available to both Carbon and Cocoa.
- It's easier for you to do some tasks in Cocoa than in Carbon, or vice versa.
- You've already created a terrific user interface in one environment and you want to access it from the other environment.
- Your programming team consists of engineers with different skill sets—Cocoa and Carbon.

Who Should Read This Document?

This document assumes you are programming for Cocoa in Objective-C and does not discuss Java integration issues. To get the full benefit of this document, you should have experience programming in either the Cocoa or Carbon environment and you should also have some basic knowledge of the other environment. The "See Also" (page 8) section contains a list of documents to help you gain this knowledge.

Organization of This Document

You should be familiar with a few fundamental concepts before you begin to integrate Cocoa and Carbon in the same application. These concepts are covered in the following articles:

- "Carbon and Cocoa User Interface Communication" (page 9) discusses how Mac OS X communicates user events between Carbon and Cocoa application environments.
- "Preprocessing Mixed-Language Code" (page 11) lists filename extensions you can use when you mix programming languages in a project.
- "Interchangeable Data Types" (page 13) provides information on Foundation (Cocoa) and Core Foundation (Carbon) data types you can use interchangeably.
- "Using Carbon and Cocoa in the Same Application" (page 15) provides a brief overview of how to combine Carbon and Cocoa code in an application. Includes a discussion of C-callable wrapper functions.

The concepts discussed in the preceding articles are put into practice in the sample code provided in these articles:

- "Using Cocoa Functionality in a Carbon Application" (page 17) describes how to use Cocoa functionality unrelated to the user interface in a Carbon application.
- "Using Carbon Functionality in a Cocoa Application" (page 23) describes how to use Carbon functionality unrelated to the user interface in a Cocoa application.
- "Using a Cocoa User Interface in a Carbon Application" (page 29) describes tasks you must perform to enable a Cocoa user interface to work properly in Carbon.
- "Using a Carbon User Interface in a Cocoa Application" (page 37) describes tasks you must perform to enable a Carbon user interface to work properly in Cocoa.
- "HICocoaView: Using Cocoa Views in Carbon Windows" (page 43) describes how HICocoaView, introduced in Mac OS X v10.5, makes it possible to use Cocoa views in a Carbon window.
- "Using Cocoa in a Navigation Services Dialog" (page 51) describes how Carbon applications using Navigation Services in Mac OS X v10.5 can directly access features in the Cocoa classes `NSOpenPanel` and `NSSavePanel`.

See Also

For additional information on developing for Mac OS X, especially in Cocoa and Carbon, see the following documents:

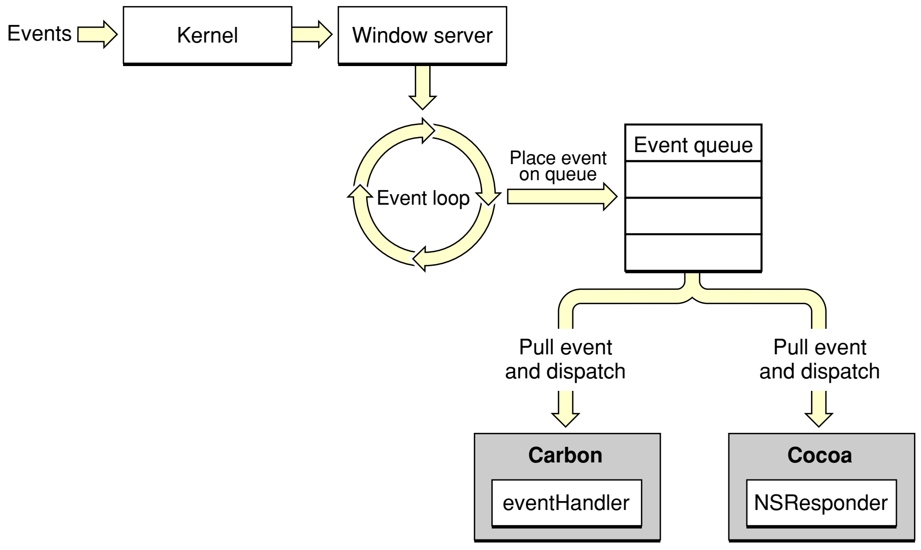
- *Getting Started with Cocoa* and *Getting Started with Carbon* provide a guided introduction and learning path for developers new to Cocoa and Carbon, respectively.
- *Getting Started with Tools* provides a guided introduction and learning path for developers new to Apple's integrated development environment (IDE).
- *Mac OS X Technology Overview* provides an orientation to the technologies available in Mac OS X, with links to relevant documentation. Appendix A, "Mac OS X Frameworks," lists the frameworks available to Mac OS X developers.
- *Memory Management Programming Guide for Cocoa* addresses the object-ownership policy and related techniques for creating, copying, retaining, and disposing of objects.

Carbon and Cocoa User Interface Communication

Prior to Mac OS X version 10.2, using Carbon windows in a Cocoa application or using Cocoa windows in a Carbon application posed a challenge because of the way user events are generated and forwarded to applications. This article provides an overview of event handling and outlines how Carbon and Cocoa communicate user events to each other.

Figure 1 shows the path of a user event (such as a click or a keypress) through the system to the Carbon and Cocoa application environments. The event originates when the device driver that controls an input device such as a mouse detects a user action and passes it to the window server.

Figure 1 The path of user events in Mac OS X



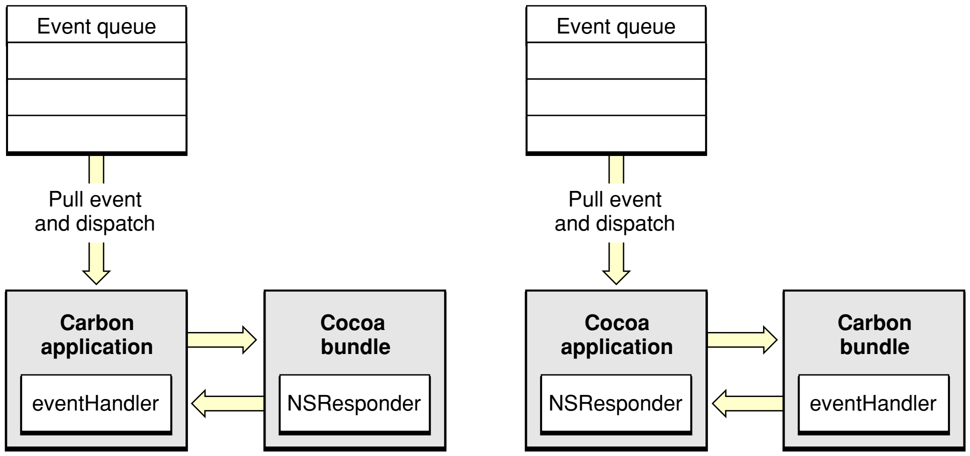
When the window server receives the event, it consults a database of currently open windows. It then sends the event to the event port of the run loop belonging to the process that owns the window in which the event occurred. The event manager gets the event from the run-loop port, packages the event in an appropriate form, and passes it to the event-handling mechanism specific to the application environment of the process. This mechanism ensures that the event is handled by the function or method associated with the control that is clicked (or key that is pressed).

Prior to Mac OS X version 10.2, events were passed to the application environment of the process. So when a Carbon application tried to use a Cocoa window, events for the Cocoa window were passed to the Carbon application. But because the Carbon application did not have a handler for events in the Cocoa window and had no way to communicate the event to the Cocoa environment that created the window, the event was dropped. The reverse was true for a Carbon window in a Cocoa application. Events for the Carbon window were passed to the Cocoa application, but the Cocoa application did not have a handler for the Carbon window and had no way to communicate the event to the Carbon environment.

Starting with Mac OS X version 10.2, the system automatically installs the appropriate handlers to allow Cocoa and Carbon to communicate events between the two environments. Figure 2 shows the communication path between Carbon and Cocoa environments for a Carbon application that uses a Cocoa source (on the left in the figure) and for a Cocoa application that uses a Carbon source (on the right in the figure).

Let's first look at how communicating user events works for a Cocoa window used in a Carbon application. The system automatically installs Carbon event handlers for the Cocoa window using a `WindowRef` object created for that purpose. When a user event (for example, a button click) occurs in the Cocoa window, the user event is passed to the Carbon application. The Carbon application dispatches the user event to the event handler for the `WindowRef` object for the Cocoa window. The system-supplied event handler knows how to package the event as a Cocoa NSEvent object. It then passes the NSEvent object to the window for processing using the normal Cocoa event-processing mechanisms, including the responder chain for events not targeted at a specific control. In the case of a button click, the button receives the button-click event and handles it using the normal Cocoa target-action mechanism.

Figure 2 Events communicated between Carbon and Cocoa



Conversely, for a Carbon window used in a Cocoa application, the system automatically creates a Cocoa NSWindow object to represent the Carbon window. When a user clicks a button in the Carbon window, an NSEvent object for the button click is passed to the Cocoa application. Cocoa's normal event-handling mechanisms pass the event to the system-supplied NSWindow object that corresponds to the Carbon window. The NSWindow object knows how to create a Carbon event and pass it to the event handler of the Carbon window. From there, the event is processed through the Carbon event target containment hierarchy.

In summary, Carbon and Cocoa can share the same window because Mac OS X has mechanisms for automatically translating between Cocoa and Carbon events at the user interface element level, rather than just at the application level.

For more information on application environments and the system architecture, see *Mac OS X Technology Overview*. For more information on Carbon events, see *Carbon Event Manager Programming Guide*. For more information on Cocoa events, see the Cocoa programming topic *Cocoa Event-Handling Guide*.

Preprocessing Mixed-Language Code

Carbon and Cocoa use different languages for their APIs. Carbon uses C while Cocoa uses Objective-C. In addition, Carbon programmers may prefer to use C++. When you build an application that uses any combination of C, Objective-C, Objective-C++, and C++ files, you must make sure the compiler performs the appropriate preprocessing. The filename extension of the file you are compiling indicates the kind of compilation that is done, as shown in Table 1.

Table 1 Filename extensions and compilation

Filename extension	Indicates to the compiler
`.c`	C source code that must be preprocessed
`.m`	Objective-C source code
`.mm`	Objective-C++ source code
`.h`	Header file (not to be compiled or linked)
`.cc`, `.cp`, `.cxx`, `.cpp`, `.c++`, `.C`	C++ source code that must be preprocessed

In an Xcode project, you can override the filename extension of a source file by changing its file type. The file type determines how Xcode preprocesses and compiles the file. For example, suppose you want to add Objective-C code to a C source file with a `.c` filename extension. You can examine and change the file type of this source file as follows:

1. Select the C source file in the project window.
2. Open the File Info window and select the General pane.
3. Find the “File Type” setting.
4. Change its value to “sourcecode.c.objc”.

Now Xcode will compile the Objective-C code in the file.

You can also direct Xcode to handle all the source files in a project as one language, regardless of their filename extensions and file types. For example, suppose you want to add Objective-C code to many of your C++ source files. To specify that the compiler treat all source files as Objective-C++ files:

1. Select the project in the Groups and Files pane of the project window.
2. Open the the Project Info window and select the Build pane.
3. Find the “Compile Sources As” setting.
4. Change its value from “According to File Type” to “Objective-C++“.

Now Xcode will compile the Objective-C code in all your C++ files. You can also change this setting on a per-target basis.

For complete Xcode documentation, see the Tools category of the ADC Reference Library. The Xcode user guide is also available from the Help menu in the Xcode application.

Interchangeable Data Types

There are a number of data types in the Core Foundation framework (Carbon) and the Foundation framework (Cocoa) that can be used interchangeably. This means that you can use the same data structure as the argument to a Core Foundation function call or as the receiver of an Objective-C message invocation. For example, `NSLocale` (see *NSLocale Class Reference*) is interchangeable with its Core Foundation counterpart, CFLocale (see *CFLocale Reference*). Therefore, in a method where you see an `NSLocale *` parameter, you can pass a `CFLocaleRef`, and in a function where you see a `CFLocaleRef` parameter, you can pass an `NSLocale` instance. You cast one type to the other to suppress compiler warnings, as illustrated in the following example.

```
NSLocale *gbNSLocale = [[NSLocale alloc] initWithLocaleIdentifier:@"en_GB"];
CFLocaleRef gbCFLocale = (CFLocaleRef) gbNSLocale;
CFStringRef cfIdentifier = CFLocaleGetIdentifier (gbCFLocale);
NSLog(@"cfIdentifier: %@", (NSString *)cfIdentifier);
// logs: "cfIdentifier: en_GB"
CFRelease((CFLocaleRef) gbNSLocale);

CFLocaleRef myCFLocale = CFLocaleCopyCurrent();
NSLocale * myNSLocale = (NSLocale *) myCFLocale;
[myNSLocale autorelease];
NSString *nsIdentifier = [myNSLocale localeIdentifier];
CFShow((CFStringRef) [@"nsIdentifier: " stringByAppendingString:nsIdentifier]);
// logs identifier for current locale
```

Note from the example that the memory management functions and methods are also interchangeable—you can use `CFRelease` with a Cocoa object and `release` and `autorelease` with a Core Foundation object.

Note: When using garbage collection, there are important differences to how memory management works for Cocoa objects and Core Foundation objects. See Using Core Foundation with Garbage Collection for details.

Data types that can be used interchangeably are also referred to as **toll-free bridged** data types. Toll-free bridging has been available since Mac OS X v10.0. Table 1 provides a list of the data types that are interchangeable between Core Foundation and Foundation. For each pair, the table also lists the version of Mac OS X in which toll-free bridging between them became available.

Table 1 Data types that can be used interchangeably between Core Foundation and Foundation

Core Foundation type	Foundation class	Availability
`CFArrayRef`	`NSArray`	Mac OS X v10.0
`CFAttributedStringRef`	`NSAttributedString`	Mac OS X v10.4
`CFCalendarRef`	`NSCalendar`	Mac OS X v10.4
`CFCharacterSetRef`	`NSCharacterSet`	Mac OS X v10.0

Core Foundation type	Foundation class	Availability
`CFDataRef`	`NSData`	Mac OS X v10.0
`CFDateRef`	`NSDate`	Mac OS X v10.0
`CFDictionaryRef`	`NSDictionary`	Mac OS X v10.0
`CFErrorRef`	`NSError`	Mac OS X v10.5
`CFLocaleRef`	`NSLocale`	Mac OS X v10.4
`CFMutableArrayRef`	`NSMutableArray`	Mac OS X v10.0
`CFMutableAttributedStringRef`	`NSMutableAttributedString`	Mac OS X v10.4
`CFMutableCharacterSetRef`	`NSMutableCharacterSet`	Mac OS X v10.0
`CFMutableDataRef`	`NSMutableData`	Mac OS X v10.0
`CFMutableDictionaryRef`	`NSMutableDictionary`	Mac OS X v10.0
`CFMutableSetRef`	`NSMutableSet`	Mac OS X v10.0
`CFMutableStringRef`	`NSMutableString`	Mac OS X v10.0
`CFNumberRef`	`NSNumber`	Mac OS X v10.0
`CFReadStreamRef`	`NSInputStream`	Mac OS X v10.0
`CFRunLoopTimerRef`	`NSTimer`	Mac OS X v10.0
`CFSetRef`	`NSSet`	Mac OS X v10.0
`CFStringRef`	`NSString`	Mac OS X v10.0
`CFTimeZoneRef`	`NSTimeZone`	Mac OS X v10.0
`CFURLRef`	`NSURL`	Mac OS X v10.0
`CFWriteStreamRef`	`NSOutputStream`	Mac OS X v10.0

Note: Not all data types are toll-free bridged, even though their names might suggest that they are. For example, `NSRunLoop` is not toll-free bridged to CFRunLoop, `NSBundle` is not toll-free bridged to CFBundle, and `NSDateFormatter` is not toll-free bridged to CFDateFormatter.

Using Carbon and Cocoa in the Same Application

It has always been possible to integrate Cocoa and Carbon functionality in the same application, at least when it comes to functionality that doesn't handle user interface elements.

Using Carbon in a Cocoa Application

For a Cocoa application to call Carbon functions, the only requirements are that the compiler has access to the appropriate header files and the application is linked against the appropriate frameworks. To access Carbon functionality, you can simply import `Carbon.h` and link against the Carbon framework. Since Objective-C is a superset of ANSI C, calling Carbon functions from a Cocoa application is easy (although prior to Mac OS X v10.2 the functions could not be user interface functions).

Using Cocoa in a Carbon Application

A Carbon application, by taking a few extra steps, can use many Cocoa and Objective-C technologies. To access Cocoa functionality, you can simply import `Cocoa.h` and link against the Cocoa framework. To access other Objective-C technologies, you may need to import additional headers and link against additional frameworks. For example, to use Web Kit, you need to import `WebKit.h` and link against the WebKit framework.

You also need to take these steps:

- Prepare your Carbon application to use Cocoa by calling the `NSApplicationLoad` function. Typically, you do this in your main function before executing any other Cocoa code.
- In functions where you're using Cocoa, allocate and initialize an `NSAutoreleasePool` object and release it when it is no longer needed. Note that if your application is running in Mac OS X v10.4 or later, an autorelease pool already exists when your functions are called, directly or indirectly, by the toolbox. For example, an `NSAutoReleasePool` object is automatically established and drained in the following locations:
    - `RunApplicationEventLoop`
    - `RunAppModalLoopForWindow`
    - `ModalDialog`
    - Window drag and resize tracking
    - Control tracking and indicator dragging
    - Across dispatch of an event by the event dispatcher target
    - When a window's compositing views are redrawn

- Use the Objective-C compiler to build those parts of your project that use Cocoa. In Xcode, there are several ways to do this:
  - ❏ Use Objective-C filename extensions.
  - ❏ Use the File Info panel to set the file type of a source file to `sourcecode.c.objc`.
  - ❏ Change the project or target build setting "Compile Sources As" to Objective-C.

Using C-Callable Wrapper Functions

When integrating Objective-C code into a Carbon project, a common approach is to write C-callable wrapper functions (or simply C-wrapper functions) for the Objective-C portion of the code. In the context of Carbon and Cocoa integration, a **C-callable wrapper function** is a C function whose body contains Objective-C code that allows data to be passed to or obtained from Cocoa. These C-wrapper functions can be placed in a separate Objective-C source file and compiled using the Objective-C compiler.

Let's look at a typical scenario for a Carbon application that accesses functionality provided in a Cocoa source file. The Cocoa source must contain all the necessary Objective-C code for its classes and methods. It must also contain a C-callable wrapper function for each method whose functionality is needed by the Carbon application. For example, for a `changeText:` method that takes a string and manipulates it in some way, the C-callable wrapper function would look similar to the following:

```
OSStatus changeText (CFStringRef message)
{
    NSAutoreleasePool *localPool;

    localPool = [[NSAutoreleasePool alloc] init];
    [[Controller sharedController] changeText:(NSString *)message];
    [localPool release];
    return noErr;
}
```

Note: The C-wrapper function usually allocates and initializes an `NSAutoreleasePool` object and then releases it when it is no longer needed, as shown in the `changeText` function. This is a requirement for C-wrapper functions that are called directly by your Carbon application, in order to achieve correct memory management.

In summary, here's how C-callable wrapper functions are used to allow Carbon applications to access Cocoa functionality:

1. Access to the Cocoa functionality is provided in an Objective-C source file. The file contains C-callable wrapper functions for any Cocoa method that's needed by the Carbon application.

2. The Carbon application invokes the C-wrapper functions as needed.

Using Cocoa Functionality in a Carbon Application

This article describes how a Carbon application can use Cocoa functionality that is unrelated to the user interface. You can access Cocoa functionality in a Carbon application in Mac OS X version 10.1 and later. You need to perform two major tasks to use Cocoa functionality in a Carbon application:

- Write a C-callable wrapper function for any Cocoa method whose functionality you want to access from your Carbon application. See "Writing the Cocoa Source" (page 18) for details.
- Write Carbon code that calls the C-wrapper function that initializes Cocoa. See "Calling C-Wrapper Functions From Your Carbon Application" (page 21) for details.

The tasks described in the following sections are illustrated using sample code taken from a working application called Spelling Checker. The sample application uses Cocoa's spell checking functionality. See "About the Spelling Checker Application" (page 17) for a description of the application. You can download the code for *SpellingChecker-CarbonCocoa*.

Although a lot of the code from the Spelling Checker application is shown in the listings in this article, not all of the code is included or explained. For example, none of the code that handles the Carbon window has been included. To see exactly how the Carbon and Cocoa pieces fit together, you should download the project.

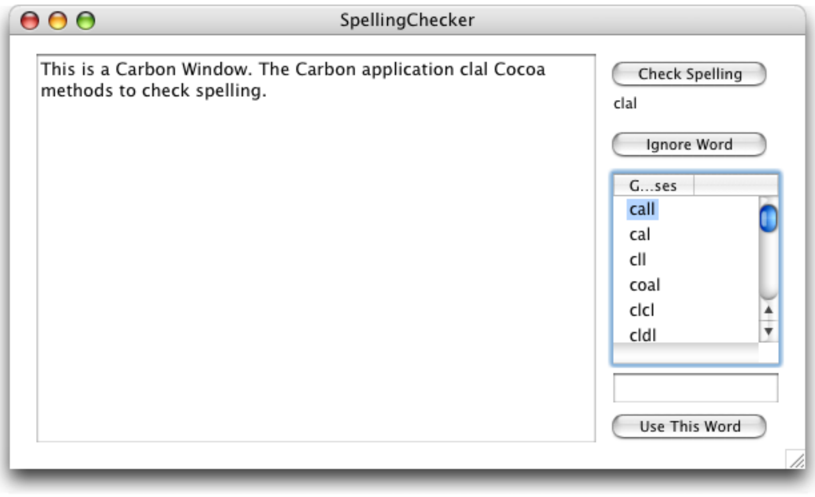
About the Spelling Checker Application

The sample Carbon application, Spelling Checker, provides spelling checking functionality for text typed into a window. The user interface is shown in Figure 1 (page 18). The Spelling Checker window is a Carbon window, created with Interface Builder. The user can type text into the large text box on the left side of the window.

To check spelling, the user clicks the Check Spelling button. The first misspelled word is displayed below the button, as shown in Figure 1 ("clal"). Suggestions for a replacement word are shown in the Guesses list. The user can choose to:

- Ignore the misspelled word by clicking the Ignore Word button.
- Replace the misspelled word by selecting a word from the list of guesses and double-clicking.
- Specify another word to use by typing a word and clicking the Use This Word button.

Figure 1 The user interface for the Spelling Checker application



This is a Carbon Window. The Carbon application clal Cocoa methods to check spelling.

Spelling checking functionality is provided by the Cocoa frameworks and accessed through C-callable wrapper functions, but called from the Carbon application.

Note: The text box in the sample application is a Unicode TextEdit control. For a more complex application, it is better to use the text editing capabilities provided by the Multilingual Text Engine (MLTE) API.

Writing the Cocoa Source

Writing the Cocoa source requires performing the tasks described in the following sections:

- "Creating a New Cocoa Source File Using Xcode" (page 18)
- "Identifying Cocoa Methods" (page 19)
- "Writing C-Callable Wrapper Functions" (page 19)

Creating a New Cocoa Source File Using Xcode

To make a Cocoa source file using Xcode, do the following:

1. Open your Carbon project in Xcode.
2. Choose File > New File.
3. Select Empty File in Project in the New File window and click the Next button.
4. Name the file so it has the appropriate `.m` extension. The sample code filename is `SpellCheck.m`.

   Recall from "Preprocessing Mixed-Language Code" (page 11) that the `.m` extension indicates to the compiler that the code is Objective-C.

5. Add the following statements to your new file:

```
#include <Carbon/Carbon.h>
#include <Cocoa/Cocoa.h>
```

As long as you create your source file using Xcode, you should not need to modify build settings and property list values.

Identifying Cocoa Methods

You need to identify the Cocoa methods that provide the functionality your Carbon application needs. For each of the methods you identify, you'll need to write a C-callable wrapper function.

The Spelling Checker application requires the functionality provided by the following methods:

- `uniqueSpellDocumentTag` returns a tag for a document. This tag is guaranteed to be unique. Using a tag with each document ensures that the spelling checking operation is unique for a document.
- `checkSpellingOfString:startingAt:` starts the search for a misspelled word in a string, starting at the specified location. This method returns the range of the first misspelled word.
- `checkSpellingOfString:startingAt:language:wrap:inSpellDocumentWithTag:wordCount:` starts the search for a misspelled word in a string, starting at the specified location and using a number of other options. This method returns the range of the first misspelled word.
- `ignoreWord:inSpellDocumentWithTag:` adds a word to the list of words to be ignored when checking a document's spelling.
- `setIgnoredWords:inSpellDocumentWithTag:` initializes the list of ignored words for a document to an array of words to ignore.
- `ignoredWordsInSpellDocumentWithTag:` returns the array of ignored words for a document.
- `guessesForWord:` returns an array of suggested spellings for a misspelled word.
- `closeSpellDocumentWithTag:` is called when a document closes to make sure the ignored-word list associated with the document is cleaned up.

For additional information, see *NSSpellChecker Class Reference*.

You also need to identify any other methods that are needed to implement the Cocoa functionality. For example, the class method `sharedSpellChecker` returns an instance of NSSpellChecker.

Writing C-Callable Wrapper Functions

After you have identified the Cocoa methods that provide the functionality you want to use, you need to write C-callable wrapper functions for those methods.

For the Spelling Checker application, there are eight Cocoa methods (see "Identifying Cocoa Methods" (page 19)) that provide functionality to manage and check spelling in a document. In order for the Carbon portion of the application to access the Cocoa methods, you need to write C-callable wrapper functions and put them in the Cocoa source file. You also need to declare the functions in a shared header file. Table 1 lists the names of the C-callable wrapper functions in the Spelling Checker application.

Table 1 C-callable wrapper functions for Cocoa methods

C-callable wrapper function	Cocoa method
`UniqueSpellDocumentTag`	`uniqueSpellDocumentTag`
`CloseSpellDocumentWithTag`	`closeSpellDocumentWithTag:`
`CheckSpellingOfString`	`checkSpellingOfString: startingAt:`
`CheckSpellingOfStringWithOptions`	`checkSpellingOfString:startingAt: language:wrap:inSpellDocumentWithTag: wordCount:`
`IgnoreWord`	`ignoreWord: inSpellDocumentWithTag:`
`SetIgnoredWords`	`setIgnoredWords: inSpellDocumentWithTag:`
`CopyIgnoredWordsIn-SpellDocumentWithTag`	`ignoredWordsInSpellDocumentWithTag:`
`GuessesForWords`	`guessesForWord:`

Listing 1 shows the C-callable wrapper function `UniqueSpellDocumentTag`. Note the code for the autorelease pool. For a Cocoa method used by a Carbon application, you must set up an autorelease pool each time it's used.

Listing 1 A C-callable wrapper function for the uniqueSpellDocumentTag: method

```
int     UniqueSpellDocumentTag ()
{
    int  tag;

    NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];
    tag  = [NSSpellChecker uniqueSpellDocumentTag];
    [pool release];

    return (tag);
}
```

All the other C-callable wrapper functions for the Spelling Checker application are written in the same manner as shown in Listing 1, using these guidelines:

- The C-wrapper function must have parameters that match what's needed by the Cocoa method. For examples, see Listing 2. The C-wrapper function parameters `stringToCheck` and `startingOffset` match the two parameters required by the `checkSpellingOfString:startingAt:` method.
- The C-wrapper function must allocate and initialize an NSAutoreleasePool object and then release it when it is no longer needed. This is a requirement for a Cocoa method that's used by a Carbon application. You can see examples of this in Listing 1 and Listing 2.
- The C-wrapper function must return the data returned by the Cocoa method it wraps. For example, the `UniqueSpellDocumentTag` function in Listing 1 returns the tag value obtained from the `uniqueSpellDocumentTag` method; the `CheckSpellingOfString` function in Listing 2 returns the range obtained from the `checkSpellingOfString:startingAt:` method.

- Where appropriate, the C-wrapper function can use toll-free bridged (interchangeable) data types. For example, the C-wrapper function in Listing 2 takes a `CFStringRef` value as a parameter, but casts it to `NSString *` when passing the string to the Cocoa method.

Listing 2 A C-callable wrapper function for the checkSpellingOfString:startingAt: method

```
CFRange  CheckSpellingOfString (CFStringRef stringToCheck,
                               int startingOffset)
{
    NSRange range = {0,0};

    NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];
    range = [[NSSpellChecker sharedSpellChecker]
                    checkSpellingOfString:(NSString *) stringToCheck
                    startingAt:startingOffset];
    [pool release];
    return ( *(CFRange*)&range );
}
```

You will also want to create a header file that contains the C-callable wrapper function declarations that can be included in the appropriate source files.

The code for the rest of the C-callable wrapper functions needed for the Spelling Checker application are in the `SpellCheck.m` Cocoa source file. You can download the code for *SpellingChecker-CarbonCocoa*.

Calling C-Wrapper Functions From Your Carbon Application

You can use the C-callable wrapper functions as needed in your Carbon application. Listing 3 shows how to call a C-callable wrapper function (`CheckSpellingOfString`) from your Carbon application's event handler. (You can see this code in context by downloading the Spelling Checker application from the developer sample code website.) A detailed explanation of each numbered line of code appears following the listing.

Listing 3 Calling a C-wrapper function from your Carbon application

```
if (command.commandID == 'Spel')                                      // 1
{
    GetControlByID (window, &controlID, &control);                    // 2
    err = GetControlData (control, 0,
               kControlStaticTextCFStringTag,
               sizeof(CFStringRef),
               &stringToSpellCheck,
               &count);                                               // 3
    if (err == noErr)
    {
        windowInfo->range = CheckSpellingOfString (stringToSpellCheck, 0);   // 4
        if (windowInfo->range.length > 0)                             // 5
           SetMisspelledWord (window, stringToSpellCheck, &windowInfo->range);
        else
            windowInfo->range.location = 0;
    }
}
```

Here's what the code does:

1. Checks to see if the command ID is the one that's issued when the user clicks the Check Spelling button.
2. Calls the Control Manager function `GetControlByID` to obtain the `ControlRef` of the Unicode TextEdit control. This is the text box that contains the text typed by the user that needs to have its spelling checked. See Figure 1 (page 18).
3. Calls the Control Manager function `GetControlData` to obtain the string that the user typed in the text box.
4. Calls the C-wrapper function `CheckSpellingOfString`. Recall that this C-wrapper function wraps the Cocoa method `checkSpellingOfString:startingAt:`.
5. Uses the location information returned from the C-wrapper function to set the location of the misspelled word if one is found.

Using Carbon Functionality in a Cocoa Application

Objective-C is a superset of ANSI C, so calling Carbon functions from a Cocoa application is easy as long as they are not user interface functions. A Cocoa application can always call low-level Carbon functions because Cocoa already links against the Application Services framework. To use high-level Carbon functions, a Cocoa application must import `Carbon.h` and link against the Carbon framework.

The following sections describe some of the situations in which you can use Carbon functions (other than user interface ones) in a Cocoa application:

- "Working With QuickTime Movies" (page 23)
- "Accessing a Resource Fork From Cocoa" (page 24)
- "Using the FSRef Data Type" (page 25)
- "Managing Core Foundation Objects in Cocoa" (page 26)

The situations in which you can call Carbon functions in a Cocoa application are not limited to the examples in this article. The examples illustrate the variety of ways that you can use non-UI Carbon functions in a Cocoa application. If you want to use a Carbon interface in a Cocoa application, read "Using a Carbon User Interface in a Cocoa Application" (page 37).

Working With QuickTime Movies

The Cocoa NSMovieView class displays an NSMovie (a wrapper for a QuickTime movie) in a frame and provides methods to play and edit the movie. Although there are methods for editing, setting the view size, setting the controller, setting play modes, and handling sound, the methods in the NSMovieView class do not access all the functionality available for working with QuickTime movies. For example, there are no methods to set the language of a QuickTime movie if alternate language tracks are available. However, you can call any QuickTime function from your Cocoa application, such as the function `SetMovieLanguage`. All you need to do is to initialize the Movie Toolbox by calling the QuickTime function `EnterMovies`.

See *QuickTime Framework Reference* for information on the QuickTime functions you can call from your application.

Listing 1 shows one example of calling QuickTime functions from a Cocoa method. The code in the listing builds an array of track media types for a QuickTime movie. The track media types are displayed in the NSTableView control in the movie's properties window. An explanation of each numbered line of code appears following the listing.

Listing 1 Building an array of track media types for a QuickTime movie

```
// Before you call QuickTime functions you must initialize the
// Movie Toolbox by calling the function EnterMovies();

- (void) myBuildTrackMediaTypesArray:(NSMovie *) movie
```

```
{
    short i;
    Movie qtmovie = [movie QTMovie];

    for (i = 0; i < GetMovieTrackCount (qtmovie); ++i)                  // 1
    {
        Str255  mediaName;
        OSErr   myErr;
        Track   movieTrack = GetMovieIndTrack (qtmovie, i+1);           // 2
        Media   trackMedia = GetTrackMedia (movieTrack);                // 3
        MediaHandler trackMediaHandler = GetMediaHandler(trackMedia);

        myErr = MediaGetName (trackMediaHandler, mediaName, 0, NULL);   // 4
        [myMovieTrackMediaTypesArray insertObject:[
                        NSString stringWithCString:&mediaName[1]
                        length:mediaName[0]]
                    atIndex:i];                                         // 5
    }
}
```

Here's what the code does:

1. Calls the QuickTime function `GetMovieTrackCount` to obtain the number of tracks in the movie.
2. Calls the QuickTime function `GetMovieIndTrack` to determine the track identifier for a track. Note that tracks start at an index value of 1.
3. Calls the QuickTime function `GetTrackMedia` to obtain the media structure that contains sample data for the track.
4. Calls the QuickTime function `MediaGetName` to obtain the name (`Str255`) of the media type.
5. Adds the media name to the NSArray as an NSString.

For more information, see *NSMovieView Class Reference*.

Accessing a Resource Fork From Cocoa

A Cocoa application that works with legacy files may need to read the resource fork of a Mac OS 9 file and then parse the resource data. The Resource Manager is a Carbon API, so you can call the appropriate functions from within your Cocoa code, as shown in Listing 2. Once you read the data you can call the Cocoa method `stringWithCString:length:` to obtain an NSString that you can then parse.

Listing 2 Calling Resource Manager functions from a Cocoa application

```
    FSRef ref;
    NSString* theFilePath;    // the full path of the resources file
    if (FSPathMakeRef ([theFilePath fileSystemRepresentation], &ref, NULL)
        == noErr)
    {
        short res = FSOpenResFile (&ref, fsRdPerm);
        if (ResError() == noErr)
        {
            // Code that calls Resource Manager functions to read resources
            // goes here.
            CloseResFile(res);
        }
    }
```

Using the FSRef Data Type

The File Manager in Carbon uses the `FSRef` data type for specifying the name and location of a file or directory. When you call Carbon functions from a Cocoa application, you may need to pass an `FSRef` as a parameter to one of the functions, or you may receive an `FSRef` as a return value. For example, if you call the Alias Manager function `FSResolveAliasFile`, you must supply an `FSRef` that identifies the alias. An `FSRef` is an opaque 80-byte structure, so typically a pointer is used to pass an `FSRef` as a parameter.

Given a path to a file that already exists, you can use the code in Listing 3 to obtain an `FSRef` that identifies the file. An explanation for each numbered line of code appears following the listing.

Listing 3 A Cocoa method to convert a path into an FSRef

```
- (BOOL) myMakeFSRef:(FSRef *) outFSRef fromPath:(NSString *)inPath
{
    OSStatus status = noErr;
    status = FSPathMakeRef ([inPath fileSystemRepresentation],
                           outFSRef,
                           NULL);                                     // 1
    return status == noErr;                                           // 2
}
```

Here's what the code does:

1. Calls the File Manager function `FSPathMakeRef` to convert a path into an `FSRef`. The `outFSRef` parameter must point to an actual `FSRef` structure.

2. Returns `YES` if the conversion was successful.

Given an `FSRef` to a file that already exists, you can use the code in Listing 4 to obtain a URL. An explanation for each numbered line of code appears following the listing.

Listing 4 A Cocoa method to convert an FSRef into a URL

```
- (NSURL *) myCreateURLFromFSRef:(FSRef *)inFSRef
{
    NSURL* url = nil;
    UInt8 path[PATH_MAX];
    OSStatus status = noErr;

    status = FSRefMakePath (inFSRef, (UInt8*)path, sizeof(path));    // 1
    if (status == noErr) {
        url = [NSURL fileURLWithPath: [NSString stringWithUTF8String:path]];  // 2
    }
    return url;                                                       // 3
}
```

Here's what the code does:

1. Calls the File Manager function `FSRefMakePath` to convert the `FSRef` into a path.
2. Uses the path to create a new `NSURL` object.
3. Returns an `NSURL` object if the conversion was successful.

For complete documentation of the File Manager, see *File Manager Reference*.

Managing Core Foundation Objects in Cocoa

Cocoa and Core Foundation use similar memory allocation conventions to allocate, retain, and release objects. In general, Core Foundation functions that have Copy or Create in their name return values the caller must release, while other functions return values the caller should not release. Cocoa objects created with `alloc`, `copy`, or `new` methods must be released by the caller, while all other return values should not be released by the caller. In addition, there is a set of data types that can be used interchangeably; these are referred to as toll-free bridged data types. (See "Interchangeable Data Types" (page 13) for a list.) As a result, you can use functions and methods from both environments in the same application.

The following code calls the Cocoa method `initWithCharacters` to initialize a newly allocated NSString. After the code that uses the string is executed, you need to release the string.

```
NSString *str = [[NSString alloc] initWithCharacters: ...];
// Your code that uses the string goes here.
[str release];
```

You can achieve the same result by calling the following Carbon code. This code uses the Core Foundation function `CFStringCreateWithCharacters`.

```
CFStringRef str = CFStringCreateWithCharacters(...);
// Your code that uses the string goes here.
CFRelease (str);
```

The following code calls the Core Foundation function `CFStringCreateWithCharacters`, casts the returned string to a Cocoa NSString, and releases the string using the Cocoa method `release`.

```
NSString *str = (NSString *) CFStringCreateWithCharacters(...);
// Your code that uses the string goes here.
[str release];
```

Similarly, the following code intermixes Core Foundation and Cocoa but calls the Cocoa method `autorelease` to dispose of the string.

```
NSString *str = (NSString *) CFStringCreateWithCharacters(...);
// Your code that uses the string goes here.
[str autorelease];
```

Note: Because Core Foundation has no concept of autoreleasing, a larger percentage of Core Foundation functions are Copy or Create functions as compared to the Cocoa methods that use `alloc`, `copy`, or `new`. You must make sure your Cocoa code releases Core Foundation objects, when appropriate, using either the `release` or `autorelease` method.

For a more information, see either the Cocoa programming topic *Memory Management Programming Guide for Cocoa* or the Core Foundation programming topic *Memory Management Programming Guide for Core Foundation*.

Using a Cocoa User Interface in a Carbon Application

You can use a Cocoa user interface in a Carbon application starting with Mac OS X version 10.2. The system provides code that allows Cocoa and Carbon to communicate user events to each other, so there are only a few tasks you must perform to enable the Cocoa user interface to work properly in Carbon. Most tasks are similar to what you would do to use non-UI Cocoa functionality in a Carbon application—that is, writing C-callable wrapper functions and calling them. (See "Using Cocoa Functionality in a Carbon Application" (page 17).)

It is important to recognize that embedding a Cocoa NSView inside a Carbon window is not supported in Mac OS X version 10.4 and earlier. For more information, see "HICocoaView: Using Cocoa Views in Carbon Windows" (page 43).

To use a Cocoa user interface in a Carbon application, you need to perform two major tasks:

- Write a Cocoa source file that contains the interface you want to use, the Cocoa methods that support the interface, and C-callable wrapper functions that allow access to the Cocoa functionality needed by the Carbon application. This is described in detail in "Writing the Cocoa Source Files" (page 30).
- Write Carbon code that provides handlers, as appropriate, for the interface. See "Setting Up the Carbon Application to Use the Cocoa Interface" (page 35) for details.

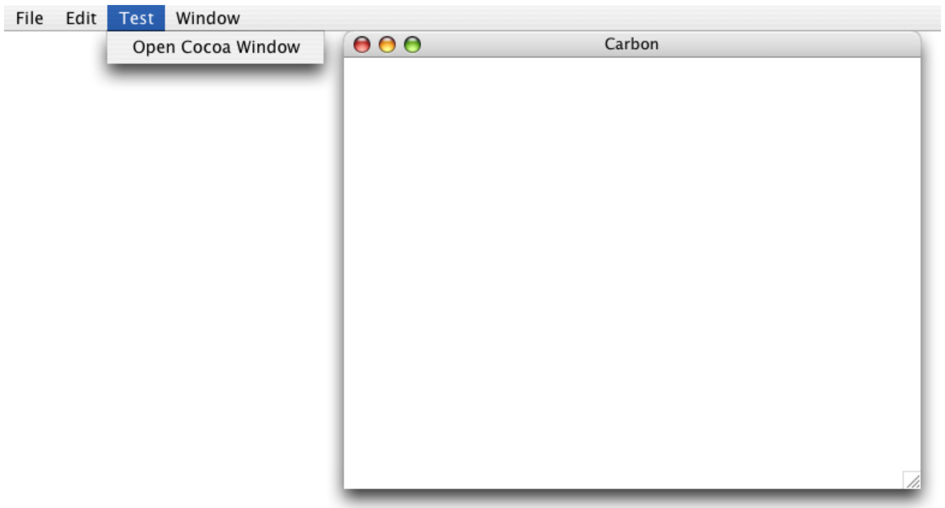
The tasks described in the following sections are illustrated using sample code taken from a working application called CocoaInCarbon. See "About the CocoaInCarbon Application" (page 29) for a description of the application. You can download the code for *CocoaInCarbon*.

Keep in mind that many parts of the CocoaInCarbon application are specific to the sample application; you need to customize the code for your own purposes. Although most of the code from the CocoaInCarbon application is shown in the listings in this article, not all of the code is included.

About the CocoaInCarbon Application

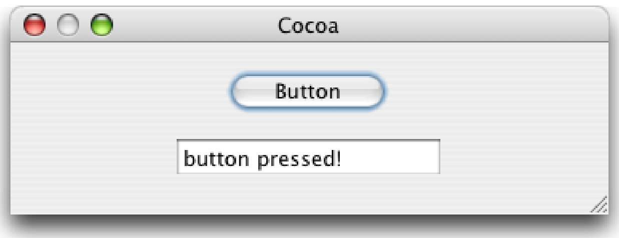
When you look at the code in the subsequent sections, it may be helpful to have an idea of how the CocoaInCarbon application behaves and what the user interface looks like. When the application is launched, an empty window appears. This window, shown in Figure 1, is a Carbon window defined in a nib file created with Interface Builder. The application provides a Test menu, that contains one command, Open Cocoa Window.

Figure 1 The Carbon user interface for the CocoaInCarbon application



When the user chooses Open Cocoa Window from the Test menu, the application calls the C-wrapper function that opens the Cocoa window (shown in Figure 2) and makes the window active.

Figure 2 The Cocoa window for the CocoaInCarbon application



When the user clicks the button shown in Figure 2, the mouse event is received by the Carbon application, which automatically dispatches the event to Cocoa. Specifically, the button receives the mouse event and sends its action method. The action method triggers a button command handler provided by the Carbon application. The button command handler calls a C-wrapper function that sends the text "button pressed!" to the text field below the button, as shown in Figure 2.

Although the sample application is obviously a contrived example, it illustrates the extent to which Carbon and Cocoa can communicate with each other.

Writing the Cocoa Source Files

Writing the Cocoa source requires performing the tasks described in the following sections:

- "Creating a Cocoa Source File Using Xcode" (page 31)
- "Writing a Common Header File" (page 31)

Creating a Cocoa Source File Using Xcode

You need to make a Cocoa source file that contains all the Cocoa functionality needed by your Carbon application. Follow these steps to create a Cocoa source file:

1. Open your Carbon project in Xcode.
2. Choose File > New File.
3. Select Empty File in Project in the New File window and click the Next button.
4. Name the file so it has the appropriate `.m` extension. The sample code filename is `Controller.m`.

   Recall from "Preprocessing Mixed-Language Code" (page 11) that the `.m` extension indicates to the compiler that the code is Objective-C.
5. Create any other files you need for the application. For example, the Cocoa source created for the CocoaInCarbon application has an interface file, `Controller.h`. You must create this file and import it to the `Controller.m` file by adding an import statement to the `Controller.m` file.

As long as you create your source file using Xcode, you should not need to modify build settings and property list values.

Writing a Common Header File

Both the Cocoa and the Carbon sources need to have a copy of the header file that declares any constants used to communicate events between them. The constant that defines the button-press event in the CocoaInCarbon application is one such constant. The C-callable wrapper function declarations could also be in this shared file. Listing 1 shows the contents of the header file (`CocoaStuff.h`) that must be included in both the Cocoa and the Carbon source for the CocoaInCarbon project.

Listing 1 The contents of the common header for Cocoa and Carbon

```
enum {
    kEventButtonPressed = 1
};

//Declare the wrapper functions
OSStatus initializeCocoa(OSStatus (*callBack)(int));
OSStatus orderWindowFront(void);
OSStatus changeText(CFStringRef message);
```

Implementing the Controller

The code to implement the controller for the Cocoa source in the CocoaInCarbon application is shown in Listing 2. The critical item in this code is the use of the `NSApplicationLoad` function (in the line numbered 2). A Carbon application cannot access the Cocoa interface unless the application includes a call to `NSApplicationLoad`. This function is not needed for Cocoa applications, but it is mandatory for Carbon applications that use Cocoa. The `NSApplicationLoad` function is available starting with Mac OS X v10.2. `NSApplicationLoad` should be called after Carbon is initialized.

A detailed explanation of each numbered line of code in Listing 2 follows the listing.

Listing 2 Implementing the controller in the Cocoa source file

```
#import "Controller.h"

static Controller *sharedController;

@implementation Controller

+ (Controller *) sharedController
{
    return sharedController;
}

- (id) init                                                            // 1
{
    self = [super init];
    NSApplicationLoad();                                               // 2
    if (![NSBundle loadNibNamed:@"MyWindow" owner:self]) {
        NSLog(@"failed to load MyWindow nib");
    }
    sharedController = self;
    return self;
}

- (void) setCallBack:(CallBackType) callBack                           // 3
{
    _callBack = callBack;
}

- (void) showWindow                                                    // 4
{
    [window makeKeyAndOrderFront:nil];
}

- (void) changeText:(NSString *)text                                   // 5
{
    [textField setStringValue:text];
}

- (IBAction) buttonPressed:(id)sender                                  // 6
{
    (*_callBack) (kEventButtonPressed);
}

@end
```

Here's what the code does:

1. Defines the method that initializes Cocoa. You'll write a C-callable wrapper function for this method later. See "Writing C-Callable Wrapper Functions" (page 34).
2. Calls `NSApplicationLoad` as required. This entry point is needed for Carbon applications using Cocoa API but is a no-op for Cocoa applications.
3. Sets a callback for the controller. The callback (as you'll see later) is provided by the Carbon application to handle the button-press event.
4. Shows the Cocoa window and makes it active and frontmost. You'll write a C-callable wrapper function for this method later. See "Writing C-Callable Wrapper Functions" (page 34).
5. Displays a string in the text field of the Cocoa window. You'll write a C-callable wrapper function for this method later. See "Writing C-Callable Wrapper Functions" (page 34).
6. Defines the Interface Builder action associated with the button in the Cocoa window. It invokes the callback (provided by Carbon) that handles the button-press event. You'll connect the action method to its button target later. See "Creating the Cocoa Window in Interface Builder" (page 35).

Note: The Interface Builder action is an example of how an action method in Cocoa could control something in a Carbon application. The example shows that such control is possible; however, the task performed by the example code doesn't require Carbon. That is, Cocoa could update the text field itself.

Declaring the Controller Interface

The code to declare the interface for the controller (`Controller.h`) is shown in Listing 3. The item of note in this listing is the `_callBack` instance variable. The Carbon application provides the callback function that's assigned to this variable.

Listing 3 Declaring the interface for the controller

```
#import <Cocoa/Cocoa.h>
#import "CocoaStuff.h"

typedef OSStatus (*CallBackType)(int);

@interface Controller : NSObject
{
    id window;
    id textField;
    CallBackType _callBack;
}

- (void)setCallBack:(CallBackType)callBack;
- (void)showWindow;
+ (id)sharedController;
@end
```

Writing C-Callable Wrapper Functions

Listing 4 shows the C-callable wrapper functions that are part of the Cocoa source file. These functions are in the `Controller.m` file in the CocoaInCarbon project. Each function in Listing 4 wraps around one of the methods shown in Listing 2 (page 32).

One of the critical items in this code is the use of and NSAutoreleasePool object. For Cocoa API used by a Carbon application, you must set up autorelease pools as needed. A detailed explanation of each numbered line of code appears following the listing.

Listing 4 C-callable wrapper functions for the Cocoa API

```
OSStatus initializeCocoa (OSStatus (*callBack)(int))                    // 1
{
    Controller *controller;
    NSAutoreleasePool *localPool;

    localPool = [[NSAutoreleasePool alloc] init];                       // 2
    controller = [[Controller alloc] init];                             // 3
    [controller setCallBack:callBack];                                  // 4
    [localPool release];                                                // 5
    return noErr;
}

OSStatus orderWindowFront(void)                                         // 6
{
    NSAutoreleasePool *localPool;

    localPool = [[NSAutoreleasePool alloc] init];
    [[Controller sharedController] showWindow];
    [localPool release];
    return noErr;
}

OSStatus changeText (CFStringRef message)                               // 7
{
    NSAutoreleasePool *localPool;

    localPool = [[NSAutoreleasePool alloc] init];
    [[Controller sharedController] changeText:(NSString *)message];     // 8
    [localPool release];
    return noErr;
}
```

Here's what the code does:

1. Defines a C-callable wrapper function that must be called by the Carbon application to initialize Cocoa.

2. Allocates and initializes an autorelease pool, as required. You must do this for each C- callable wrapper function. See "Using Carbon and Cocoa in the Same Application" (page 15) for more information.

3. Calls the `init` method. Recall that this method makes the required call to `NSApplicationLoad`.

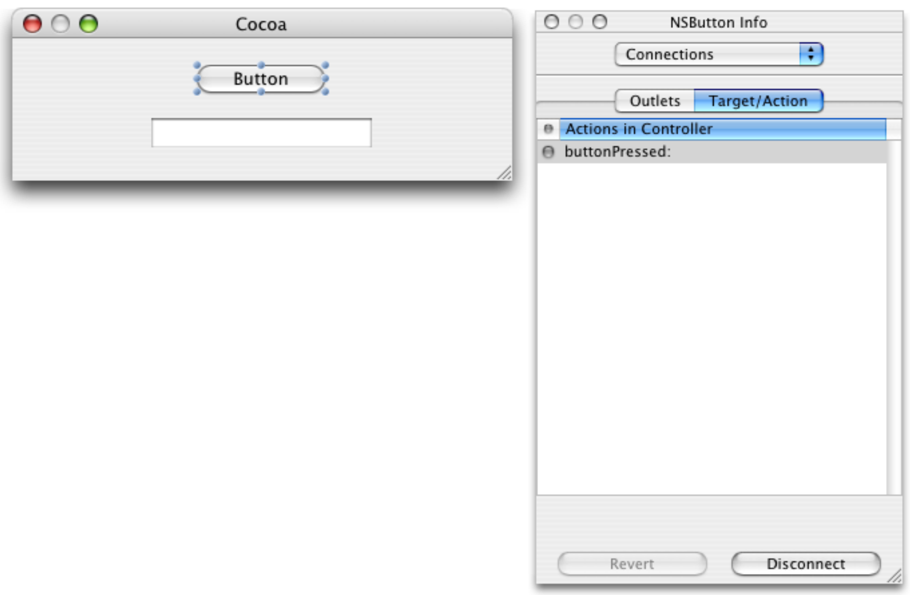
4. Assigns the callback function provided by the Carbon application to the controller's callback instance variable.

5. Releases the local autorelease pool.
6. Defines a C-callable wrapper function for the `showWindow` method.
7. Defines a C-callable wrapper function for the `changeText:` method.
8. Casts a `CFStringRef` value to NSString *. Recall from "Interchangeable Data Types" (page 13) that the `CFString` data type is toll-free bridged to the NSString class.

Creating the Cocoa Window in Interface Builder

You must use Interface Builder to add the Cocoa window to the appropriate nib file and connect targets to the appropriate actions. A nib file contains the definition of a set of user-interface elements. When the button in the sample application is pressed, it triggers the `buttonPressed` action method (see Figure 3). Recall from Listing 2 (page 32) that the `buttonPressed` method calls the callback function assigned to the controller. In the CocoaInCarbon application, the callback is provided by the Carbon application.

Figure 3 Connecting a button to a button-pressed action



For more information on creating a Cocoa window in Interface Builder, see *Cocoa Application Tutorial*.

Setting Up the Carbon Application to Use the Cocoa Interface

Your Carbon application must perform a number of tasks to use the interface provided by the Cocoa. These tasks are described in the following sections:

- "Including the Common Header File" (page 36)

- "Writing a Command Handler" (page 36)

Including the Common Header File

Both the Cocoa and the Carbon source files need to have a copy of the header file that declares any constants used to communicate events. Listing 1 (page 31) shows the header file (`CocoaStuff.h`) that needs to be included with both the Cocoa and the Carbon source files for the CocoaInCarbon project. See "Writing a Common Header File" (page 31) for details.

Writing a Command Handler

The Carbon code in the CocoaInCarbon application provides a function (`handleCommand`) that is passed as a parameter to the C-wrapper function that initializes Cocoa. When the user presses the button in the Cocoa window, the action method associated with the button invokes the `handleCommand` function. In the CocoaInCarbon application, this function just sends a string back to the Cocoa method, so it is not really necessary. However, the `handleCommand` function does illustrate how a more complex application could be notified of user actions in the Cocoa source.

The `handleCommand` function is shown in Listing 5. A detailed explanation of each numbered line of code appears following the listing.

Listing 5 A Carbon function that handles a Cocoa button press

```
static OSStatus handleCommand (int commandID)
{
    OSStatus osStatus = noErr;
    if (commandID == kEventButtonPressed)                    // 1
    {
        osStatus = changeText (CFSTR("button pressed!"));    // 2
        require_noerr (osStatus, CantCallFunction);
    }
CantCallFunction:
    return osStatus;
}
```

Here's what the code does:

1. Checks to make sure the event is a button-press event, the only event handled by the function.

2. Calls the `changeText` C-wrapper function with the string "button pressed!".

Using a Carbon User Interface in a Cocoa Application

This article describes how you can use a Carbon user interface in a Cocoa application. In Mac OS X version 10.2 and later, the system provides code that allows Cocoa and Carbon to communicate user events to each other. Communication between the two application environments is what enables a Cocoa application to use a Carbon user interface.

It is important to recognize that embedding a Carbon control inside a Cocoa window is not supported.

Using a Carbon interface in a Cocoa application requires you to perform the following major tasks:

- "Creating the Carbon User Interface" (page 38). You use Interface Builder to create a Carbon window and add controls and other items to the window.
- "Setting Up the Cocoa Application to Use the Carbon User Interface" (page 39). You load the nib file that specifies the Carbon window, create an NSWindow to allow management of the Carbon window with Cocoa methods, and show the window.

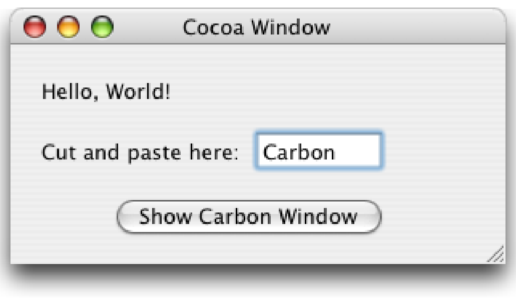
The tasks described in the following sections are illustrated using sample code taken from a working application, CarbonInCocoa. See "About the CarbonInCocoa Application" (page 37) for a description of the application. You can download the code for *CarbonInCocoa*.

Although most of the code from the CarbonInCocoa application is shown in the listings in this article, not all of the code is included. You should download the CarbonInCocoa Xcode project for a more complete picture of how the Cocoa and Carbon pieces fit together.

About the CarbonInCocoa Application

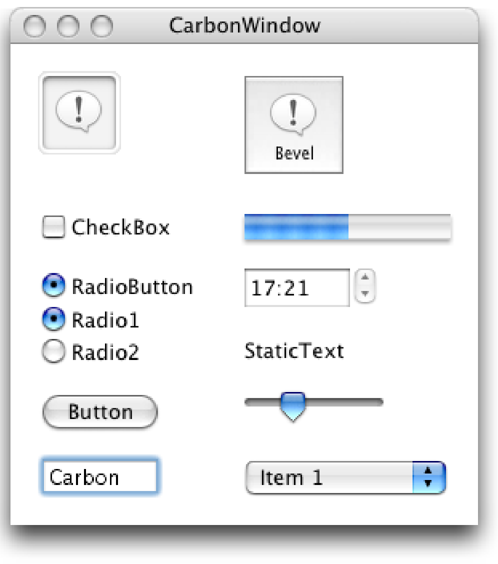
As you read the subsequent sections it may be helpful to have an idea of how the CarbonInCocoa application behaves and what the user interface looks like. When the application is launched, the window shown in Figure 1 appears. This window is a Cocoa window created with Interface Builder.

Figure 1 The Cocoa user interface for the CarbonInCocoa application



When the user clicks the Show Carbon Window button, the window shown in Figure 2 opens and becomes the frontmost and active window. The window in Figure 2 is a Carbon window created with Interface Builder. When the user clicks in one of the windows, that window becomes the active window. The user can type text in the text field in either window, copy the text from one window to the other, or cut the text from either window.

Figure 2 The Carbon user interface for the CarbonInCocoa application



The CarbonInCocoa application is simple. Its purpose is to show how little code you need to provide to use a Carbon user interface in a Cocoa application.

Creating the Carbon User Interface

You should use Interface Builder to create the Carbon user interface. Follow these steps to create a Carbon window:

1. Open Interface Builder.
2. Under Carbon in the Starting Point dialog, select Window and click New.
3. When the window appears, drag items form the Carbon palette to the window to create the interface.

   See *Interface Builder*, or Interface Builder help from within Interface Builder, for details on how to use Interface Builder.

   See *Apple Human Interface Guidelines* for information on making an Aqua-compliant interface.
4. Save the file.

   Interface Builder saves the interface in a nib file. (The “ib” in “nib” represents Interface Builder.) A nib file contains an archive of the interface. When you want to show the interface, you need to unarchive the nib file. You’ll see how to do this in “Loading the Nib File” (page 40).

Setting Up the Cocoa Application to Use the Carbon User Interface

You must perform the following tasks to enable your Cocoa application to use the Carbon user interface:

- "Adding the Nib File that Specifies the Carbon Interface" (page 39)
- "Declaring the Interface for the Controller" (page 39)
- "Loading the Nib File" (page 40)
- "Creating an NSWindow Object for the Carbon Window" (page 40)
- "Showing the Carbon Window" (page 41)

Adding the Nib File that Specifies the Carbon Interface

To add the nib file that specifies the Carbon interface, do the following:

1. Open the Xcode project for your Cocoa application.
2. Choose Project > Add Files.
3. Locate the nib file you just created and double-click its name.
4. Click Add in the dialog that appears.

   If your Cocoa application has more than one target, you need to choose the appropriate target before you click Add.

Declaring the Interface for the Controller

Your controller for the CarbonInCocoa application has two instance variables: a `WindowRef` structure for the Carbon window and an NSWindow object. The NSWindow object allows management of the Carbon window using Cocoa methods. (See "Showing the Carbon Window" (page 41).)

The sample application's controller has no other instance variables, but your application would need to declare any other instance variables that are appropriate. Listing 1 shows the declaration for the controller in the CarbonInCocoa application.

Listing 1 The declaration for the controller

```
@interface MyController : NSObject
{
    WindowRef   window;
    NSWindow    *cocoaFromCarbonWin;
}
@end
```

Loading the Nib File

The Cocoa nib file is loaded automatically when the Cocoa application is launched; however, you must explicitly load the nib file that contains the Carbon interface. For the CarbonInCocoa application, the nib file is loaded when the user clicks the Show Carbon Window button in the Cocoa window.

Listing 2 shows the code needed to load the Carbon nib file at runtime. An explanation of each numbered line of code appears following the listing.

Listing 2 Loading a nib file for a Carbon window

```
CFBundleRef bundleRef;
IBNibRef    nibRef;
OSStatus    err;

bundleRef = CFBundleGetMainBundle();                              // 1
err = CreateNibReferenceWithCFBundle (bundleRef,
                CFSTR("SampleWindow"),
                &nibRef);                                         // 2
if (err!=noErr)
            NSLog(@"failed to create carbon nib reference");

err = CreateWindowFromNib (nibRef,
                        CFSTR("CarbonWindow"),
                        &window);                                 // 3
if (err!=noErr)
            NSLog(@"failed to create carbon window from nib");

DisposeNibReference (nibRef);                                     // 4
```

Here's what the code does:

1. Calls the Core Foundation opaque type CFBundle function `CFBundleGetMainBundle` to obtain an instance of the application's main bundle. You need this reference for the next call.

2. Calls the Interface Builder Services function `CreateNibReferenceWithCFBundle` to create a reference to the Carbon window's nib file. The Core Foundation string you provide must be the name of the nib file, but without the `.nib` extension.

3. Calls the Interface Builder Services function `CreateWindowFromNib` to unarchive the Carbon window from the nib file. On return, `window` is a `WindowRef` to the Carbon window. Recall from "Declaring the Interface for the Controller" (page 39) that `window` is a controller instance variable.

4. Calls the Interface Builder Services function `DisposeNibReference` to dispose of the reference to the nib file. You should call this function immediately after you have finished unarchiving an object.

Creating an NSWindow Object for the Carbon Window

You do not need to create an NSWindow object for the Carbon window; however, doing so lets you use Cocoa methods rather than Carbon functions to manage the Carbon window, which may be desirable in some applications and is what the CarbonInCocoa application does.

You can create an NSWindow object for a Carbon window by allocating an NSWindow object and then initializing the object using the `initWithWindowRef:` method, as shown in the following line of code:

```
cocoaFromCarbonWin = [[NSWindow alloc] initWithWindowRef:window];
```

Recall from "Loading the Nib File" (page 40) that `window` is a reference to the Carbon window.

You can find more information about the NSWindow object and the `initWithWindowRef:` method in *NSWindow Class Reference*.

Showing the Carbon Window

Because you created an NSWindow object for the Carbon window, you can call the `makeKeyAndOrderFront:` method to show the window instead of calling the Carbon function `ShowWindow`. You show the Carbon window with the following line of code:

```
 [cocoaFromCarbonWin makeKeyAndOrderFront:nil];
```

HICocoaView: Using Cocoa Views in Carbon Windows

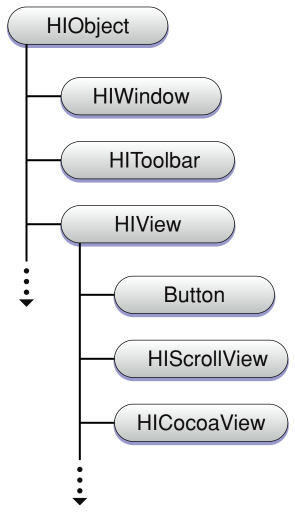
Cocoa provides views that are either not currently available in HIToolbox or are available without full support. These include views such as `WebView`, `PDFView`, `QTMovieView`, and `NSTokenField`. In addition, the Cocoa and Carbon control hierarchies are incompatible, so it has been difficult or impossible to have views from both frameworks embedded within the same window.

A new type of HIView called HICocoaView solves these problems. In Mac OS X v10.5 and later, you can embed a Cocoa view (any subclass of `NSView`) inside the HIView control hierarchy in a Carbon window. This is accomplished by associating the Cocoa view with a Carbon wrapper view called HICocoaView, a subclass of HIView. You can use standard HIView functions to manipulate the wrapper view, and you can use Cocoa methods to manipulate the associated Cocoa view.

Note: HICocoaView is supported only in Carbon windows with compositing enabled. For more information about compositing windows, see *HIView Programming Guide*.

Figure 1 shows how HICocoaView fits into the HIObject class hierarchy.

Figure 1 HICocoaView class hierarchy



Because HICocoaView is a subclass of HIView, you can use HIView functions to manipulate a wrapper view. For example, you can use `HIViewFindByID` to find the view in a window's view hierarchy. If the view needs to be made visible, you can call `HIViewSetVisible`. If it needs to be redrawn, you can call `HIViewSetNeedsDisplay`. If you need more control over your view, you can also intercept any of the Carbon control events and implement them yourself. Note that you don't need to handle the `kEventControlDraw` event; the wrapper view takes care of drawing its Cocoa view.

There are no restrictions on the type of Cocoa view you can wrap. A wrapped Cocoa view may also contain other Cocoa views. To gain access to the functionality of a wrapped Cocoa view, you can use any of its methods. To invoke these methods, you will need to use Objective-C to create and send messages to the Cocoa view. For example, if you want a `PDFView` object associated with a Carbon wrapper view to advance to the next page, you need to send the message `goToNextPage:` to the object.

When you modify the state of a Carbon wrapper view, adjustments are automatically made to the state of the associated Cocoa view. This is a one-way process, however—messages sent to the Cocoa view do not necessarily change the state of the wrapper view. For example, sending the `setFrame:` message to the Cocoa view does not reposition the wrapper view within its parent view.

When you embed a wrapper view inside a window, there are some limitations:

- The wrapper view must be embedded inside the content view of the window.
- The wrapper view cannot overlap other wrapper views.
- The wrapper view cannot contain any other Carbon views. (This restriction does not apply to the wrapped Cocoa view, which is allowed to contain other Cocoa views.)

The next section describes how to incorporate HICocoaView into your application.

Using HICocoaView

The HICocoaView API is easy to understand and use. There are three functions:

`HICocoaViewCreate`	Creates a Carbon view that serves as a wrapper for a Cocoa view.
`HICocoaViewSetView`	Associates a Cocoa view with a Carbon wrapper view.
`HICocoaViewGetView`	Returns the Cocoa view associated with a Carbon wrapper view.

This section explains when and how to use these functions.

Preparing your Carbon Project to use Cocoa

Before you can use the HICocoaView feature, you need to take the following steps to prepare your Carbon project to use Objective-C and Cocoa:

- Add the appropriate Cocoa frameworks to your project target. For example, if you're going to use a Cocoa web view, add `Cocoa.framework` and `WebKit.framework` to the list of frameworks to link against.
- Import the necessary Cocoa headers. For example, if you're going to use a Cocoa web view, you would add the following code to your source file:

```
#import <Cocoa/Cocoa.h>
#import <WebKit/WebKit.h>
```

- In functions where you're using Cocoa, allocate and initialize an `NSAutoreleasePool` object and release it when it is no longer needed. (If your application is running in Mac OS X v10.4 or later, you don't need to use autorelease pools in functions that are called, directly or indirectly, by the toolbox.)
- Prepare your Carbon application to use Cocoa by calling the `NSApplicationLoad` function. Typically, you do this in your main function before executing any other Cocoa code.
- Use the Objective-C or Objective-C++ compiler to build those parts of your project that use Cocoa. The article "Preprocessing Mixed-Language Code" (page 11) describes how to configure an Xcode project to use the appropriate compiler.

Creating a Wrapper View

To create a Carbon wrapper view and add it to the view hierarchy of a Carbon window, you use one of two approaches:

- Call the `HICocoaViewCreate` function to create the wrapper view at runtime, specifying the Cocoa view you want to wrap. Then embed the wrapper view inside the window's view hierarchy. For information about embedding and positioning views in a view hierarchy, see *HIView Programming Guide*.
- Use the Interface Builder application to design a nib-based Carbon window that contains a placeholder for the wrapper view. Interface Builder does not provide a way to associate a Cocoa view with the wrapper view, so you'll need to make this association at runtime. When you instantiate the window, the system creates an empty wrapper view for you and adds it to the window's view hierarchy.

Using HICocoaViewCreate

The following code example shows how to use `HICocoaViewCreate` to create a wrapped Cocoa view that can be embedded inside the content view of a Carbon window:

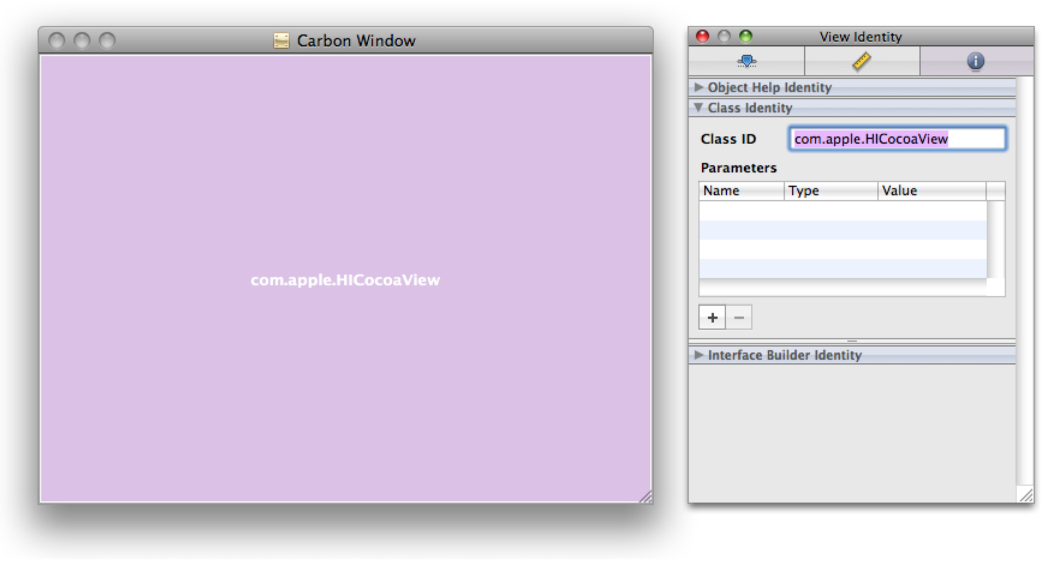
```
NSView *myCocoaView = [[SomeNSView alloc] init];
HIViewRef myHICocoaView;
HICocoaViewCreate (myCocoaView, 0, &myHICocoaView);
[myCocoaView release];
```

Using Interface Builder

If you're using Interface Builder, the first step is to add an HIView to your Carbon window by dragging an HIView object from the Carbon Objects palette into the window. If you like, resize the view to fill a larger area of the window. Now select the view and use the Inspector window to assign the class ID `"com.apple.HICocoaView"` to the view. You also need to assign a control signature and ID; you'll use these values to find the view at runtime.

Figure 2 shows how a nib-based Carbon window that contains a wrapper view might look in Interface Builder.

Figure 2 A Carbon wrapper view in Interface Builder



Class ID: com.apple.HICocoaView

Associating a Cocoa View with a Wrapper View

To associate a Cocoa view with an existing Carbon wrapper view, you use the `HICocoaViewSetView` function. There are two occasions for using this function:

- You have an empty wrapper view, and now you're ready to use it to wrap a Cocoa view. Typically, this happens when you use Interface Builder to create a wrapper view as a placeholder until a Cocoa view can be associated with it at runtime.
- You have a wrapper view with an associated Cocoa view, and you want to replace this Cocoa view with a new one.

The following code example shows how to find a wrapper view in a window's content view hierarchy and associate a Cocoa web view with the wrapper view:

```
const HIViewID kMyHICocoaViewID = { 'Test', 1 };
HIViewRef myHICocoaView = NULL;
HIViewFindByID (HIViewGetRoot(myWindow), kMyHICocoaViewID, &myHICocoaView);
if (myHICocoaView != NULL) {
   WebView *myWebView = [[WebView alloc] init];
   HICocoaViewSetView (myHICocoaView, myWebView);
   [myWebView release];
}
```

Getting the Cocoa View from a Wrapper View

If you have a Carbon wrapper view with an associated Cocoa view, you can use the `HICocoaViewGetView` function to get a pointer to the Cocoa view. Typically, you use this function when you want to send a message to the Cocoa view.

The following code example shows how to obtain a Cocoa web view from an existing wrapper view and load a webpage:

```
NSString *urlText = @"http://developer.apple.com/referencelibrary/";
WebView *myWebView = (WebView*) HICocoaViewGetView (myHICocoaView);
if (myWebView != NULL)
   [[myWebView mainFrame] loadRequest:[NSURLRequest requestWithURL:[NSURL
URLWithString:urlText]]];
```

Using a Nib-Based Cocoa User Interface

If you use a Cocoa nib file to specify a more complex user interface, your Carbon application needs to load the nib at runtime in order to embed the Cocoa user interface in an HICocoaView. One way to do this is to use a custom controller object to load the nib and access the UI. You can use the `NSViewController` class to simplify this task. `NSViewController` makes it easy to load a nib and get access to the `NSView`-based user interface inside. The approach described here is adapted from a working sample application called *HIView-NSView*. The sample uses a subclass of `NSViewController` called `WebViewController` to implement some features in the user interface.

Listing 1 shows how to implement a wrapper function that creates a nib-based Carbon window, creates a nib-based Cocoa view that contains the user interface for a simple web browser, and embeds the user interface in an HICocoaView. An explanation for each numbered line of code follows the listing.

Listing 1 Using a nib-based Cocoa user interface in a Carbon window

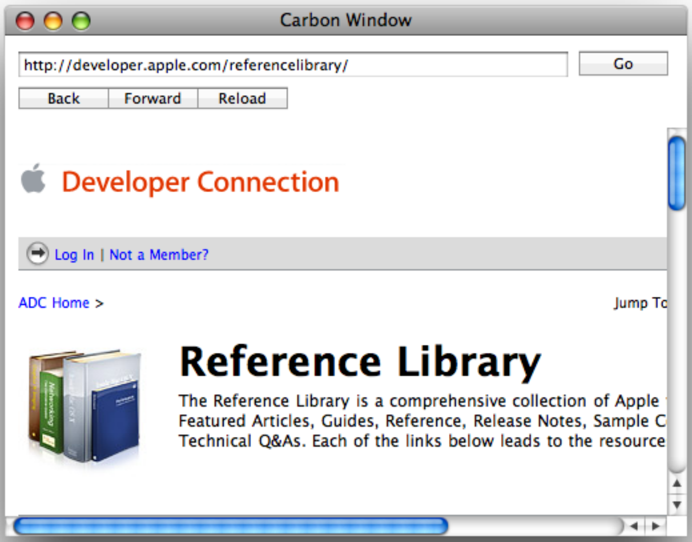
```
static OSStatus MyNewWindow (void)
{
    OSStatus status = noErr;

    NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];              // 1

    status = CreateWindowFromNib (gMainNibRef, CFSTR("MainWindow"), &gWindow); // 2

    require_noerr(status, CantCreateWindow);

    WebViewController* controller =
        [[WebViewController alloc] initWithNibName:@"WebView" bundle:nil];     // 3
    SetWRefCon(gWindow, (SRefCon)controller);                                  // 4

    HIViewRef carbonView;
    status = HIViewFindByID (HIViewGetRoot(window), kMyHICocoaViewID,          // 5
&carbonView);
    require_noerr(status, CantFindHICocoaView);

    NSView* cocoaView = [controller view];                                     // 6
    if (cocoaView != nil)
        status = HICocoaViewSetView (carbonView, cocoaView);                   // 7

    ShowWindow(gWindow);                                                       // 8

CantCreateWindow:
CantFindHICocoaView:
    [pool release];                                                    // 9
    return status;
}
```

Here's what the code does:

1. Creates a local autorelease pool. This step is necessary because this function is not being called by the toolbox.
2. Creates a nib-based Carbon window. In this example, global variables are used for both the main nib object and the new window object.
3. Creates a Cocoa view controller to gain access to the nib-based Cocoa view and to implement the Cocoa view's UI.
4. Stores the Cocoa view controller as window data. This information is used later to release the controller when the window is closed.
5. Finds the HICocoaView wrapper view in the Carbon window.
6. Retrieves the Cocoa view from the view controller.
7. Embeds the Cocoa view in the HICocoaView wrapper view.
8. Makes the Carbon window visible.
9. Drains and releases the local autorelease pool.

Figure 3 illustrates a simple Cocoa web browser view displayed inside a Carbon window.

Figure 3 Cocoa user interface inside a Carbon window



The Reference Library is a comprehensive collection of Apple
Featured Articles, Guides, Reference, Release Notes, Sample C
Technical Q&As. Each of the links below leads to the resource

To learn how to write an `NSViewController` subclass that implements the user interface in Figure 3, see the sample application *HIView-NSView*.

Using Cocoa in a Navigation Services Dialog

Beginning in Mac OS X v10.5, Navigation Services dialogs are implemented using custom subclasses of the Cocoa `NSOpenPanel` and `NSSavePanel` classes. This ensures that Carbon applications using Navigation Services receive the same user interface improvements as Cocoa applications. This change applies to sheet dialogs as well.

As a result of this change, Carbon applications can directly access features in these two Cocoa classes. A Navigation Services `NavDialogRef` object can be cast to a pointer to an instance of either `NSOpenPanel` or `NSSavePanel`, depending on the dialog type. By casting your `NavDialogRef` object to the appropriate Cocoa object, you can use any of the Objective-C accessor methods in the Cocoa object's class. This is similar to toll-free bridging (see "Interchangeable Data Types" (page 13)), but it only works in one direction.

There are some limitations to the bridging between Navigation Services and Cocoa. Dialogs created with Navigation Services must be invoked with `NavDialogRun` rather than with `NSOpenPanel` or `NSSavePanel` methods. As implied above, dialogs created with `NSOpenPanel` or `NSSavePanel` constructors cannot be cast to an `NavDialogRef` object. The Carbon and Cocoa models have some overlapping functionality such as file filtering, setting an accessory view, and setting a delegate object. In these cases, to avoid conflicts you should not mix the two models.

Here's an example that uses Navigation Services to create an Open File dialog and casts it to an `NSOpenPanel` object to set the title and message text displayed in the dialog:

```
// navOptions structure has been created and initialized for text documents
NavDialogRef theDialog = NULL;
NavCreateChooseFileDialog(&navOptions, NULL, NULL, NULL, Handle_NavFilter, NULL,
 &theDialog);
[(NSOpenPanel*)theDialog setTitle:@"Open Document"];
[(NSOpenPanel*)theDialog setMessage:@"Choose a text document to open:"];
NavDialogRun(theDialog);
```

Document Revision History

This table describes the changes to *Carbon-Cocoa Integration Guide*.

Date	Notes
2007-10-31	Updated for Mac OS X v 10.5. Made minor editorial and technical corrections.
2007-05-15	Added new articles about using Cocoa views in a Carbon window and using Cocoa in Navigation Services dialogs.
	Removed the article "Loading Cocoa Lazily Into Your Carbon Application."
2006-05-23	Corrected code listing to follow the Core Foundation memory-management rules.
	Corrected Listing 2 (page 40) in "Using a Carbon User Interface in a Cocoa Application" (page 37).
2005-08-11	Updated to describe proper use of CFRelease and availability of toll-free bridging between Carbon and Cocoa in Mac OS X. Made small corrections.
	Updated "Interchangeable Data Types" (page 13) to describe the availability of toll-free bridging in Mac OS X.
	Updated "Using a Cocoa User Interface in a Carbon Application" (page 29) to clarify that nib files are edited with Interface Builder.
	Updated introduction to include cross-reference to *Memory Management Programming Guide for Cocoa*.
	Changed `.ccx` to `.cxx` in "Preprocessing Mixed-Language Code" (page 11).
2005-04-08	Changed title from "Integrating Carbon and Cocoa in Your Application." Corrected errors in code examples.
	Corrected Listing 2 (page 32) by adding code that loads the MyWindow nib file.
	Corrected syntax error in `FSPathMakeRef` call in Listing 2 (page 24).
2004-06-28	Updated to reflect that loading Cocoa from a bundle is not required. Document reorganized into articles, finalized, and indexed.
	Added "Loading Cocoa Lazily Into Your Carbon Application" which describes loading Cocoa from a bundle. Other examples throughout the document have been simplified and no longer use bundles.
2003-01-03	Where possible, links to documentation are now to the local version of the document. Corrected some capitalization errors in Cocoa class names.

Date	Notes
2002-12-11	Updated formatting.
2002-05-01	Preliminary version of this document.

Index