Kernel Extension Programming Topics

Darwin > Kernel

2007-10-31

Contents

Figures, Tables, and Listings

Introduction to Kernel Extension Concepts

Mac OS X provides a kernel extension mechanism as a means of allowing dynamic loading of pieces of code into the kernel, without the need to recompile. These pieces of code are known generically as *plug-ins* or, in the Darwin kernel, as *kernel extensions* or *KEXTs*.

Who Should Read This Document?

Kernel Extension Concepts assumes you have no prior experience developing kernel extensions for Mac OS X or using Xcode, Apple's integrated development environment. The information in this series of topics is very basic and is intended to introduce you to the fundamental techniques you need to develop, debug, and package kernel extensions.

Organization of This Document

This document contains the following information:

- "About Kernel Extensions" (page 11) provides some basic information about kernel extensions, including how to decide if you need to write one.
- "Hello Kernel: Creating a Kernel Extension With Xcode" (page 15) is a tutorial that shows you how to write, test, load, and unload a simple kernel extension.
- "Hello I/O Kit: Creating a Device Driver With Xcode" (page 31) is a tutorial that shows you how to write, test, load, and unload a simple I/O Kit driver.
- "Hello Debugger: Debugging a Device Driver With GDB" (page 53) is a tutorial that shows you how to debug a kernel extension using GDB (a command-line debugger).
- "Packaging Your KEXT for Distribution and Installation" (page 71) describes how to use the Package Maker application to package your KEXT.
- "Loading Kernel Extensions at Boot Time" (page 87) describes how kernel extensions are loaded in Mac OS X.
- "Kernel Extension Dependencies" (page 91) describes how KEXTs declare dependencies and lists version numbers of kernel subcomponents and loadable extensions.

- "Kernel Extension Ownership and Permissions" (page 103) describes the ownership and permissions rules that apply to all KEXTs.

About Kernel Extensions

This article provides some general information about kernel extensions (KEXTs).

Why to Avoid KEXTs

Because KEXTs provide both modularity and dynamic loadability, they are a natural choice for any relatively self-contained service that requires access to kernel internal interfaces. Many of the components of the kernel environment support this extension mechanism, although in different ways.

For example, some networking features involve the use of network kernel extensions (*NKEs*). The ability to dynamically add a new file-system implementation is based on VFS KEXTs. Device drivers and device families in the I/O Kit are implemented using KEXTs. KEXTs make development much easier for developers writing drivers or those writing code to support a new volume format or networking protocol.

Because KEXTs run in supervisor mode in the kernel's address space, they are also harder to write and debug than user-level modules, and must conform to strict guidelines. Further, kernel resources are wired (permanently resident in memory) and are thus more costly to use than resources in a user-space task of equivalent functionality.

In addition, although memory protection keeps applications from crashing the system, no such safeguards are in place inside the kernel. A badly behaved kernel extension in Mac OS X can actually cause more trouble than a badly behaved application or extension could in previous version of the Mac OS.

Bugs in KEXTs can have far more severe consequences than bugs in user-level code. For example, a memory access error in a user application can, at worst, cause that application to crash. In contrast, a memory access error in a KEXT causes a system *panic*, crashing the operating system.

Finally, for security reasons, some customers restrict or don't permit the use of third-party KEXTs. As a result, use of KEXTs is strongly discouraged in situations where user-level solutions are feasible. The Darwin kernel guarantees that user threads are just as efficient as kernel threads, so efficiency should not be an issue. Unless your application requires low-level access to kernel interfaces or the data stream, you should use a higher level of abstraction when developing code for Mac OS X.

When you are trying to determine if a piece of code should be a KEXT, the default answer is generally *no*.

When You Need a KEXT

In particular, if your code was a system extension in previous versions of the Mac OS, such as Mac OS 9, that does not imply that it must necessarily be a kernel extension in Mac OS X. There are only a few good reasons for a developer to write a kernel extension:

- Your code needs to take a primary interrupt, that is, something in the hardware needs to interrupt the CPU.
- The primary client of your code is inside the kernel, for example, a block device whose primary client is a file system.
- A sufficiently large number of running applications require a resource that your code provides; for example, you have written a file-system stack.
- Your code needs to multiplex between multiple client applications that require high speed, excellent synchronization, or low latency.

If your code does not meet any of the above criteria, you should consider developing it as a library or a user-level daemon, or using one of the user-level plug-in architectures (such as QuickTime components or the Core Graphics framework) instead of writing a kernel extension.

If you are writing device drivers or code to support a new volume format or networking protocol, however, KEXTs may be the only feasible solution.

KEXT Tutorials

Fortunately, while KEXTs may be more difficult to write than user-space code, several tools and procedures are available to enhance the development and debugging process. The tutorials collected in this topic are designed to give you some hands-on experience with creating, building, and debugging KEXTs.

1. "Hello Kernel: Creating a Kernel Extension With Xcode" (page 15)
2. "Hello I/O Kit: Creating a Device Driver With Xcode" (page 31)
3. "Hello Debugger: Debugging a Device Driver With GDB" (page 53)
4. "Packaging Your KEXT for Distribution and Installation" (page 71)

Be sure to complete "Hello Kernel" first. This tutorial describes how to create and test a kernel extension (KEXT) for Mac OS X. In this tutorial, you'll create a very simple extension that prints text messages when loading and unloading.

The second tutorial, "Hello IOKit", describes how to write an I/O Kit device driver for Mac OS X. The driver you'll create is a simple driver that prints text messages but doesn't actually control any hardware.

The third tutorial, “Hello Debugger”, builds upon what you have learned in the first two. Using the sample driver you created in “Hello IOKit”, this tutorial describes how to prepare to debug a device driver for Mac OS X. You will learn how to set up a two-machine debugging environment and how to start using GDB, a command-line debugger, to perform remote debugging.

Although “Hello Debugger” is written with a device driver as the example, the steps for debugging are similar for debugging any type of kernel extension (KEXT). If you wish, you can substitute your own code for the HelloIOKit example. Note, however, that you may encounter a few inconsistencies. For example, examples of GDB commands may be dependent on the underlying source code language—I/O Kit extensions (drivers) use C++; the GDB commands for C may differ.

In the packaging tutorial, you’ll learn how to package a kernel extension for distribution and installation on Mac OS X. The KEXT can be any type: a device driver, file system stack, or Network Kernel Extension (NKE). This tutorial may be completed out of sequence; that is, you may decide to work through this tutorial before completing “Hello IOKit” or “Hello Debugger”.

Hello Kernel: Creating a Kernel Extension With Xcode

This tutorial describes how to create and test a kernel extension (KEXT) for Mac OS X. In this tutorial, you'll create a very simple kernel extension that prints text messages when loading and unloading. The tutorial assumes that you are working in a Mac OS X development environment.

Anatomy of a KEXT

To better understand what you're doing, it helps to know what's inside a kernel extension. In Mac OS X, all kernel extensions are implemented as *bundles* — folders that the Finder treats as single entities. Kernel extensions have names ending in `.kext`. In addition, all kernel extensions contain the following:

- A *property list* (plist) —a text file (in XML, Extensible Markup Language, format) that describes the KEXT's contents and requirements. This is a required file. A KEXT need contain nothing more than a property list file.
- A *KEXT binary* —Generally, a KEXT has one binary file, but it can have none (locally). However, if it has none, its property list must reference another KEXT and change its default settings. In this way, one KEXT can override or change the behavior of another KEXT. A KEXT binary is also sometimes called a kernel module.
- Optional *Resources* —Resources are useful if your KEXT needs to display an icon.

Roadmap

In this tutorial, you'll create, build, load, and run a simple kernel extension. Here are the steps you will follow:

1. "Create a new Project using Xcode" (page 16)
2. "Build the Kernel Extension" (page 21)
3. "Test the Kernel Extension" (page 24)

You'll use the Xcode application to create and build your KEXT. You'll use the Terminal application to type the commands to load and test your KEXT and view the results.

If you have never used Xcode before, you may also wish to read *Xcode Quick Tour Guide*.

Create a new Project using Xcode

A project is a document that contains all your files and targets. Targets are the things you can build from your project's files. A simple project has just one target that builds an application. A complex project may contain several targets.

The parts of your project can be found later on your disk. These include the source files as well as the targets (your KEXT). Xcode does not store these files in any special format, so you can view or edit the source files with another editing program if you wish. For now, we recommend using Xcode.

Here's how you'll create the kernel extension project:

1. "Create a Kernel Extension Project" (page 16)
2. "Implement the Needed Methods" (page 17)
3. "Edit the KEXT's Settings" (page 18)

The examples below assume that you will be logging in as an administrator of your machine. The account name in the examples is `admin`. If you use a different login account, be sure to substitute accordingly. Some of the commands you will be using require `root` privileges so you will be using the **sudo** command. The **sudo** command allows permitted users (such as an `admin` user) to execute a given command with `root` privileges. If you use a different login account, make sure it has administrative access.

Create a Kernel Extension Project

Kernel extensions are created and edited in Xcode, Apple's Integrated Development Environment (IDE).

From a Desktop Finder window, locate and launch the Xcode application, found at `/Developer/Applications/Xcode`. If this is the first time you've run Xcode, you'll see the new user Assistant. The Assistant asks you to make some decisions about your environment. For now, choose the defaults.

When you have finished with the Assistant, choose New Project from the File menu. In the New Project Assistant, scroll down to the Kernel Extension section and choose Generic Kernel Extension. Click Next.

For the Project Name, enter "HelloKernel". The default location is your home directory; however, if you create many projects, you should make a directory to hold them. Edit the Location to specify a "Projects" subdirectory, for example:

```
/Users/admin/Projects
```

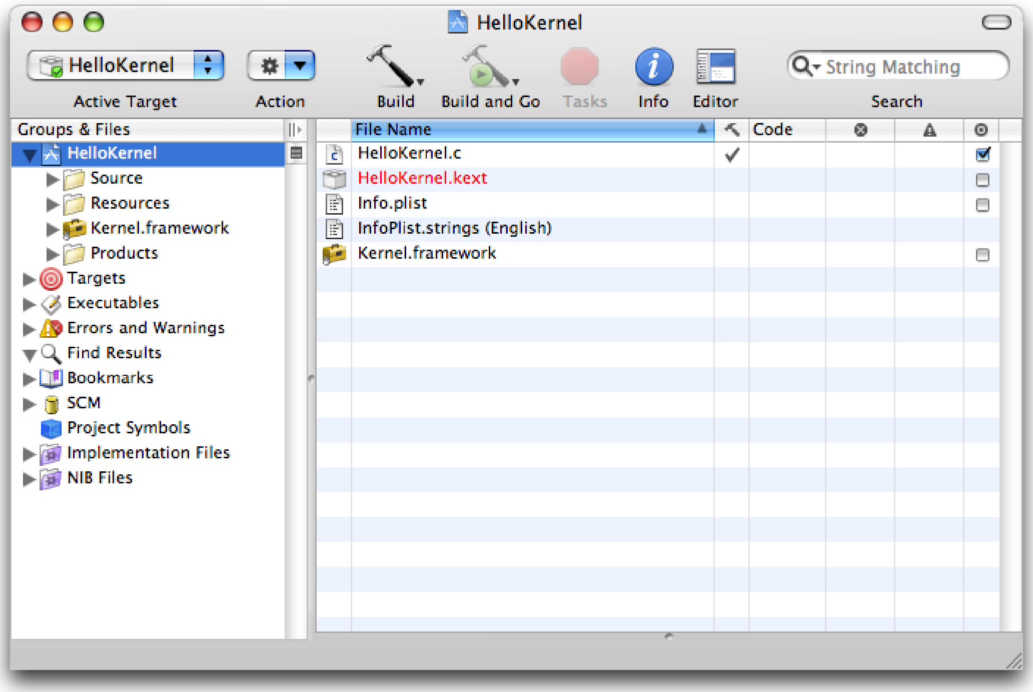
When you click Finish, Xcode creates the new project and displays its project window. The new project contains several files already, including a default source file, `HelloKernel.c`.

Implement the Needed Methods

If necessary, select HelloKernel in the Groups & Files pane. Double-click the `HelloKernel.c` in the detail view to display the source code in a separate editor window. Figure 1 (page 17) shows where you will find the `HelloKernel.c` file in the project window.

The default source file does nothing but return a successful status; you'll need to add some additional code. In particular, you will need to implement the initialization and termination code for your KEXT. The default template merely contains suggestions for these routines, as a place for you to begin.

Figure 1 Viewing source code in Xcode



Change the contents of `HelloKernel.c` to match the code in Listing 1.

Listing 1 HelloKernel.c

```
#include <sys/systm.h>
#include <mach/mach_types.h>

kern_return_t HelloKernel_start (kmod_info_t * ki, void * d)
{
 printf("KEXT has loaded!\n");
 return KERN_SUCCESS;
}

kern_return_t HelloKernel_stop (kmod_info_t * ki, void * d)
{
 printf("KEXT will be unloaded\n");
 return KERN_SUCCESS;
}
```

Save your changes by choosing File > Save. Close the separate editor window by clicking the red close button in the upper left.

Notice that `HelloKernel.c` includes two header files, `sys/systm.h` and `mach/mach_types.h`. Both header files reside in `Kernel.framework`. When you develop your own KEXT, be sure only to include header files from `Kernel.framework` (in addition to any header files you create) because only these files have meaning in the kernel environment. If you include headers from outside `Kernel.framework`, your KEXT might compile, but the functions and services those headers define will not be available in the kernel.

Edit the KEXT's Settings

Your kernel extension contains a property list, or *plist*, that tells the operating system what your KEXT contains and what it needs. If viewed from a text editor, the property list would be in XML (Extended Markup Language) format. However, you will be viewing and editing the plist information from within Xcode.

By default, Xcode allows you to edit your KEXT's property list as plain XML text in the Xcode editor window. However, it's easier to view and edit the property list file with the Property List Editor application. You can tell Xcode to open property list files using Property List Editor by following these steps:

1. Choose Xcode > Preferences and click the File Types icon. The File Types pane lists all the folder and file types that Xcode handles and the preferred editor for each type.
2. Click the disclosure triangle for file. Then click the disclosure triangle for text.
3. Select text.xml and click Default (Plain Text File) in the Preferred Editor column.
4. Choose External Editor and select Other from the menu that appears.
5. Select Property List Editor in `/Developer/Applications/Utilities/Property List Editor` and click OK.
6. Now Xcode lists External Editor (Currently Property List Editor) in the Preferred Editor column for text.xml. Click OK to close the Xcode Preferences window.

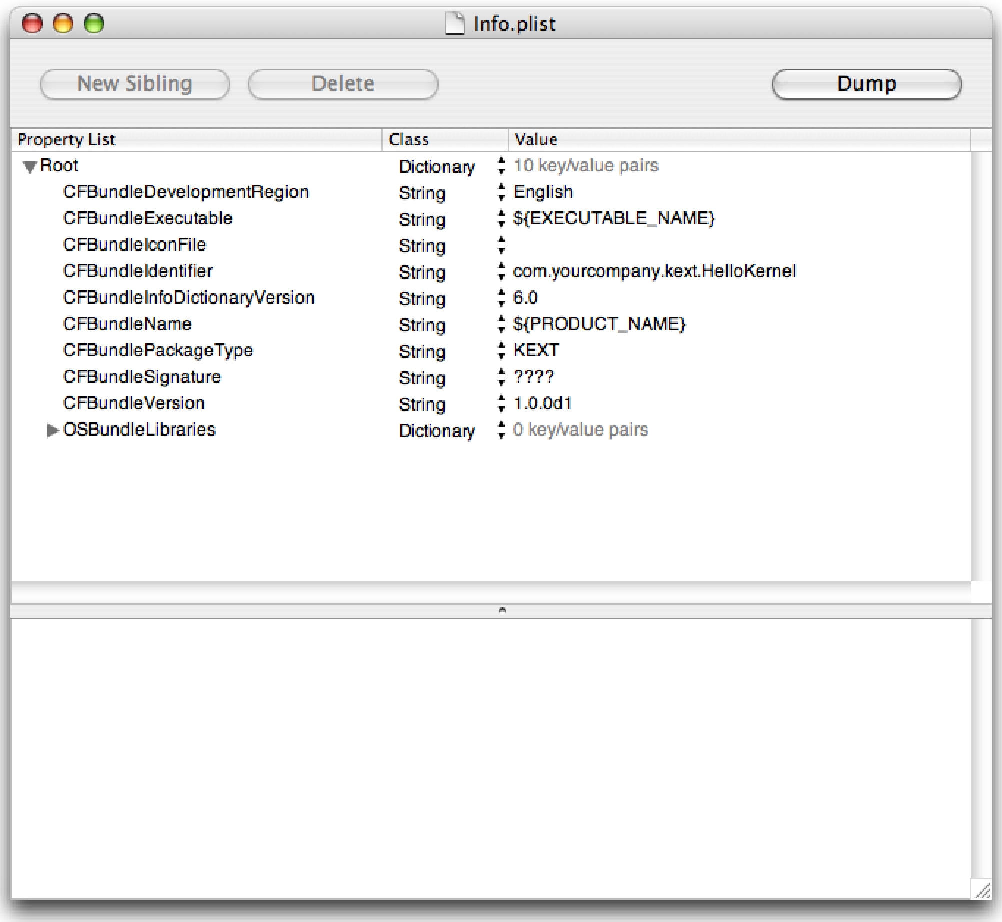
Now you can edit your KEXT's `Info.plist` file using Property List Editor:

1. Select HelloKernel in the Groups & Files view. Double-click Info.plist in the Xcode project window. Xcode starts Property List Editor, which displays the `Info.plist` file. (If Property List Editor displays a second editing window named Untitled, dismiss it by clicking the red close button in the upper left.)

If this doesn't work due to a misconfigured project, you can control-click the name of the file and choose "Open with Finder" from the resulting contextual menu. This should have the same effect.

2. In Property List Editor, click the disclosure triangle next to Root. You should see the elements of the property list file, as shown in Figure 2 (page 19).

Figure 2 HelloKernel property list



Property List	Class	Value
▼Root	Dictionary	10 key/value pairs
CFBundleDevelopmentRegion	String	English
CFBundleExecutable	String	${EXECUTABLE_NAME}
CFBundleIconFile	String	
CFBundleIdentifier	String	com.yourcompany.kext.HelloKernel
CFBundleInfoDictionaryVersion	String	6.0
CFBundleName	String	${PRODUCT_NAME}
CFBundlePackageType	String	KEXT
CFBundleSignature	String	????
CFBundleVersion	String	1.0.0d1
▶OSBundleLibraries	Dictionary	0 key/value pairs

3. Change the value of the `CFBundleIdentifier` property from the default chosen by Xcode.

   Apple has adopted a "reverse-DNS" naming convention to avoid namespace collisions. This is important because all kernel extensions share a single "flat" namespace.

   By default, Xcode names a new KEXT `com.yourcompany.kext.`<*projname*>, where <*projname*> is the name you chose for your project. You should replace the first two parts of this name with your actual reverse-DNS prefix, (for example: `com.apple`, `edu.ucsd`, and so forth). For this tutorial, you will use the prefix `com.MyTutorial`.

   On the line for `CFBundleIdentifier`, double-click on `com.yourcompany.kext.HelloKernel` in the Value field. Double-click on `yourcompany` and change this string to `MyTutorial`.

4. The `CFBundleVersion` property contains the version number of your kernel extension in the `'vers'` resource style (for more information on this, see http://developer.apple.com/documentation/mac/Toolbox/Toolbox-454.html and Technical Note TN1132).

   By default, Xcode assigns new kernel extensions the version number `1.0.0d1`. You can change this value if you wish, but your KEXT must declare *some* version in the `'vers'` resource style to be loaded successfully. To change the version value of your KEXT, click on the line for `CFBundleVersion` and double-click on `1.0.0d1` in the Value field.

   For this tutorial, leave the `CFBundleVersion` property with the default value `1.0.0d1`.

5. Mac OS X requires a KEXT that depends on other loadable extensions or on in-kernel components to declare its dependencies in its plist. This information must be contained in the `OSBundleLibraries` dictionary at the top level of the plist.

   You need to determine which loadable extensions or in-kernel components your KEXT depends on. You should examine the `#include` directives in your kernel extension's code and find the corresponding KEXTs in "Kernel Extension Dependencies" (page 91). In this tutorial, "HelloKernel" includes `sys/systm.h` and `mach/mach_types.h` so you will create an entry for `com.apple.kernel.bsd` and an entry for `com.apple.kernel.mach`.

   Click on the line for `OSBundleLibraries` and click on the disclosure triangle at the beginning of the line. The button previously labeled New Sibling should change to New Child. It not, check to be sure the disclosure triangle next to `OSBundleLibraries` is pointing down.

   Click the button that now says New Child (as soon as you do, it will change back to New Sibling). A new item will appear below `OSBundleLibraries`. Change the name from `New item` to `com.apple.kernel.bsd`. Double-click on the value field and enter `6.9.9`.

   Click the New Sibling button and change the name from `New item` to `com.apple.kernel.mach`. Double-click on the value field and enter `6.9.9`.

6. Save your changes by choosing File > Save from the Property List Editor menu. You won't be making any more changes to your KEXT's `Info.plist` file in this tutorial, so you can quit the Property List Editor application and return to Xcode by choosing Property List Editor > Quit Property List Editor.

Xcode also allows you to modify or create build settings for your KEXT. These settings define information that will be compiled into the kernel extension's executable.

1. On the left, in the Groups & Files view, click the disclosure triangle next to Targets and select HelloKernel (you don't have to click the disclosure triangle next to HelloKernel).

2. Click the Get Info button in the tool bar (it's the round blue button with an "i" in the middle).

3. Select the Build view and scroll down to the bottom of the Customized Settings list. Change the name of the module from the default chosen by Xcode (the value of the `MODULE_NAME` setting).

   Click on the line for `MODULE_NAME` and double-click on `com.yourcompany.kext.HelloKernel` in the Value field. Double-click on `yourcompany` and change this string to `MyTutorial`. Be sure that the `MODULE_NAME` value matches the value of the `CFBundleIdentifier` you entered in your kernel extension's `Info.plist` file or your KEXT will not run.

4. The `MODULE_START` and `MODULE_STOP` variables contain the names of your KEXT's initialization and termination routines. Be sure that these variables have the values used in your `HelloKernel.c` file. That is, for this tutorial, be sure these variables have the values `HelloKernel_start` and `HelloKernel_stop`, respectively. If you changed the names of the initialization and termination routines when entering the code in the previous section, make sure that the values for these variables match the names you used! Otherwise, your KEXT will not run.

5. The value of the `MODULE_VERSION` variable is in the `'vers'` resource style and must be numerically identical to the `CFBundleVersion` value in the kernel extension's `Info.plist` file or your KEXT may not load. For this tutorial, make sure the `MODULE_VERSION` variable has the value `1.0.0d1`.

Build the Kernel Extension

Here's how you'll build the kernel extension:

1. "Build the Project" (page 21)
2. "Fix Any Errors" (page 22)
3. "If There Were No Errors" (page 23)
4. "Build the Project Again" (page 24)

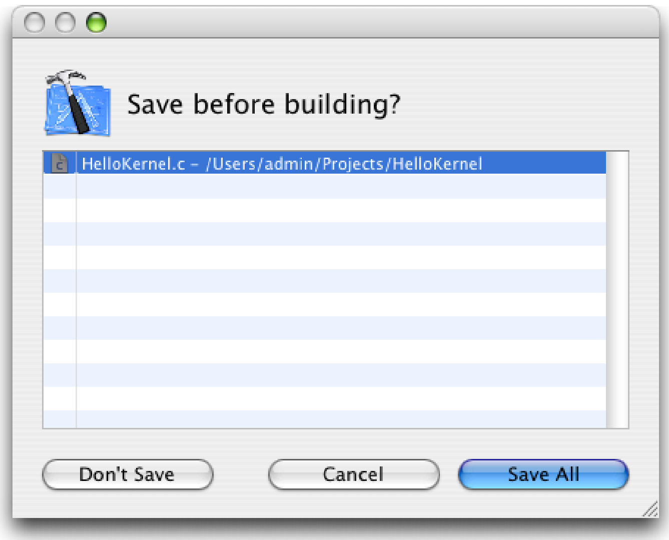
Build the Project

Click the Build button in the upper left corner of the editor window or select Build from the Build menu. The Build button looks like a hammer and is illustrated in Figure 3.

Figure 3 The build button looks like a hammer

If Xcode asks you whether to save some modified files, select all the files and click "Save All". Figure 4 shows the Save dialog.

Figure 4 Save before building

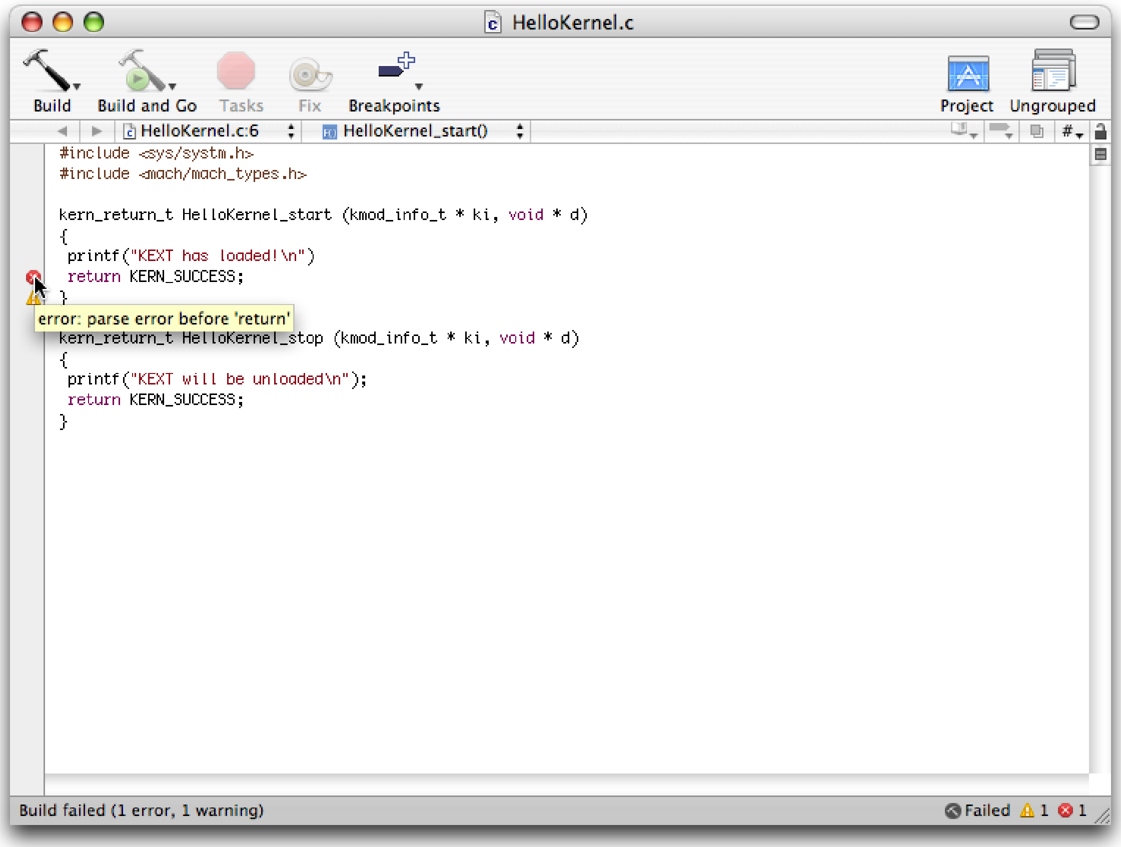


Xcode starts building your project. It stops if it reaches an error in the code.

Fix Any Errors

If you made any errors typing in the code, Xcode will stop and show you the errors in both the editor window (if you haven't closed it) and the main project window. In the project window, click the disclosure triangle next to Errors and Warnings in the Groups & Files view to reveal the files that contain errors. If you select a file listed under Errors and Warnings, Xcode lists the errors in that file, each accompanied by a short description of the problem. Double-click the file name to open an editor window and edit the source code. The source code with the error will appear in the editing window, as shown in Figure 5.

Figure 5 Xcode shows code errors



```
#include <sys/systm.h>
#include <mach/mach_types.h>

kern_return_t HelloKernel_start (kmod_info_t * ki, void * d)
{
 printf("KEXT has loaded!\n")
 return KERN_SUCCESS;
}
kern_return_t HelloKernel_stop (kmod_info_t * ki, void * d)
{
 printf("KEXT will be unloaded\n");
 return KERN_SUCCESS;
}
```

```
error: parse error before 'return'
```

```
Build failed (1 error, 1 warning)
Failed 1 1
```

You can edit the source code inside the editing window. Correct the error, save your changes, and build the project again.

If There Were No Errors

If you didn't have any errors, you can insert one so that you can try out the error-handling facility. Try removing the semicolon at the end of one of the lines of code, as shown in Listing 2.

Listing 2 HelloKernel.c with error

```
#include <sys/systm.h>
#include <mach/mach_types.h>

kern_return_t HelloKernel_start (kmod_info_t * ki, void * d)
{
 printf("KEXT has loaded!\n") // <--ERROR! Missing semicolon!
 return KERN_SUCCESS;
}

kern_return_t HelloKernel_stop (kmod_info_t * ki, void * d)
{
    printf("KEXT will be unloaded\n");
    return KERN_SUCCESS;
}
```

Build the Project Again

Click the Build button. If Xcode asks you whether to save some modified files, select all the files and click “Save All”.

Xcode starts building your project again. If there is an error, fix the error as described above in “Fix Any Errors” (page 22). Otherwise, if the build succeeds, you can move on to loading the extension.

Test the Kernel Extension

This section shows how to test the kernel extension. You’ll load your KEXT with the **kextload** command, you’ll use the **kextstat** command to see that it’s loaded, and finally, you’ll unload your KEXT from the kernel with the **kextunload** command.

You’ll use the Terminal application to type the commands to load and unload your KEXT. You’ll view the results as they are written to the system log file, `/var/log/system.log`.

Note: This tutorial uses the “`%`” prompt when it shows the commands you type in the Terminal application. This is the default prompt of the `tcsh` shell. If you’re using a different shell, you may see a different prompt. For example, the prompt in the `bash` shell, which is the default shell in Mac OS X version 10.3, is “`$`”.

Here’s how you’ll test your KEXT:

1. “Getting Root Privileges” (page 24)
2. “Start the Terminal Application” (page 25)
3. “Load the KEXT Binary” (page 26)

Getting Root Privileges

To use the **kextload** and **kextunload** commands you must have `root` or super user privileges. Instead of logging in as `root`, for this tutorial you will use the **sudo** command which gives permitted users the ability to execute a given command as `root`.

By default, Mac OS X allows `admin` and `root` users to use the **sudo** command, so make sure you are currently logged in to an `admin` account before using **sudo**. An admin user is a user who has “Allow user to administer this computer” checked in the Security view of the Accounts system preference.

Start the Terminal Application

1. Start the Terminal application. From a Desktop Finder window, locate and launch the Terminal application, found at `/Applications/Utilities/Terminal`.

2. To view the system log file, enter the following command at the Terminal prompt:

   ```
   % tail -f /var/log/system.log
   ```

3. Open a second window in the Terminal application. Choose File > New Shell from the Terminal menu. Position this window so that you can view both windows easily. You will load your KEXT from the second window.

4. In the second Terminal window, move to the directory that contains your KEXT. Xcode stores your KEXT in the `Debug` folder of the `build` directory of your project location (unless you've set a different location for build products using Xcode's Preferences dialog).

   Use the **cd** command to move to the appropriate directory. For example:

   ```
   % cd Projects/HelloKernel/build/Debug
   ```

   This directory contains your KEXT. You can use the **ls** command to view the contents of this directory. For example:

   ```
   % ls
   HelloKernel.kext
   ```

   Your KEXT should have the name `HelloKernel.kext`. Note that this name is formed from the project name and a suffix, `.kext`.

   Important: For purposes of packaging, distribution, and installation, the filename of the KEXT (apart from the suffix) does not matter. However, the name of the KEXT binary (stored in the KEXT's property list) should be unique, using the recommended "reverse-DNS" naming convention.

   From a Desktop Finder window, a KEXT appears as a single file (look for it from the Desktop if you like). From the Terminal application, however, a KEXT appears as a directory. The KEXT directory contains the contents of your KEXT, including the plist (`Contents/Info.plist`) and the KEXT binary (`Contents/MacOS/HelloKernel`).

5. View the contents of the KEXT directory. Use the **find** command.

   For example:

   ```
   % find HelloKernel.kext
   HelloKernel.kext
   HelloKernel.kext/Contents
   HelloKernel.kext/Contents/Info.plist
   HelloKernel.kext/Contents/MacOS
   HelloKernel.kext/Contents/MacOS/HelloKernel
   HelloKernel.kext/Contents/Resources
   HelloKernel.kext/Contents/Resources/English.lproj
   HelloKernel.kext/Contents/Resources/English.lproj/InfoPlist.strings
   ```

You are now ready to load and run (and then unload) your KEXT.

Load the KEXT Binary

Because kernel extensions contain code and data that are loaded into the kernel, the most protected environment in the operating system, their file ownership and permissions must be set to prevent unauthorized tampering. All KEXT bundles (all files and folders in the KEXT, including the KEXT binary) must be owned by the user `root` and the group `wheel`. In addition, the folders and files of the KEXT bundle must have their permissions set so that they are not writable by any user other than the super user.

For development purposes, however, you can make a `root`-owned copy of your KEXT binary to load and test. You should not change the ownership and permissions of any of your KEXT files in your own directory, because you will no longer be able to save them after working on them. For this tutorial, you will use the **sudo** command to copy the KEXT binary (`HelloKernel.kext`) to the `/tmp` directory and load and unload the KEXT from there. Using **sudo** to copy the KEXT binary gives the KEXT the super user's ownership and permissions and leaves the original KEXT alone so you can revise and save it as you choose.

Note: Every time you make changes to your KEXT and rebuild it, you need to repeat the following steps to copy the new version to the `/tmp` directory to load and test it.

1. At the prompt, you'll use the **cp -R** command with the **sudo** command to copy your KEXT to `/tmp`. The **cp** command copies files from one place to another and the *-R* option tells **cp** to copy a directory and its entire subtree (like `HelloKernel.kext`). When prompted for a password, enter your `admin` password. (Note that nothing is displayed as you type the password.)

   Warning: If you use tab-completion to avoid typing all of `HelloKernel.kext`, you'll get a "/" after the KEXT name. Be sure to delete this slash before you press Return. If you don't, the **cp** command will copy only the *contents* of the `HelloKernel.kext` directory to `/tmp`, instead of copying the directory and its entire subtree.

   Type the following to copy your KEXT to `/tmp`:

   ```
   % sudo cp -R HelloKernel.kext /tmp
   Password:
   ```

   Check the ownership and permissions of your KEXT by moving to the `/tmp` directory and using the **ls -l** command:

   ```
   % cd /tmp
   % ls -l
   drwxr-xr-x 3 root wheel 102 Jun 19 10:30 HelloKernel.kext
   ```

   The *-l* makes the **ls** command display extra information about the files in a directory.

2. From the `/tmp` directory, use the **kextload** command with the **sudo** command; this loads your KEXT and runs its initialization (`start`) function. Note that you may not have to re-enter your password this time. This is because you're allowed to continue to use **sudo** for a short period (usually about 5 minutes) without reauthenticating.

   For example:

   ```
   % sudo kextload -v HelloKernel.kext
   kextload: extension HelloKernel.kext appears to be valid
   kextload: loading extension HelloKernel.kext
   kextload: sending 1 personality to the kernel
   kextload: HelloKernel.kext loaded successfully
   ```

   The *-v* is optional; it makes **kextload** provide more verbose information.

3. In the other Terminal window, view the system log. In a few moments, this line will appear:

   ```
   localhost kernel: KEXT has loaded!
   ```

4. Use the **kextstat** command to check the status of the KEXT. This command displays the status of all dynamically-loaded KEXTs. Enter the command:

   ```
   % kextstat
   ```

   You'll see several lines of output, including a line for your KEXT at the end.

   ```
   Id Refs Address Size Wired Name (Version) <Linked Against>
    1 0 0x4b94000 0x17000 0x16000 ATIR128 (0.1)
    2 2 0x4bbb000 0x10000 0xf000 com.apple.IOAudioFamily (0.1a)
    3 1 0x4bcb000 0x1e000 0x1d000 com.apple.IOFireWireFamily (0.1a)
   ...
   10 0 0x4d18000 0x7000 0x6000 SIP-NKE (0.1a)
   11 0 0x5047000 0x3000 0x2000 com.MyTutorial.kext.HelloKernel...
   ```

5. Unload the KEXT binary. Use the **kextunload** command with the **sudo** command. This command unloads your KEXT and runs its termination (`stop`) function.

   For example, from the `/tmp` directory:

   ```
   % sudo kextunload HelloKernel.kext
   kextunload: unload kext HelloKernel.kext succeeded
   ```

   Note that you may have to enter your `admin` password this time if it's been more than a few minutes since the last time you entered your password.

6. View the system log. In a few moments, several lines will appear, including:

   ```
   localhost kernel: KEXT will be unloaded
   ```

7. Stop the system log display. The **tail -f** command you used to view the system log will continue to run until you stop it.

In the Terminal window displaying the system log, press the `control` key and the `c` key at the same time.

Using Console Mode

As an alternative to the Terminal application, you can load and test your KEXT from console mode. In console mode, all system messages (such as this kernel extension's message "KEXT has loaded!") are written directly to your monitor screen. Messages appear much more quickly than when they are written to the system log file.

However, you should keep in mind that console mode is not as flexible as the Terminal application. There are no windows, you cannot view your code in Xcode or run other applications at the same time, and you cannot use copy or paste.

To use console mode, follow these steps:

1. Log out of your account.

   From the Desktop, choose Log Out from the Finder menu.

2. From the login screen, log in to console mode.

   Type `>console` as the user name, leave the password blank, and press Return. Be sure to include the `>` character at the beginning of the name. The screen turns black and looks like an old ASCII "glass terminal". This is console mode.

3. At the prompt, log in to your `admin` account.

4. Move to the directory that contains your KEXT binary. Use the **cd** command.

   For example:

   ```
   cd /Users/admin/Projects/HelloKernel/build
   ```

5. Follow the instructions given in "Load the KEXT Binary" (page 26). Remember that the console messages will come directly to your screen; you do not need to view the system log file.

6. When you have finished, log out of console mode by entering the command **logout**.

Where to Go Next

Congratulations! You've now written, built, loaded, and unloaded your own kernel extension. In the next tutorial in this series, "Hello I/O Kit: Creating a Device Driver With Xcode" (page 31), you'll learn how to create a device driver, a special kind of KEXT that allows the kernel to interact with devices.

If you're interested, you can use the **man** command to read the manual pages for **kextload**, **kextstat**, and **kextunload**. For example, from a Terminal window, enter the command

```
man kextstat
```

More information about the 'vers' resource can be found in the "Finder Interface" chapter of *Inside Macintosh—Files*, or online at http://developer.apple.com/documentation/mac/Toolbox/Toolbox-454.html.

Additional useful reference material can be found in Core Foundation Documentation. Look here for documentation about Bundles, Property Lists, Collections (such as Dictionaries) and more.

As you become more experienced in KEXT development, you should also read the articles in this document that cover more advanced topics, such as how to declare dependencies so that your KEXT targets the appropriate version of Mac OS X. The following articles provide more information on KEXT-related topics:

"Loading Kernel Extensions at Boot Time" (page 87)
"Kernel Extension Dependencies" (page 91)
"Kernel Extension Ownership and Permissions" (page 103)

Hello I/O Kit: Creating a Device Driver With Xcode

This tutorial describes how to write an I/O Kit device driver for Mac OS X. The driver you'll create is a simple driver that prints text messages, but doesn't actually control any hardware. The tutorial assumes that you are working in a Mac OS X development environment.

Important: This tutorial does not show you how to build a universal I/O Kit device driver. If you need to do this, you first should work through the tutorial to learn the basics of driver development on Mac OS X. Then, to find out how to configure your Xcode project to build a universal driver, see Technical Note TN2163: Building Universal I/O Kit Drivers.

For information on issues related to building a universal device driver, see Developing a Device Driver to Run on an Intel-Based Macintosh; for information on building universal binaries in general, see *Universal Binary Programming Guidelines, Second Edition*.

Anatomy of a Device Driver

An I/O Kit device driver is a special type of kernel extension (KEXT) that tells the kernel how to handle a particular device or family of devices. In Mac OS X, all kernel extensions are implemented as *bundles*, folders that the Finder treats as single files. Kernel extensions have names ending in `.kext`. In addition, all kernel extensions contain the following:

- A *property list* (plist)—a text file (in XML, Extensible Markup Language, format) that describes the driver's contents, settings, and requirements. This is a required file. A KEXT need contain nothing more than a property list file.
- A *KEXT binary* —Generally, a KEXT has one binary file, but it can have none (locally). However, if it has none, its property list must reference another KEXT and change its default settings. In this way, one KEXT can override or change the behavior of another KEXT. A KEXT binary is also sometimes called a kernel module.
- Optional *Resources*—Resources are useful if your driver needs to display an icon.

In addition, a device driver has several special requirements that other KEXTs do not. Specifically:

- You must create a Personality dictionary for I/O Kit drivers. See "Edit the Driver's Property List" (page 34) for details.
- The I/O Kit provides the initialization and termination routines for drivers; you need not (and should not) specify these routines in your source code or in the driver's Info.plist Entries.

- You must create a header file for a driver. See "Implement the Header File" (page 40).
- The I/O Kit requires several specific entry points, described in the section, "Implement the Driver's Entry Points" (page 42). These must be included in your driver's code.
- The source code for a driver must reference two macros that are defined by I/O Kit and generate runtime type identification information for I/O Kit:
  - `OSDeclareDefaultStructors`
  - `OSDefineMetaClassAndStructors`

  If you don't include these macros in your code, or you include them in the wrong places, your driver will not work properly. See "Implement the Header File" (page 40) and "Implement the Driver's Entry Points" (page 42) for examples of the correct placement of these macros in your code.
- In Mac OS X, kernel-resident device drivers are written in C++. However, it is possible to wrap plain C code inside a C++ class if you need to use non-object-oriented code from another platform or from Mac OS 9.

Roadmap

Here are the steps you'll follow to create the HelloIOKit device driver:

1. "Create an I/O Kit Project using Xcode" (page 32)
2. "Build the Project" (page 44)
3. "Test the Device Driver" (page 45)

You'll use the Xcode application to create and build your driver. You'll use the Terminal application to type the commands to load and test your driver and view the results.

If you have not used Xcode much, you may wish to read *Xcode Quick Tour Guide*.

Create an I/O Kit Project using Xcode

This section describes how to create the project that will be used in writing your device driver. A project is a document that contains all of your files and targets. Targets are the things you can build from your project's files. A simple project has just one target that builds an application. A complex project may contain several targets.

The parts of your project can be found later in your Desktop. These parts include the source files, as well as the targets (your KEXT). Xcode does not store these files in any special format, so you can view or edit the source files with another editing program if you wish. For now, however, we recommend using Xcode.

Here's how you'll create the device driver project:

1. “Create an I/O Kit Extension Project” (page 33)
2. “Edit the Driver’s Build Settings” (page 39)
3. “Edit the Driver’s Property List” (page 34)
4. “Implement the Header File” (page 40)

The examples below assume that you will be logging in as an administrator of your machine. The account name in the examples is `admin`. If you use a different login account, be sure to substitute accordingly. Some of the commands you will be using require `root` privileges so you will be using the **sudo** command. The **sudo** command allows permitted users (such as an `admin` user) to execute a given command with `root` privileges. An admin user is a user who has “Allow user to administer this computer” checked in the Security view of the Accounts system preference. If you use a different login account, make sure it has administrative access.

Create an I/O Kit Extension Project

Device drivers are created and edited in Xcode, Apple’s Integrated Development Environment (IDE).

From a Desktop Finder window, locate and launch the Xcode application, found at `/Developer/Applications/Xcode`. If this is the first time Xcode has been run, you’ll see the new user Assistant. The Assistant asks you to make some decisions about your environment. For now, choose the defaults.

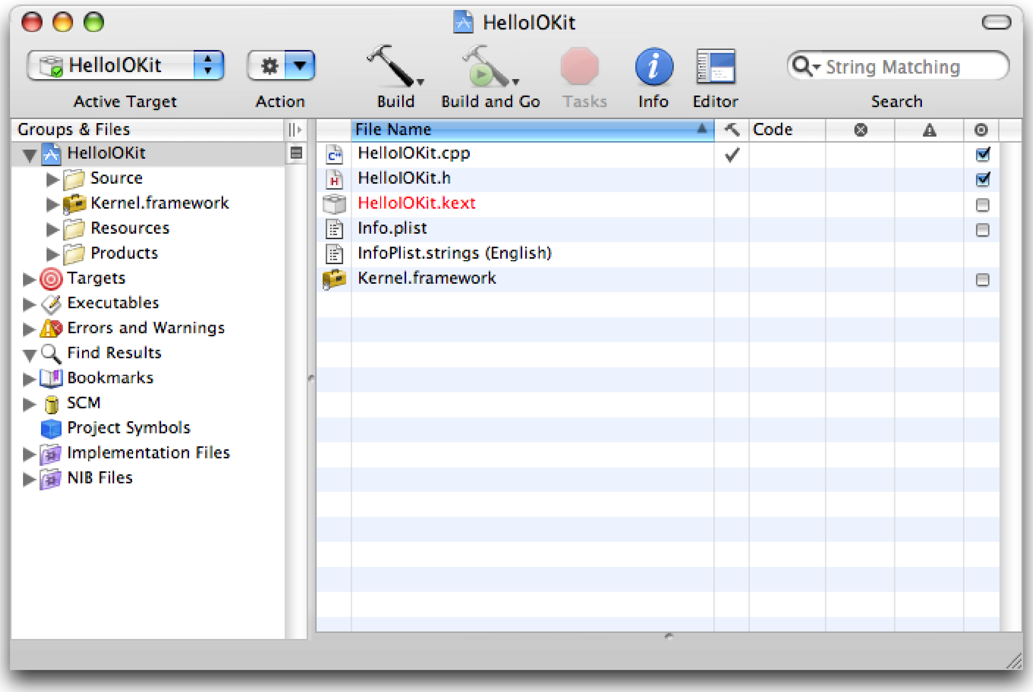
When you have finished with the Assistant, choose New Project from the File menu. In the New Project Assistant, scroll down to the Kernel Extension section and choose IOKit Driver. Click Next.

For the Project Name, enter “HelloIOKit”. The default location is your home directory; however, if you create many projects, you should make a directory to hold them. Edit the Location to specify a “Projects” subdirectory, for example:

```
/Users/admin/Projects
```

When you click Finish, Xcode creates the new project and displays its project window, as shown in Figure 1 (page 34). Notice that the new project contains several files already, including two source files — `HelloIOKit.h` and `HelloIOKit.cpp`.

Figure 1 The new HelloIOKit driver project in Xcode



Edit the Driver's Property List

Your kernel extension contains a property list, or *plist*, which tells the operating system what your KEXT contains and what it needs. If viewed from a text editor, the property list would be in XML format. However, you will be viewing and editing the plist information from within Xcode, so you don't need to worry about the underlying format.

Each element of a property list has a name, or *key*, a Class type (for example, String, Number, *Dictionary*, and so forth), and a value. A Dictionary is a collection of zero or more objects, each of which is associated with a name. Objects may be added, removed, or located in a Dictionary by their name.

By default, Xcode allows you to edit your driver's property list as plain XML text in the Xcode editor window. However, it's easier to view and edit the property list file with the Property List Editor application. You can tell Xcode to open property list files using Property List Editor by following these steps:

1. Choose Xcode > Preferences and click the File Types icon. The File Types pane lists all the folder and file types that Xcode handles and the preferred editor for each type.

2. Click the disclosure triangle for file. Then click the disclosure triangle for text.

3. Select text.xml and click Default (Plain Text File) in the Preferred Editor column.

4. Choose External Editor and select Other from the menu that appears.

5. Select Property List Editor in `/Developer/Applications/Utilities/Property List Editor` and click OK.

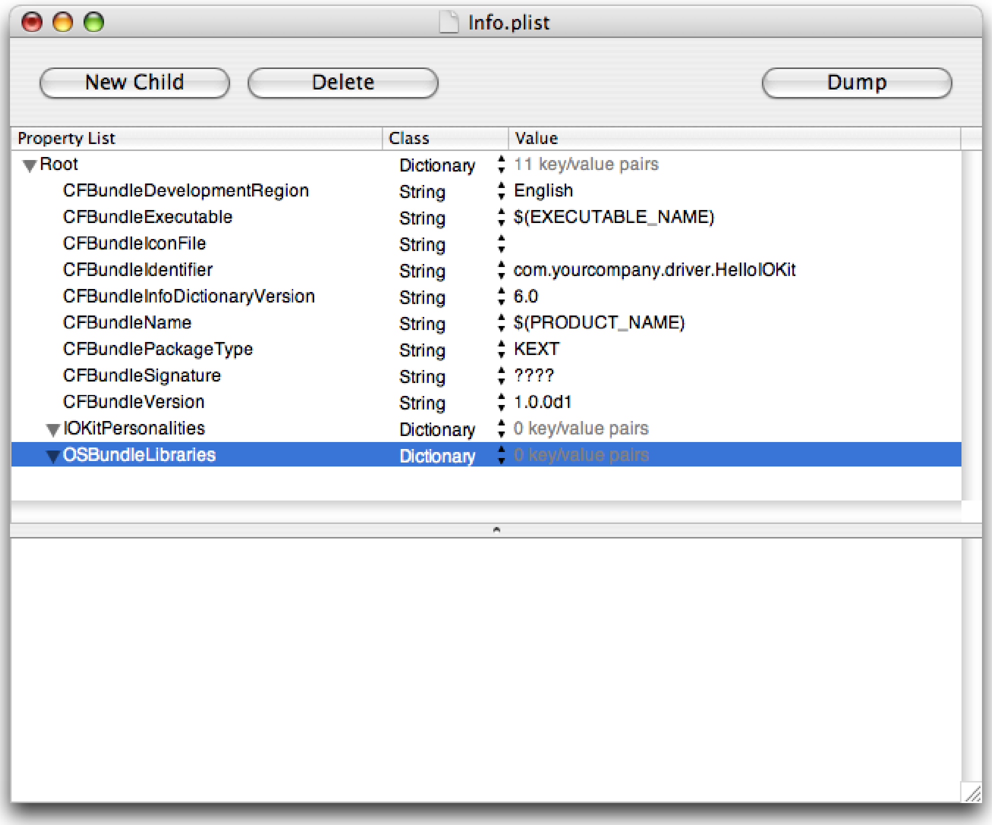
6. Now Xcode lists External Editor (Currently Property List Editor) in the Preferred Editor column for text.xml. Click OK to close the Xcode Preferences window.

Now you can edit your driver's `Info.plist` file using Property List Editor:

1. Select HelloIOKit in the Groups & Files view. Double-click Info.plist in the Xcode project window. Xcode starts Property List Editor, which displays the `Info.plist` file. (If Property List Editor displays a second editing window named Untitled, dismiss it by clicking the red close button in the upper left.)

2. In Property List Editor, click the disclosure triangle next to Root. You should see the elements of the property list file, as shown in Figure 2 (page 35).

Figure 2 Info.plist entries



Property List	Class	Value
▼Root	Dictionary	11 key/value pairs
CFBundleDevelopmentRegion	String	English
CFBundleExecutable	String	$(EXECUTABLE_NAME)
CFBundleIconFile	String	
CFBundleIdentifier	String	com.yourcompany.driver.HelloIOKit
CFBundleInfoDictionaryVersion	String	6.0
CFBundleName	String	$(PRODUCT_NAME)
CFBundlePackageType	String	KEXT
CFBundleSignature	String	????
CFBundleVersion	String	1.0.0d1
▼IOKitPersonalities	Dictionary	0 key/value pairs
▼OSBundleLibraries	Dictionary	0 key/value pairs

3. Change the value of the `CFBundleIdentifier` property from the default chosen by Xcode.

   Apple has adopted a "reverse-DNS" naming convention to avoid namespace collisions. This is important because all kernel extensions share a single "flat" namespace.

By default, Xcode names a new I/O Kit driver `com.yourcompany.driver.`<*projname*>, where <*projname*> is the name you chose for your project. You should replace the first two parts of this name with your actual reverse-DNS prefix (for example: `com.apple`, `ecu.ucsd`, and so forth). For this tutorial, you will use the prefix `com.MyTutorial`.

On the line for `CFBundleIdentifier`, double-click on `com.yourcompany.driver.HelloIOKit` in the Value field. Double-click on `yourcompany` and change this string to `MyTutorial`.

4. The `CFBundleVersion` property contains the version number of your driver in the '`vers`' resource style (for more information on this, see http://developer.apple.com/documentation/mac/Toolbox/Toolbox-454.html and Technical Note TN1132).

   By default, Xcode assigns new kernel extensions the version number `1.0.0d1`. You can change this value if you wish, but your driver must declare *some* version in the '`vers`' resource style to be loaded successfully.

   For this tutorial, leave the `CFBundleVersion` property set to the default value `1.0.0d1`.

5. Darwin/Mac OS X requires every KEXT to declare its dependencies on other loadable extensions or in-kernel components in the `OSBundleLibraries` dictionary at the top level of its plist. Because every KEXT is linked against the kernel, every KEXT must at least declare its dependence on the kernel itself. However, it is often better to declare dependence on specific kernel subcomponents rather than on the kernel as a whole. For more information on dependencies and to find out how to determine the dependencies of other KEXTs you may write, see "Kernel Extension Dependencies" (page 91). For this tutorial, you will add three elements to the `OSBundleLibraries` dictionary to declare your driver's dependence on the kernel subcomponents `com.apple.kernel.iokit`, `com.apple.kernel.libkern`, and `com.apple.kernel.mach`.

   Click on the line for `OSBundleLibraries` and click the disclosure triangle at the beginning of the line. The button previously labeled New Sibling should change to New Child. If not, check to be sure the disclosure triangle next to `OSBundleLibraries` is pointing down.

   Click the button that now says New Child (as soon as you do, it will change back to New Sibling). A new item will appear below `OSBundleLibraries`. Change the name from `New item` to `com.apple.kernel.iokit`. Leave the Class set to `String` and double-click on the Value field and enter 6.9.9.

   Clicking the New Sibling button each time, add two more elements to the `OSBundleLibraries` dictionary.

   ```
   com.apple.kernel.libkern String 6.9.9
   com.apple.kernel.mach String 6.9.9
   ```

6. Note that `IOKitPersonalities` is a Dictionary. Specifically, `IOKitPersonalities` is a representation of an `OSDictionary`, serialized into XML property list format. This Dictionary defines personalities for your driver; each personality contains properties for matching and loading a driver. You will implement one personality, named `HelloIOKit`, within the `IOKitPersonalities` Dictionary. The `HelloIOKit` personality is also a Dictionary, containing additional elements with various types.

   Click on the line for `IOKitPersonalities` and click the disclosure triangle at the beginning of the line. The button previously labeled New Sibling should change to say New Child. If not, check to be sure the disclosure triangle next to `IOKitPersonalities` is pointing down.

Click the button that now says New Child (as soon as you do, it will change back to New Sibling). A new item will appear below `IOKitPersonalities`. Change the name from `New item` to `HelloIOKit`. Click on the Class field for this new item (it currently says `String`). In the popup menu, select `Dictionary`.

With the line for `HelloIOKit` selected, click the disclosure triangle at the beginning of the line. The button previously labeled New Sibling changes to say New Child.

Click the New Child Button (as soon as you do, it will change back to New Sibling). A new item will appear below `HelloIOKit`. Change the name from `New item` to `CFBundleIdentifier` (you can copy and paste the name from above in the property list). For the value, type (or copy and paste) `com.MyTutorial.driver.HelloIOKit`.

Click the button that now says New Sibling. A new item will appear within the `HelloIOKit` dictionary. Change the name from `New item` to `IOClass`. Edit the value to be `com_MyTutorial_driver_HelloIOKit`. Note that this is the same name used before as the CFBundleIdentifier, except that you must use underbars, "_", rather than dots, ".", in the class name. Again, Apple recommends this "reverse-DNS" convention for naming your class, to ensure consistency and reduce name collisions among drivers.

Important: Be sure that the value field of `IOClass` contains underbars, "_", rather than dots, ".", as used in `CFBundleIdentifier`. This string will be used to create the class for your device driver.

Now you will create the property list elements that define a successful match for your driver, so that it can be loaded. You'll also add the `IOKitDebug` property to help in debugging driver matching.

Note: If you include the `IOKitDebug` property with a nonzero value in your driver's property list, it will provide information about your driver's matching and loading. When you write a functional driver following the guidelines in this tutorial, however, you should give the `IOKitDebug` property a zero value because a nonzero value will prevent your driver from loading into the kernel during a safe-boot. For more information on safe booting, see Loading Kernel Extensions at Boot Time (page 87).

Important: `IOMatchCategory` is a special property list element that allows multiple drivers to match on a single nub. The presence of `IOMatchCategory` allows `HelloIOKit` to match on `IOResources` (a special nub at the root of the I/O Registry that provides system-wide resources) without preventing other drivers from matching on it. When you write a functional driver following the guidelines in this tutorial, you should *not* include the `IOMatchCategory` element in your driver's property list at all unless there is a valid reason for your driver to match on the same device nub at the same time as another driver (for example, a serial port with multiple devices attached to it).

The vast majority of drivers should not match on `IOResources`. A rare exception to this is a driver for a virtual device, because a virtual device does not produce its own nub in the I/O Registry. If you're developing such a driver, you must include the `IOMatchCategory` property to make sure your driver doesn't claim `IOResources` and prevent other drivers from matching on it. To ensure your driver's `IOMatchCategory` value is unique, use your driver's class name, in reverse-DNS notation and with underbars instead of dots (for example, `com_MyTutorial_driver_HelloIOKit`).

Clicking the button that says New Sibling each time, add the following property list elements. Be sure to change the class of the first, `IOKitDebug`, from a string to a number using the popup menu. Be sure to enter the names and values exactly as shown below.

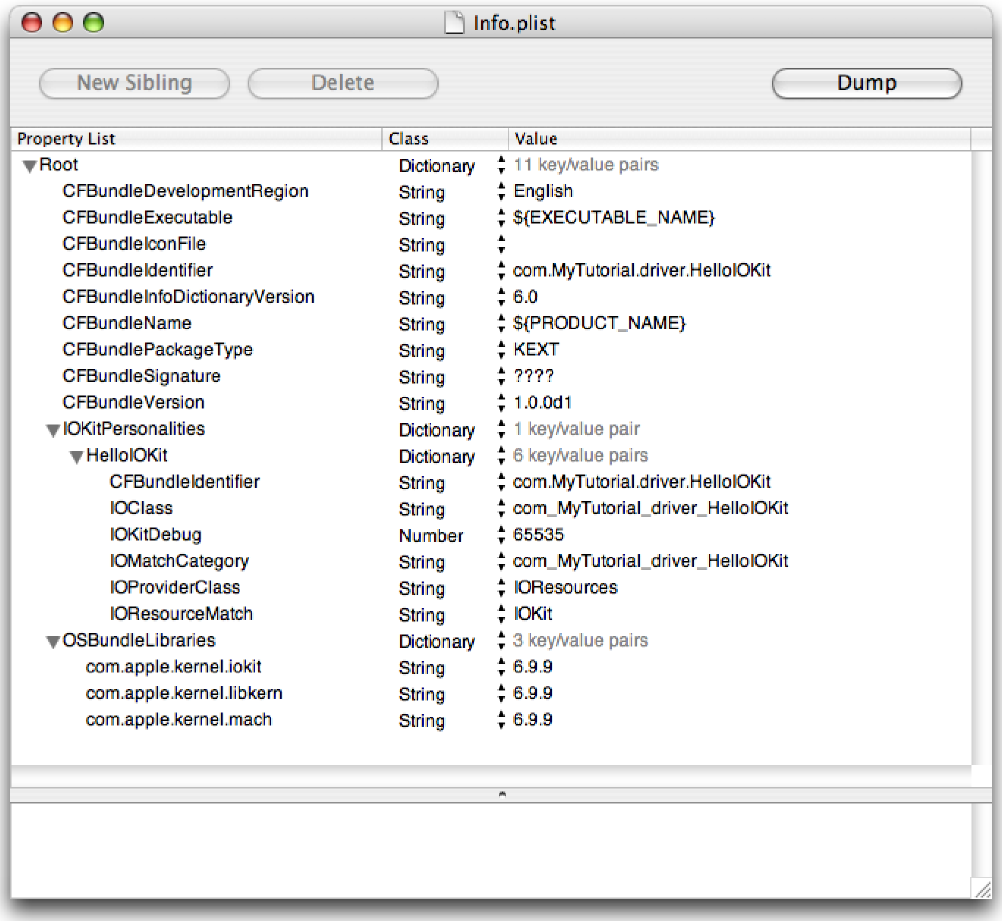
Table 1 HelloIOKit personality dictionary values

Name	Class	Value
`IOKitDebug`	Number	`65535`
`IOMatchCategory`	String	`com_MyTutorial_driver_HelloIOKit`
`IOProviderClass`	String	`IOResources`
`IOResourceMatch`	String	`IOKit`

Important: If you insert these elements in a different order, Property List Editor immediately rearranges the list to be alphabetical. As you add or edit each element, pay careful attention to be sure you are editing the line you think!

When you have finished adding property list elements, the screen should look like the example shown in Figure 3 (page 39).

Figure 3 Info.plist entries after additions



Property List	Class	Value
▼Root	Dictionary	11 key/value pairs
CFBundleDevelopmentRegion	String	English
CFBundleExecutable	String	${EXECUTABLE_NAME}
CFBundleIconFile	String	
CFBundleIdentifier	String	com.MyTutorial.driver.HelloIOKit
CFBundleInfoDictionaryVersion	String	6.0
CFBundleName	String	${PRODUCT_NAME}
CFBundlePackageType	String	KEXT
CFBundleSignature	String	????
CFBundleVersion	String	1.0.0d1
▼IOKitPersonalities	Dictionary	1 key/value pair
▼HelloIOKit	Dictionary	6 key/value pairs
CFBundleIdentifier	String	com.MyTutorial.driver.HelloIOKit
IOClass	String	com_MyTutorial_driver_HelloIOKit
IOKitDebug	Number	65535
IOMatchCategory	String	com_MyTutorial_driver_HelloIOKit
IOProviderClass	String	IOResources
IOResourceMatch	String	IOKit
▼OSBundleLibraries	Dictionary	3 key/value pairs
com.apple.kernel.iokit	String	6.9.9
com.apple.kernel.libkern	String	6.9.9
com.apple.kernel.mach	String	6.9.9

7. Click File > Save in the Property List Editor menu to save your changes. Then choose Property List Editor > Quit Property List Editor to return to your project in Xcode.

Edit the Driver's Build Settings

Xcode defines several settings that contain information about your KEXT that get compiled into its executable. You will be editing a few of these settings to make sure they correspond to elements in the driver's plist.

1. On the left, in the Groups & Files view, click the disclosure triangle next to Targets and select HelloIOKit (you don't have to click the disclosure triangle next to HelloIOKit).

2. Click the Get Info button in the tool bar (it's the round blue button with an "i" in the middle).

3. Select the Build view and be sure Customized Settings is selected in the Collection pop-up menu. Scroll down to the bottom of the Customized Settings list and change the name of the module from the default chosen by Xcode (the value of the `MODULE_NAME` setting).

   Click on the line for `MODULE_NAME` and double-click on `com.yourcompany.driver.HelloIOKit` in the Value field. Double-click on `yourcompany` and change this string to `MyTutorial`. Be sure that the `MODULE_NAME` value matches the value of the `CFBundleIdentifier` you entered in your driver's `Info.plist` file or your driver will not run.

4. Note that the `MODULE_START` and `MODULE_STOP` settings are not listed. The I/O Kit provides the initialization and termination routines for drivers; you need not (and should not) specify these methods in your source code or in the driver's build settings.

5. The value of the `MODULE_VERSION` setting is in the `'vers'` resource style and must be numerically identical to the `CFBundleVersion` value in the driver's plist or your driver may not load.

   For this tutorial, make sure `MODULE_VERSION` has the value `1.0.0d1`.

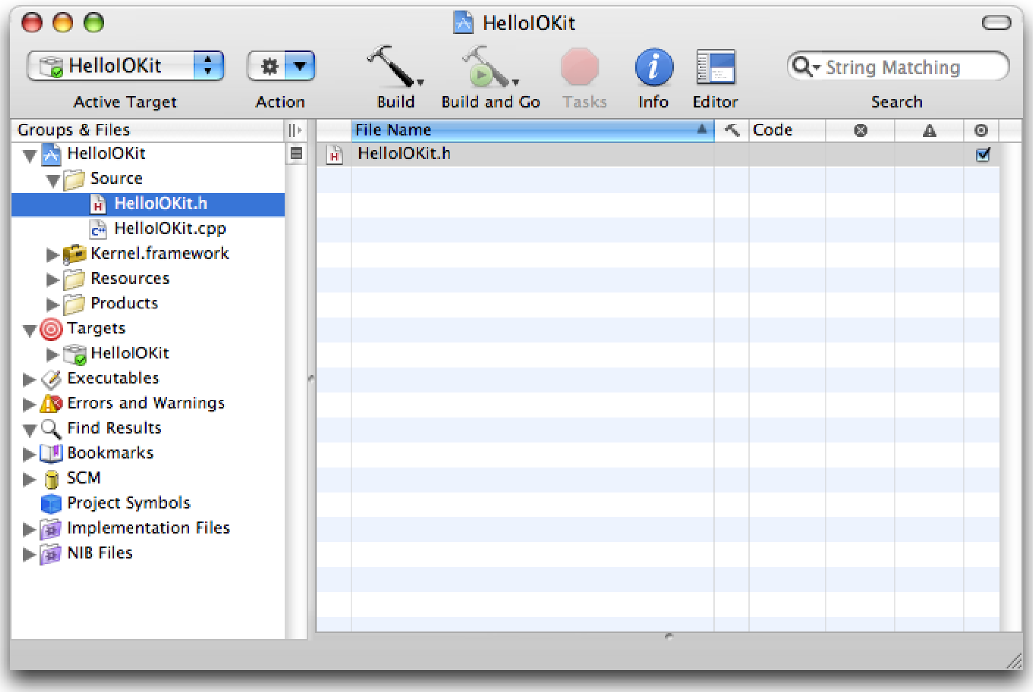
   Important: You must make sure that the `MODULE_VERSION` value in the Customized Settings panel is numerically equal to the `CFBundleVersion` string in the `Info.plist` file or your driver may not load. You must also make sure the `MODULE_NAME` value in the Customized Settings panel matches the `CFBundleIdentifier` string in the `Info.plist` file or your driver will not run.

Before going on to the next section, you might want to go back to the Property List Editor view of the `Info.plist` file and copy the string `com_MyTutorial_driver_HelloIOKit`; you'll need it for the next few steps. To do this, click the disclosure triangle next to HelloIOKit in the Groups & Files view, click the disclosure triangle next to Resources, and double-click Info.plist. View the Info.plist file by clicking the disclosure triangle next to Root. Select `com_MyTutorial_driver_HelloIOKit` in the Value field for the `IOClass` member of the HelloIOKit personality and choose Copy from the Edit menu. Then choose Property List Editor > Quit Property List Editor to return to Xcode.

Implement the Header File

With the disclosure triangle next to HelloIOKit pointing down, click on the disclosure triangle next to Source. Double-click on `HelloIOKit.h` to display the header file. The default header file does nothing. You need to add code before it does anything useful. Figure 4 (page 41) shows where you will find the `HelloIOKit.h` file in the project window.

Figure 4 Viewing source code in Xcode



Edit the contents of `HelloIOKit.h` to match the code in Listing 1 (page 42), pasting the string you copied from the `Info.plist` file (`com_MyTutorial_driver_HelloIOKit`) where shown.

Notice that the first line of `HelloIOKit.h` includes the header file `IOService.h`. This header file defines many of the methods and services on which device drivers depend. The header file is located in the `IOKit` folder of `Kernel.framework`. When you develop your own driver, be sure only to include header files from `Kernel.framework` (in addition to header files you create) because only these files have meaning in the kernel environment. If you include other header files, your driver might compile but the functions and services defined in those header files would not be available in the kernel.

Pay special attention to the lines that contain the string `com_MyTutorial_driver_HelloIOKit`. If this were not a tutorial, the actual name of your driver's class (as copied and pasted) would go here.

Important: Be very sure that this string is exactly as entered for the value of `IOClass` in the `Info.plist` file. Other than changing dots to underbars, this name should also be identical to the CFBundleIdentifier defined in the `Info.plist` file and to the `MODULE_NAME` in the Customized Settings panel. If these strings do not match, your driver will not load and run properly.

Aside from the importance of the class name, the `OSDeclareDefaultStructors` macro used in this file is very important. If you don't use this macro correctly, or in the proper place, your driver won't run.

In the header file, the `OSDeclareDefaultStructors` macro must be the first line in the class's declaration. It takes one argument: the class's name. It declares the class's constructors and destructors for you, in the manner the I/O Kit expects.

Edit `HelloIOKit.h` to match the code below:

Listing 1 HelloIOKit.h

```
#include <IOKit/IOService.h>
class com_MyTutorial_driver_HelloIOKit : public IOService
{
OSDeclareDefaultStructors(com_MyTutorial_driver_HelloIOKit)
public:
    virtual bool init(OSDictionary *dictionary = 0);
    virtual void free(void);
    virtual IOService *probe(IOService *provider, SInt32 *score);
    virtual bool start(IOService *provider);
    virtual void stop(IOService *provider);
};
```

When you have finished editing `HelloIOKit.h`, copy the class name string (`com_MyTutorial_driver_HelloIOKit`) again; you will need it in the next step. Then double-click on `HelloIOKit.cpp` in the Groups & Files view of the Xcode project window. Again, the default file does nothing. You will need to add code before it can do anything useful.

Implement the Driver's Entry Points

The following list describes some of the entry points that the I/O Kit uses. You will need to implement these entry points in your driver's source file.

Most of the methods come in pairs: one performs some action and creates some data structures, the other undoes that action and releases those data structures. Generally a module must define only the `probe`, `start` and `stop` methods, using its superclass's definitions for the rest.

- The `init` method is the first instance method called on each instance of your driver class. By convention, the first call to any newly created object is `init`; this method will only be called once on each instance. The `free` method is the last one called on any object. Any resources allocated in `init` should be disposed of in `free`. A driver is unloaded as a result of all its objects being freed. Note that the `init` method operates on objects; this is your opportunity to prepare an object to receive calls. Use `probe` or `start` to do the real driver work.
- The `probe` method is called if your driver needs to talk to the hardware to determine whether there's a match. This method must leave the hardware in a good state when it returns, as other drivers may probe it as well.
- The `start` method is the first one called in the actual driver life cycle; it tells the driver to start driving hardware. After `start` is called, the driver can begin publishing nubs and vending services. The `stop` method is the first to be called before your driver is unloaded. When `stop` is called, your driver should unpublish all existing nubs; no more nubs will be published and the driver stops vending services. The `start` and `stop` methods talk to the hardware via your driver's provider.

Important: Be very sure that the class name is exactly the same string you used in `HelloIOKit.h` and entered for the value of the `IOClass` member of the HelloIOKit personality. Other than changing dots to underbars, this name should also be identical to the CFBundleIdentifier defined in the `Info.plist` file and to the `MODULE_NAME` in the Customized Settings panel. If these strings do not match, your driver will not load and run properly.

Aside from the importance of the class name, the `OSDefineMetaClassAndStructors` macro used in this file is very important. If you don't use this macro correctly, or in the proper place, your driver won't run.

In the C++ source file, the `OSDefineMetaClassAndStructors` macro must appear before you define any of your class's methods. This macro takes two arguments: your class's name and the name of your class's superclass. The macro defines the class's constructors, destructors, and several other methods I/O Kit requires.

Notice that `HelloIOKit.cpp` includes only `Kernel.framework` header files, in addition to `HelloIOKit.h`. Aside from header files you define, you should include only header files from `Kernel.framework` in your driver.

Edit `HelloIOKit.cpp` to match the code below:

Listing 2 HelloIOKit.cpp

```
#include <IOKit/IOLib.h>
#include "HelloIOKit.h"
extern "C" {
#include <pexpert/pexpert.h>//This is for debugging purposes ONLY
}

// Define my superclass
#define super IOService

// REQUIRED! This macro defines the class's constructors, destructors,
// and several other methods I/O Kit requires. Do NOT use super as the
// second parameter. You must use the literal name of the superclass.
OSDefineMetaClassAndStructors(com_MyTutorial_driver_HelloIOKit, IOService)

bool com_MyTutorial_driver_HelloIOKit::init(OSDictionary *dict)
{
    bool res = super::init(dict);
    IOLog("Initializing\n");
    return res;
}

void com_MyTutorial_driver_HelloIOKit::free(void)
{
    IOLog("Freeing\n");
    super::free();
}

IOService *com_MyTutorial_driver_HelloIOKit::probe(IOService *provider, SInt32
*score)
{
    IOService *res = super::probe(provider, score);
    IOLog("Probing\n");
    return res;
}

bool com_MyTutorial_driver_HelloIOKit::start(IOService *provider)
{
    bool res = super::start(provider);
    IOLog("Starting\n");
    return res;
}

void com_MyTutorial_driver_HelloIOKit::stop(IOService *provider)
{
    IOLog("Stopping\n");
    super::stop(provider);
}
```

The `IOLog` method is similar to `printf`, but runs faster and flushes its output more frequently.

Two additional methods are implemented by I/O Kit and rarely, if ever, should be overridden by your driver code. `attach` is called by your driver's provider after a successful match; it causes your driver to be added to the I/O Registry. `detach` is called after an unsuccessful `probe`, or when a driver is removed from the I/O Registry.

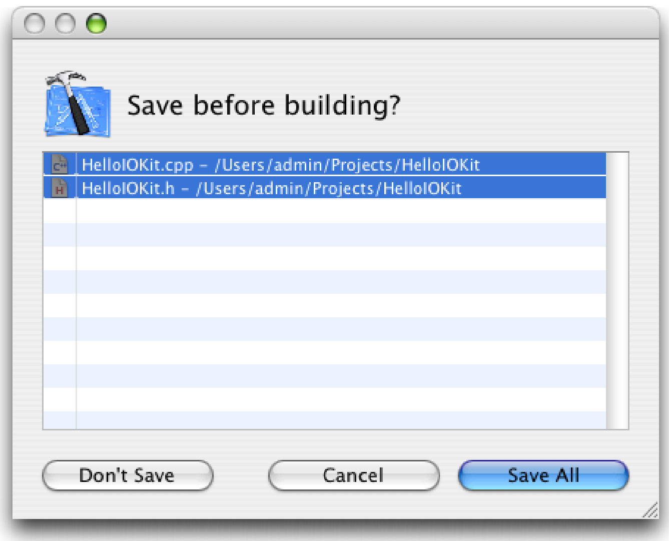
Save your changes by choosing File > Save in the Xcode menu.

Build the Project

Click the Build button (it looks like a hammer) in the upper left corner of the Xcode editor window or select Build from the Build menu.

If you didn't save earlier, Xcode asks you whether to save some modified files. If this is the case, select all the files and click "Save All". Figure 5 (page 45) shows the Save dialog.

Figure 5 Save before building


Save before building?

HelloIOKit.cpp – /Users/admin/Projects/HelloIOKit
HelloIOKit.h – /Users/admin/Projects/HelloIOKit

Xcode starts building your project. It stops if it reaches an error in the code.

Fix Any Errors

If you made any errors typing in the code, Xcode will stop and show you the errors, in both the editor window (if you haven't closed it) and the main project window. In the project window, click the disclosure triangle next to Errors and Warnings in the Groups & Files view to reveal the files that contain errors. If you select a file listed under Errors and Warnings, Xcode lists the errors in that file, each accompanied by a short description of the problem. Double-click the file name to open an editor window and edit the source code. Correct any errors and build the project again.

Test the Device Driver

This section shows how to test the driver. You'll load your driver using the **kextload** command, you'll use the **kextstat** command to see that the driver has been loaded, and finally, you'll unload your driver from the kernel using the **kextunload** command.

You'll use the Terminal application to type the commands to load and unload your module. You'll view the results as they are written to the system log file, `/var/log/system.log`.

Note: This tutorial uses the "`%`" prompt when it shows the commands you type in the Terminal application. This is the default prompt of the `tcsh` shell. If you're using a different shell, you may see a different prompt. The prompt in the `bash` shell, which is the default shell in Mac OS X version 10.3, is "`$`".

Note that you use **kextload** and **kextunload** only when testing a driver. When a KEXT is fully installed under Darwin/Mac OS X, the Kernel Extension Manager takes care of loading and running (and unloading) drivers.

The following sections show you how to test your driver.

Getting Root Privileges

To use the **kextload** and **kextunload** commands you must have `root` or super user privileges. Instead of logging in as `root`, for this tutorial you will use the **sudo** command which gives permitted users the ability to execute a given command as `root`.

By default, Darwin/Mac OS X allows `admin` and `root` users to use the **sudo** command, so make sure you are currently logged in to an `admin` account before using **sudo**. Recall that an admin user is a user who has "Allow user to administer this computer" checked in the Security view of the Accounts system preference.

Start the Terminal Application

1. Start the Terminal application. From a Desktop Finder window, locate and launch the Terminal application, found at `/Applications/Utilities/Terminal`.

2. To view the system log file, enter the following command at the Terminal prompt:

   ```
   % tail -f /var/log/system.log
   ```

3. Open a second window in the Terminal application. Choose File > New Shell from the Terminal menu. Position this window so that you can view both windows easily. You will load your driver from the second window.

4. In the second Terminal window, move to the directory that contains your driver. Xcode stores your driver in the `Debug` folder of the `build` directory of your project location (unless you've set a different location for build products using Xcode's Preferences dialog).

   Use the **cd** command to move to the appropriate directory. For example:

   ```
   % cd Projects/HelloIOKit/build/Debug
   ```

   This directory contains your KEXT. You can use the **ls** command to view the contents of this directory. For example:

   ```
   % ls
   HelloIOKit.kext
   ```

   Your KEXT should have the name `HelloIOKit.kext`. Note that this name is formed from the Project name and a suffix, `.kext`.

Important: For purposes of packaging, distribution, and installation, the filename of the KEXT (apart from the suffix) does not matter. However, the name of the KEXT binary and the KEXT's class (stored in the KEXT's property list) should be unique, using the recommended "reverse-DNS" naming convention.

From a Desktop Finder window, a KEXT appears as a single file (look for it from the Desktop if you like). From the Terminal application, however, a KEXT appears as a directory. The KEXT directory contains the contents of your KEXT, including the plist (`Contents/Info.plist`) and the KEXT binary (`Contents/MacOS/HelloIOKit`).

5. View the contents of the KEXT directory. Use the **find** command.

   For example:

```
% find HelloIOKit.kext
HelloIOKit.kext
HelloIOKit.kext/Contents
HelloIOKit.kext/Contents/Info.plist
HelloIOKit.kext/Contents/MacOS
HelloIOKit.kext/Contents/MacOS/HelloIOKit
HelloIOKit.kext/Contents/Resources
HelloIOKit.kext/Contents/Resources/English.lproj
HelloIOKit.kext/Contents/Resources/English.lproj/InfoPlist.strings
HelloIOKit.kext/Contents/Resources/Info.plist
```

You are now ready to load and run (and then unload) your driver.

Load the Driver

Because kernel extensions contain code and data that are loaded into the kernel, the most protected environment in the operating system, their file ownership and permissions must be set to prevent unauthorized tampering. All KEXT bundles (all files and folders in the KEXT, including the KEXT binary) must be owned by the user `root` and the group `wheel`. In addition, the folders and files of the KEXT bundle must have their permissions set so that they are not writable by any user other than the super user.

For development purposes, however, you can make a `root`-owned copy of your KEXT binary to load and test. You should not change the ownership and permissions of any of your KEXT files in your own directory, because you will no longer be able to save them after working on them. For this tutorial, you will use the **sudo** command to copy the KEXT binary (`HelloIOKit.kext`) to the `/tmp` directory and load and unload the KEXT from there. Using **sudo** to copy the KEXT binary gives the KEXT the super user's ownership and permissions and leaves the original KEXT alone so you can revise and save it as you choose.

Note: Every time you make changes to your KEXT and rebuild it, you need to repeat the following steps to copy the new version to the `/tmp` directory to load and test it.

1. At the prompt, you'll use the **cp -R** command with the **sudo** command to copy your driver to `/tmp`. The **cp** command copies files from one place to another and the `-R` option tells **cp** to copy a directory and its entire subtree (like `HelloIOKit.kext`). When prompted for a password, enter your `admin` password. (Note that nothing is displayed as you type the password.)

   Warning: If you use tab-completion to avoid typing all of `HelloIOKit.kext`, you'll get a "/" after the KEXT name. Be sure to delete this slash before you press Return. If you don't, the **cp** command will copy only the *contents* of the `HelloIOKit.kext` directory to `/tmp`, instead of copying the directory and its entire subtree.

   Type the following to copy your driver to `/tmp`:

   ```
   % sudo cp -R HelloIOKit.kext /tmp
   Password:
   ```

   Check the ownership and permissions of your driver by moving to the `/tmp` directory and using the **ls -l** command:

   ```
       % cd /tmp
       % ls -l
       drwxr-xr-x 3 root wheel 102 Jun 19 10:30 HelloIOKit.kext
   ```

   The `-l` makes the **ls** command display extra information about the files in a directory.

2. From the `/tmp` directory, use the **kextload** command with the **sudo** command; this loads your driver, runs its initialization (`start`) method, and registers it with the kernel. Note that you may not have to re-enter your password this time. This is because you're allowed to continue to use **sudo** for a short period (usually about 5 minutes) without reauthenticating.

   For example:

   ```
   % sudo kextload -v HelloIOKit.kext
   kextload: extension HelloIOKit.kext appears to be valid
   kextload: notice: extension HelloIOKit.kext has debug properties set
   kextload: loading extension HelloIOKit.kext
   kextload: HelloIOKit.kext loaded successfully
   kextload: loading personalities named:
   kextload:   HelloIOKit
   kextload: sending 1 personality to the kernel
   kextload: matching started for HelloIOKit.kext
   ```

   The `-v` is optional; it makes **kextload** provide more verbose information.

3. In the other Terminal window, view the system log. In a few moments, several lines will appear, for example:

```
Matching service count = 1
Initializing
com_MyTutorial_driver_HelloIOKit::probe(IOResources)
Probing
com_MyTutorial_driver_HelloIOKit::start(IOResources) <1>
Starting
```

4. Use the **kextstat** command to check the status of the driver. This command displays the status of all dynamically-loaded kernel modules. Enter the command:

   ```
   % kextstat
   ```

   You'll see several lines of output, including a line for your driver (at the end).

   ```
   Id  Refs AddressSizeWiredName (Version) <Linked Against>
    1  0   0x4b940000x170000x16000ATIR128 (0.1)
    2  2   0x4bbb0000x100000xf000com.apple.IOAudioFamily (0.1a)
   10  0   0x4d180000x70000x6000SIP-NKE (0.1a)
   11  0   0x50c0000 0x40000x3000 com.MyTutorial.driver.HelloIOKit (1.0.0d1) <10>
   ...
   ```

5. Unload the driver. Use the **kextunload** command with the **sudo** command. This unloads your driver by running its termination (`stop`) function.

   For example, from the `/tmp` directory:

   ```
   % sudo kextunload HelloIOKit.kext
   unload kext HelloIOKit.kext succeeded
   ```

   Note that you may have to enter your `admin` password this time if it's been more than a few minutes since the last time you entered your password.

6. View the system log. In a few moments, several lines will appear, for example:

   ```
   Stopping
   Freeing
   ```

7. Stop the system log display. The **tail -f** command you used to view the system log will continue to run until you stop it.

   In the Terminal window displaying the system log, press the `control` key and the `c` key at the same time.

Using Console Mode

As an alternative to the Terminal application, you can load and test your KEXT from console mode. In console mode, all system messages (such as this kernel extension's message "Probing") are written directly to your monitor screen. Messages appear much more quickly than when they are written to the system log file.

However, you should keep in mind that console mode is not as flexible as the Terminal application. There are no windows, you cannot view your code in Xcode or run other applications at the same time, and you cannot use copy or paste.

To use console mode, follow these steps:

1. Log out of your account.

   From the Desktop, choose Log Out from the Finder menu.

2. From the login screen, log in to console mode.

   Type `>console` as the user name, leave the password blank, and press Return. Be sure to include the `>` character at the beginning of the name. The screen turns black and looks like an old ASCII "glass terminal". This is console mode.

3. At the prompt, log in to your `admin` account.

4. Move to the directory that contains your KEXT. Use the **cd** command.

   For example:

   ```
   cd /Users/admin/Projects/HelloIOKit/build
   ```

5. Follow the instructions given in "Load the Driver" (page 47).

6. Remember that the console messages will come directly to your screen; you do not need to view the system log file.

7. When you have finished, log out of console mode by entering the command **logout**.

Where to Go Next

Congratulations! You've now written, built, loaded, and unloaded a device driver. In the next tutorial in this series, "Hello Debugger: Debugging a Device Driver With GDB" (page 53), you'll learn how to debug your driver using a two-machine debugging environment.

If you're interested, you can use the **man** command to read the manual pages for **kextload**, **kextstat**, and **kextunload**. For example, from a Terminal window, enter the command

```
man kextstat
```

More information about the 'vers' resource can be found in the "Finder Interface" chapter of *Inside Macintosh—Files*, or online at http://developer.apple.com/documentation/mac/Toolbox/Toolbox-454.html.

Additional useful reference material can be found in Core Foundation Documentation. Look here for documentation about Bundles, Property Lists, Collections (such as Dictionaries) and more.

As you become more experienced in driver development, you should also read the articles in this document that cover more advanced topics, such as how to declare dependencies so that your driver targets the appropriate version of Mac OS X. The following articles provide more information on these topics:

"Loading Kernel Extensions at Boot Time" (page 87)
"Kernel Extension Dependencies" (page 91)
"Kernel Extension Ownership and Permissions" (page 103)

In addition, you might want to read Technical Note TN2163, "Building Universal I/O Kit Drivers" to learn the steps you need to take to configure an Xcode I/O Kit driver project to produce a universal driver.

Hello Debugger: Debugging a Device Driver With GDB

This tutorial describes how to prepare to debug a device driver for Darwin/Mac OS X. You will learn how to set up a two-machine debugging environment and how to start using GDB, a command-line debugger, to perform remote debugging.

The tutorial assumes that you are working in a Mac OS X development environment.

Although this tutorial is written with a device driver as the example, the steps for debugging are similar for debugging any type of kernel extension (KEXT). If you wish, you can substitute your own code for the example. Note, however, that you may encounter a few inconsistencies. For example, examples of GDB commands may be dependent on the underlying source code language—I/O Kit extensions (drivers) use C++; the GDB commands for C may differ.

Preparation

Before you begin this tutorial, make sure you have met the following requirements.

1. Be sure you understand the material presented in the KEXT and driver development tutorials.

   The tutorials "Hello Kernel: Creating a Kernel Extension With Xcode" (page 15) and "Hello I/O Kit: Creating a Device Driver With Xcode" (page 31) describe how to create a kernel extension's project, correct code errors, and how to create a simple I/O Kit device driver. This tutorial uses the example from "HelloIOKit". Be sure that you have completed these tutorials (or are familiar with KEXT or driver development in Mac OS X) before beginning this one.

2. For remote debugging, you need two machines. On the *development machine*, you create, build, and, for this tutorial, debug your driver. On the *target machine*, you load and run the driver.

   Before you begin this tutorial, you should prepare the two machines. Note that:

   - Both machines must be running the same version of Mac OS X.
   - Both machines must be connected via Ethernet using the built-in Ethernet ports.
   - If you're running a version of Mac OS X earlier than Mac OS X version 10.3, both machines must be on the same subnet.
   - You must have login access to both machines; this tutorial assumes you are logging in as an administrator of your machine (in this tutorial, the user `admin`). You will need `root` privileges for some of the commands so you will be using the **sudo** command which allows permitted

users to execute a given command as `root`. If you are using another account, substitute that account name in the examples that follow, but make sure that account has administrative access.

- Make sure that Personal File Sharing and FTP Access are enabled for the target machine. Use the Sharing panel in the System Preferences (click System Preferences in the Apple menu or in the Dock) to enable FTP Access and Personal File Sharing.
- If the target machine has Apple Remote Desktop enabled, you should disable it before you continue with this tutorial. To do this, open System Preferences, click Sharing, and uncheck the Apple Remote Desktop checkbox in the Services pane.

3. Transfer a copy of your KEXT to the target machine.

   Before you can begin to debug your driver (and each time you rebuild it), you must transfer a copy of your KEXT to the target machine. You can do this in any of several ways. For example:

   - Use File Sharing to transfer your KEXT over the network.
   - Use the **tar** and **ftp** commands (from within the Terminal application), to package your KEXT and transfer it over the network.

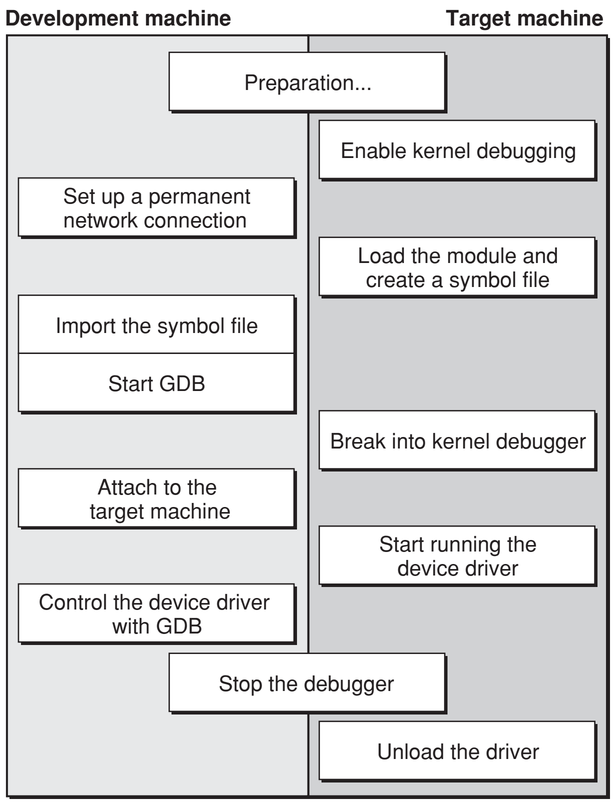
   You can also automate this process using **ssh**. This tutorial does not tell you how to do this, but you can find resources on the web that describe the process.

Roadmap

This tutorial has many steps. It is important to keep the two machines clear in your mind as you work through the steps. It may help if you take a piece of paper, tear it in half, and write “Development” on one piece and “Target” on the other. Then place the pieces of paper next to the two machines.

You will be moving back and forth between the two machines many times. Figure 1 (page 55) illustrates the steps you will take for each machine.

Figure 1 Steps to set up a debugging environment


Development machine
Target machine
Preparation...
Enable kernel debugging
Set up a permanent network connection
Load the module and create a symbol file
Import the symbol file
Start GDB
Break into kernel debugger
Attach to the target machine
Start running the device driver
Control the device driver with GDB
Stop the debugger
Unload the driver

After reading "Preparation" (page 53), you should have already prepared the two machines and attached them to the network. The next two steps will enable kernel debugging and set up a permanent network connection between the two machines.

1. "On Target Machine, Enable Kernel Debugging" (page 56)

2. "On Development Machine, Set Up a Permanent Network Connection" (page 57)

Once these steps are completed, you will not need to repeat them, even if you need to start over part way through the tutorial. The next set of steps prepare the driver for debugging, set up GDB, and attach the two machines to begin debugging your driver. If you need to start over part way through the tutorial, you should repeat all of these steps from the beginning.

1. "Create a Symbol File" (page 58)

2. "On Development Machine, Import the Symbol File" (page 61)

3. "On Development Machine, Start GDB " (page 62)

4. "On Target Machine, Break into Kernel Debugging Mode" (page 63)

5. "On Development Machine, Attach to the Target Machine" (page 64)

6. "On Target Machine, Start Running the Device Driver" (page 65)

7. "On Development Machine, Control the Driver With GDB" (page 65)

The last two steps stop the debugger and unload the driver. If you need to start over part way through, you may need to skip ahead to these steps to reset the environment completely, then start again at "Create a Symbol File" (page 58).

1. "Stop the Debugger" (page 66)
2. "On Target Machine, Unload the Driver" (page 67)

On Target Machine, Enable Kernel Debugging

In this step and the next, you will set up the two-machine environment. These two steps will enable kernel debugging and set up a permanent network connection between the two machines.

By default, Darwin/Mac OS X does not let you debug the kernel. Before you can debug your driver, you must first enable kernel debugging. On the target machine, do the following:

1. Start the Terminal application. From a Finder window, locate and launch the Terminal application, found at `/Applications/Utilities/Terminal`.

2. Set the kernel debug flags.

   To enable kernel debugging, you will be setting an NVRAM (non-volatile random access memory) variable. To do this, you must use the **sudo** command which gives permitted users the ability to execute a given command as `root`. When **sudo** prompts you for a password, enter your `admin` password. (Note that nothing is displayed as you type the password.)

   The **sudo** command will allow you to execute commands with `root` privileges without re-entering your password for a few minutes (usually 5) but after that time is up, it will ask you for your password again. This tutorial does not show the password prompt every time the **sudo** command is used.

   If you're using a beige Power Macintosh G3 and you're running a version of Mac OS X prior to 10.1, enter this command:

   ```
   $ sudo nvram boot-command="0 bootr debug=0x14e"
   ```

   If your target machine is a PowerPC-based Macintosh model not listed above or an Intel-based Macintosh, enter this command:

   ```
   $ sudo nvram boot-args="debug=0x14e"
   Password:
   ```

   For more information on debugging flags, see Building and Debugging Kernels in *Kernel Programming Guide*.

3. Restart the computer.

   The computer restarts and displays the login screen.

On Development Machine, Set Up a Permanent Network Connection

Your development and target machines must be continuously connected by a reliable network connection. In this section, you will create such a connection.

Note: If your target machine is running Mac OS X 10.3 or later and is configured with the 0x40 debug flag set (`DB_ARP`), you can skip this section.

After the target machine has restarted, do the following steps on the development machine:

1. Start the Terminal application. From a Finder window, locate and launch the Terminal application, found at `/Applications/Utilities/Terminal`.

2. Make sure your development machine can connect to your target machine.

   Use the **ping** command with your target machine's hostname or IP address. For example:

   ```
   $ ping -c 1 target.apple.com
   ```

   or

   ```
   $ ping -c 1 192.102.207.119
   ```

   This command makes sure your development machine can reach your target machine and creates a temporary connection between them. The `-c 1` option tells **ping** to stop after sending (and receiving) one ICMP (Internet Control Message Protocol) packet.

   You should see output similar to this:

   ```
   PING target.my-company.com (192.102.207.119): 56 data bytes
   64 bytes from 192.102.207.119: icmp_seq=0 ttl=255 time=2.099 ms

   --- target.apple.com ping statistics ---
   1 packets transmitted, 1 packets received, 0% packet loss
   round-trip min/avg/max = 2.099/2.099/2.099 ms
   ```

3. If you don't already know it, determine the hardware Ethernet address of the target machine.

   Enter this command:

   ```
   $ arp -a
   ```

   The **arp** command with the `-a` option lists the static hardware Ethernet addresses of all machines your development machine has recently accessed. Make a note of the entry for your target machine's address. Because this address is stored in the hardware, you should keep track of this number for future debugging sessions (or for debugging possible kernel panics).

   For example, you should see output similar to this:

   ```
   aniil33 (1192.102.207.102) at 0:0:f:0:86:c3
   target (192.102.207.119) at 0:5:2:b0:b3:20
   dev (192.102.207.124) at 0:5:2:59:2f:20
   ```

In this example, the hardware Ethernet address to remember is `0:5:2:b0:b3:20`.

4. Remove the current, temporary, connection between your development and target machines. Use the **arp** command with your target machine's hostname. Because you'll be using the `-d` option to delete the entry for the target machine, this command must be executed as `root`, so use the **sudo** command.

   For example:

   ```
   $ sudo arp -d target.apple.com
   ```

   You should see output like this:

   ```
   target.apple.com (192.102.207.119) deleted
   ```

5. Create a new, permanent, connection between your development and target machines. Use the **arp** command with the target machine's hostname and Ethernet address. The **arp** command with the `-s` option must be executed as `root`, so use the **sudo** command.

   For example:

   ```
   $ sudo arp -s target.apple.com 0:5:2:b0:b3:20
   ```

   You can copy and paste the Ethernet address from the previous output in the Terminal window.

6. Make sure the connection was made correctly.

   Enter this command:

   ```
   $ arp -a
   ```

   Make sure the entry for your target machine now contains the word "permanent."

   You should see output similar to this:

   ```
   niil33 (192.102.207.102) at 0:0:f:0:86:c3
   target (192.102.207.119) at 0:5:2:b0:b3:20 permanent
   dev (192.102.207.124) at 0:5:2:59:2f:20
   ```

Create a Symbol File

The previous steps prepared the two machines to communicate with each other. The next steps will load the driver, prepare and transfer a symbol file, and start the debugger.

Warning: If you make a mistake in any of the following steps, or need to retrace any steps, you should begin again at this step.

There are two ways to create a symbol file. If the target machine is running, you can generate the symbol file on the target machine itself. If the target machine is crashed, you must generate the symbol file on the host machine. This section describes both methods.

Creating a Symbol File on the Target Machine

On the target machine, do the following:

1. At your option, either launch Terminal as an admin user or login as `>console` (with a blank password), then

2. If you logged in as `>console`, at the next login prompt, log in as an admin user.

   For example:

   ```
   Darwin/BSD (dev) (console)

   login: admin
   Password for admin:
   admin$
   ```

3. Copy your KEXT to `/tmp`.

   Because you will be loading your KEXT into the kernel, your KEXT must be owned by the user `root` and the group `wheel` and the folders and files of the KEXT bundle must have their permissions set so that they are not writable by any user other than the super user. If you haven't already done so, use the **sudo** command to copy the KEXT binary (`HelloIOKit.kext`) to the `/tmp` directory so you can load and unload the KEXT from there. Using **sudo** to copy the KEXT binary gives the KEXT the super user's ownership and permissions and leaves the original KEXT alone so you can revise and save it as you choose.

   Note: Every time you make changes to your KEXT and rebuild it, you need to repeat this step to copy the new version to the `/tmp` directory to load and debug it.

   For example:

   ```
   $ sudo cp -R HelloIOKit.kext /tmp
   ```

   Warning: If you use tab-completion to avoid typing all of `HelloIOKit.kext`, you'll get a "/" after the KEXT name. Be sure to delete this slash before you press Return. If you don't, the **cp** command will copy only the *contents* of the `HelloIOKit.kext` directory to `/tmp`, instead of copying the directory and its entire subtree.

4. For debugging purposes, you first use **kextload** with some options to load the driver and create a symbol file for it, but not start the driver. You'll register the driver and start it running (again using **kextload**) in a later step, "On Target Machine, Start Running the Device Driver" (page 65). The **kextload** command must be run as `root`, so you must also use the **sudo** command.

   Warning: In the KEXT development tutorials, "Hello Kernel: Creating a Kernel Extension With Xcode" (page 15) and "Hello I/O Kit: Creating a Device Driver With Xcode" (page 31), you used **kextload** to load your KEXT and start it running all in one step. Now you will break this step in two, using different options to **kextload** each time.

   Load the driver (without starting it) and create a symbol file for it.

For example, enter the command:

```
$ sudo kextload -ls /tmp HelloIOKit.kext
```

The `-l` (this is a lower-case "L") option tells **kextload** to load the KEXT's code into the kernel but not to start I/O Kit matching, which would start the KEXT running. The `-s` option tells **kextload** to create symbol files in `/tmp` for the KEXT and any of its dependencies. To ensure uniqueness the symbol files are named after each KEXT's CFBundleIdentifier, plus a `.sym` or `.dSYM` extension.

For the most part, the two symbol file formats are equivalent. For this reason, this document does not distinguish between them except where behavior differs.

Do not rename the symbol file if you are debugging with a DWARF symbol file (`.dSYM`). If you are debugging with a STABS symbol file (`.sym`), you may rename the file so long as the name still ends with `.sym`.

The symbol file contains information that GDB needs to debug your driver. You must create the symbol file for a loaded driver, then copy the file to the development machine where it will be used. Note that you must recreate the symbol file again if you unload (and reload) the driver. This is because the KEXT may be loaded at a different address each time and the symbol files contain the actual virtual address of each symbol.

5. Optionally, you can rename the symbol file.

   For example:

   ```
   $ mv /tmp/com.MyTutorial.driver.HelloIOKit.sym /tmp/HelloIOKit.sym
   ```

Creating a Symbol File on the Host Machine

To create a symbol file on the host machine, you must have a copy of the kernel extension on the host machine. The host machine must have the same architecture as the target machine (for example, two Intel machines). You must also have the kernel debug kit that correspondes with the version of Mac OS X installed on the target machine.

1. Create the symbol file.

   ```
   sudo kextload -n -k /Volumes/KernelDebugKit/mach_kernel -s /tmp/syms/
   HelloIOKit.kext
   ```

   Adjust the path to the kernel debug kit and your kernel extension as appropriate.

2. The `kextload` tool will now prompt you for the load addresses of your KEXT and any KEXT upon which it depends. Enter the addresses as requested.

   If the target machine is still running, you can obtain a full list by running `kextstat` (with no arguments) on the target machine. See the manual page for `kextstat(8)` for more information.

   In the case of a non-graphical kernel panic, you should find a list of these address on screen.

   In the case of a graphical kernel panic or a hang, you can get this information by connecting to the target machine using GDB as described in "Postmortem Debugging" (page 67).

   When you finish, you should see a symbol file in `/tmp/syms`.

3. There's no step 3!

On Development Machine, Import the Symbol File

On the development machine, import the symbol file (`.sym` or `.dSYM`) from the target machine, for use with GDB. You can do this by copying the file to a keychain drive or an external hard drive, or by using the `ftp(1)` (file transfer protocol) or `scp(1)` (secure copy) command. For security reasons, the `scp` command is recommended over the `ftp` command.

If you are debugging a driver built with DWARF symbols (a `.dSYM` file), you must also copy the kernel extension itself. The easiest way to do this is to create a ZIP archive of the KEXT in Finder or use tar on the command line.

For example, the following two commands create an archive of a KEXT and extract that archive, respectively:

```
tar -czf mykext.tar.gz mykext.kext # create archive
tar -xzf mykext.tar.gz # extract archive
```

To use the `scp` command, you need an account on the target machine and you need to have Remote Login enabled in the Sharing System Preferences pane. Then, issue the following commands:

```
cd /path/to/store/the/symbol/file/
scp username@target.apple.com:/path/to/remote/file.sym .
```

Change the remote username, host name, and paths as appropriate. The scp command then asks you for the remote password. Type the password, and a few seconds later, you should have a copy of the symbol file in the destination directory on your development machine.

To use the `ftp` command, you need an account on the target machine and you need to enable FTP in the Sharing System Preferences pane. A sample ftp session is shown below. Enter the ftp commands as shown.

Warning: Be sure to use the actual name or IP address of the target machine; in this example it is `target.apple.com`. Be sure to log in with your actual account name; this example uses `admin`.

```
$ ftp target.apple.com
Connected to target.apple.com.
220 target.apple.com FTP server (Version 6.00) ready.
Name (dev:admin): admin
331 Password required for admin.
Password:

230- Welcome to Darwin!
230 User admin logged in.
Remote system type is BSD.
ftp> bin
200 Type set to I.
ftp> get /tmp/com.MyTutorial.driver.HelloIOKit.sym
local: /tmp/com.MyTutorial.driver.HelloIOKit.sym remote:
/tmp/com.MyTutorial.driver.HelloIOKit.sym
```

```
200 PORT command successful.
150 Opening BINARY mode data connection for
'/tmp/com.MyTutorial.driver.HelloIOKit.sym'
226 Transfer complete.
180356 bytes received in 0.15 seconds (1.15 MB/s)
ftp> bye
```

Note that you must recreate (and import) the symbol file again if you unload (and reload) the driver on the target machine. You should also note that any files in the `/tmp` directory are temporary and will be deleted whenever Darwin/Mac OS X is restarted.

On Development Machine, Start GDB

Now you can start GDB. On the development machine, do the following from the Terminal application:

1. Start the debugger on the kernel.

   Enter this command:

   ```
   $ gdb -arch ppc /mach_kernel
   ```

   Substitute `-arch i386` if your debug target is an Intel-based Macintosh. You should see output like this:

   ```
   GNU gdb 4.18-20000320 (Apple version gdb-172)
   Copyright 1998 Free Software Foundation, Inc.
   ...
   (gdb)
   ```

2. Load the symbol file you created and transferred.

   If you are debugging a kernel extension compiled with DWARF symbols, at the GDB prompt, use the **add-kext** command with the name of the KEXT.

   Note: When debugging with DWARF symbols (a file ending in `.dSYM`), you must place the debug symbols file in the same directory as the KEXT.

   For example:

   ```
   (gdb) add-kext /tmp/com.MyTutorial.driver.HelloIOKit.kext
   ```

   If you are debugging a kernel extension compiled with STABS symbols, at the GDB prompt, use the **add-symbol-file** command with the name of the symbol file you've created.

   For example:

   ```
   (gdb) add-symbol-file /tmp/com.MyTutorial.driver.HelloIOKit.sym
   ```

3. Tell the debugger that you're remotely debugging a device driver.

   At the GDB prompt, enter this command:

   ```
   (gdb) target remote-kdp
   ```

On Target Machine, Break into Kernel Debugging Mode

Before you can connect GDB to the target machine, you must break into kernel debugging mode. There are four ways to break into kernel debugging mode.

- You can force a break into the debugger by briefly pressing the power button on the machine. Because you enabled kernel debugging in "On Target Machine, Enable Kernel Debugging" (page 56), briefly pressing the power button sends a nonmaskable interrupt (NMI) instead of sleeping or waking the machine. You will use this method in this tutorial.
- On some machines, you can force a break into the debugger using the programmer's button (check your computer manual for its location) to send an NMI.
- If your keyboard contains a power button, you can force a break into the debugger from the keyboard.

  To use this method, you need to hold a combination of keys for three seconds. To estimate three seconds, try counting aloud "one thousand one, one thousand two, one thousand three".

  On a USB keyboard, or if you're using a PowerBook or iBook , hold down the *Command* (or Apple) key and the *Power* button.

  If you're running Mac OS X version 10.4 or later, hold down the following five keys: *Command*, *Option*, *Control*, *Shift*, and *Escape*.

  On an ADB keyboard, hold down *Control* and *Power*.

  Be sure to hold down the keys for only three seconds. You could reboot the machine if you hold the keys down for too long.
- In Mac OS X 10.4 and later, if you are using a USB keyboard, you can use the key combination Command-Option (Alt)-Control-Shift-Escape.
- You can put a call to `PE_enter_debugger` into your code; your driver will enter the debugger when it reaches this point. For example, you could add this line to your driver's `init` method:

  ```
  PE_enter_debugger("debug")
  ```

Note: Unless you're in Console mode, it may be difficult to tell when the kernel stops and the target machine is in kernel debugging mode. One way to tell is to enable the seconds display in the menu bar clock. When the seconds stop counting, the kernel is stopped. To enable the seconds display, open System Preferences and click Date & Time. Select the Clock tab and check the box next to Display the time with seconds.

On the target machine, do the following:

1. Break into the debugger. Briefly press the power button on the machine.
2. The kernel stops and the machine waits for a remote debugger connection.

   If you're in Console mode, you'll see the following message on the screen:

   ```
   ethernet MAC address: 0:5:2:b0:b3:20
   ip address: 192.102.207.119
   ```

3. If the seconds don't stop counting (or if you don't see the "Waiting..." message in Console mode), briefly press the power button again.
4. Immediately move to the development machine and attach to the target machine from GDB.

On Development Machine, Attach to the Target Machine

Now you can tell GDB to attach to the target machine. On the development machine, do the following:

1. Attach to the target machine. At the GDB prompt, use the `attach` or `kdp-reattach` macro with the target machine's name or IP address.

   For example:

   ```
   (gdb) attach target.apple.com
   or
   (gdb) kdp-reattach target.apple.com
   ```

2. The target machine prints:

   ```
   Connected to remote debugger.
   ```

   Note that the target machine is currently unable to accept keyboard input.

On Development Machine, Set Breakpoints in GDB

This is the best place to set breakpoints in your code, before you actually run the driver on the target machine. For this tutorial, set a breakpoint at the `start` method.

1. In GDB, set a breakpoint in the `start` method.

   For example:

   ```
   (gdb) break 'com_MyTutorial_driver_HelloIOKit::start(IOService
   *)'
   Breakpoint 1 at 0x53d93d0: file HelloIOKit.cpp, line 54.
   ```

   Hint: You don't need to type the entire class name for the breakpoint; GDB can do name completion. Type a single quote, `'`, then the first part of the name, then press the tab key. If GDB does not complete the name, you may need to type a few more characters to allow it to choose an unambiguous match. For example, type

   ```
   (gdb) break 'com_<TAB>
   ```

   When you press the tab key, GDB will complete the name as far as it can, unambiguously. If it cannot complete the entire name of the method, you will need to give it more clues. For example, all of the methods in this tutorial begin with `com_MyTutorial_driver_HelloIOKit`. When GDB has completed that much, it will stop. Then you can type

```
::start<TAB>
```

GDB will finish the entry. After GDB completes the method name (with a final closing single quote), be sure to press return.

2. Allow the target machine to continue. On the development machine, enter the **continue** command at the GDB prompt. For example:

   ```
   (gdb) continue
   ```

3. The target machine is again able to accept keyboard input. Now you can run the driver.

On Target Machine, Start Running the Device Driver

Now that you have GDB attached and breakpoints set, you can start the driver running. On the target machine, do the following.

1. If you are not already there, move to the `/tmp` directory, using the **cd** command. For example

   ```
   $ cd /tmp
   ```

2. Start the driver running, using the **kextload** command. Recall that you previously loaded the driver using **kextload** with the `-l` option, which does not start I/O Kit matching for the driver, meaning that the code doesn't start running. You must now use **kextload** with the `-m` option to finish the process and run the driver. For example:

   ```
   $ sudo kextload -m HelloIOKit.kext
   ```

3. The driver executes until it hits a breakpoint.

On Development Machine, Control the Driver With GDB

Now that GDB is running on the development machine, the target machine is attached to the debugger, and the driver is running, you can actually start debugging!

The driver executes on the target machine until it hits a breakpoint. When this happens, GDB will print some status on the development machine. For example:

```
Breakpoint 1, com_MyTutorial_driver_HelloIOKit::start (this=0xcdcfc0,
dict=0xbcfe40) at HelloIOKit.cpp:54
54 HelloIOKit.cpp: No such file or directory.
Current language: auto; currently c++
```

Now you can debug your driver (almost) as you would any other executable. However, because driver debugging happens at such a low level, you won't be able to take advantage of all of GDB’s features. For example:

- You can't call a function or method in your driver.

- You can't debug interrupt routines.
- Kernel debug sessions don't last indefinitely. Because you must halt the target machine's kernel to use GDB, internal inconsistencies may appear that will cause the target kernel to panic or hang, forcing you to reboot the target machine.

For help with GDB, use its **help** command. Most GDB commands have shortcuts; for example, you can type `c` instead of `continue`.

Here are a few things you can do. From GDB, try the following:

- Single step through some code (**s** command).
- List a method's source code (**l** command).
- Print the contents of a variable (**p** command). For example

  ```
  (gdb) p res
  ```

Stop the Debugger

When you've finished debugging, you should stop the debugger and then unload the driver. Do the following:

1. On the target machine, break into the kernel debugger. Briefly press the power button as described in "On Target Machine, Break into Kernel Debugging Mode" (page 63).

   If you're in Console mode, you'll see the following message on the screen:

   ```
   Debugger(Programmer Key)
   ```

2. On the development machine, enter this command at the GDB prompt:

   ```
   (gdb) quit
   ```

3. Answer yes to the prompt that appears:

   ```
   The program is
   running. Exit anyway? (y or n)
   ```

4. The target machine prints

   ```
   Remote debugger
   disconnected
   ```

The debugging session ends.

Note: If your target machine is running Mac OS X Server or contains a PMU (for example, a PowerBook G4 or an early G5 desktop computer), you may find that it reboots during kernel debugging or shuts down when you exit kernel debugging mode. These problems have different causes.

If a breakpoint or explicit NMI causes kernel debugger entry in the middle of a PMU transaction, the PMU transaction times out after a few milliseconds. The result is that the PMU driver and the PMU hardware are now out of sync. The resulting confusion can cause the computer to shut down or stall for a period of time after you leave the debugger.

If you experience this, you may find that forcing the PMU driver to operate in polled mode fixes the problem. To do this, set the NVRAM variable `pmuflags` to `1` when you enable kernel debugging (see "On Target Machine, Enable Kernel Debugging" (page 56)), as shown below:

```
$ sudo nvram boot-args="pmuflags=1"
```

You can set the `pmuflags` variable separately, as shown above, or you can set it at the same time you set the `debug` variable, as shown below:

```
$ sudo nvram boot-args="debug=0x14e pmuflags=1"
```

In Mac OS X Server, the watchdog daemon enables a hardware watchdog timer. That watchdog timer is designed to reboot the server if it crashes or hangs. Because the daemon that tickles this timer every 30 seconds cannot run while you are debugging, debugging is indistinguishable from a crash or hang, and thus, the watchdog timer assumes that the computer has crashed and reboots it.

To disable the watchdog timer, you must terminate the watchdog daemon in such a way that it exits cleanly and disables the hardware watchdog timer. To do this, issue the following command prior to entering the debugger:

```
$ sudo killall -TERM watchdogtimerd
```

For more information, see the manual page for `watchdogtimerd(8)`.

On Target Machine, Unload the Driver

On the target machine, unload the driver. Use the **kextunload** command. This command unloads your driver by running its termination (stop) function. For example:

```
$ sudo kextunload HelloIOKit.kext
IOCatalogueTerminate(Module com.MyTutorial.driver.HelloIOKit) [0]
done.
```

Postmortem Debugging

The previous sections assume that the target machine is accessible and has not crashed. In some cases, you may need to debug a computer that has already crashed or frozen, and thus, running kextload to generate a symbol file is not possible.

To do this, you first need a development machine of the same architecture as the target machine (Intel if your target machine is Intel, for example). Next, you need a copy of the kernel debug kit for the version of Mac OS X that is running on the target machine.

1. If the target machine is frozen (as opposed to a kernel panic), drop into the debugger on the target machine as described in the section "On Target Machine, Break into Kernel Debugging Mode" (page 63).

2. Start the debugger on the kernel.

   Enter this command:

   ```
   $ gdb -arch ppc /mach_kernel
   ```

   Substitute `-arch i386` if your debug target is an Intel-based Macintosh. You should see output like this:

   ```
   GNU gdb 4.18-20000320 (Apple version gdb-172)
   Copyright 1998 Free Software Foundation, Inc.
   ...
   (gdb)
   ```

3. Tell the debugger that you're remotely debugging a device driver.

   At the GDB prompt, enter this command:

   ```
   (gdb) target remote-kdp
   ```

4. Attach to the remote machine with the following command:

   ```
   kdp-reattach 192.168.1.47
   ```

   Substitute the actual IP number of the remote machine.

5. Once you have connected to the target, you must load the GDB macros from the kernel debug kit by issuing the following command:

   ```
   source /Volumes/KernelDebugKit/kgmacros
   ```

6. Assuming the target machine is configured to display panic text, you already have a list of KEXTs involved in the panic. If not, you can get this information by typing the following command:

   ```
   paniclog
   ```

   If the target machine is not in a panicked state, you can obtain a list of KEXT load addresses by typing one of two commands. If you are running Mac OS X v10.4 or earlier, use the following command:

   ```
   showallkmods
   ```

   Otherwise, use the following command:

   ```
   showallkexts
   ```

   The debugger will then display a list of KEXTs and their load addresses.

7. Once you have a list of KEXT load addresses, you must create symbol files for the relevant extension(s). For each KEXT that you care about, issue the following command in a new terminal window on the development machine:

   ```
   sudo kextload -n -k /Volumes/KernelDebugKit/mach_kernel -s  /tmp
   /path/to/the/desired.kext
   ```

After you have generated symbols for all relevant KEXTs, you can continue following the steps in this tutorial beginning with the section "On Development Machine, Import the Symbol File" (page 61), skipping any steps you have already performed in this section.

Where to Go Next

Congratulations! You've now learned how to set up a two-machine debugging environment to debug a driver (or another type of KEXT) using GDB.

If you're interested, you can use the `man(1)` command to read the manual pages for `kextload(8)`, `kextstat(8)`, and `kextunload(8)`. In particular, you may want to read about the many ways to use **kextload** to get debug symbols for all types of KEXTs, including I/O Kit drivers. For example, from a Terminal window, enter the following command:

```
$ man kextload
```

You may also want to familiarize yourself with GDB. Further information is available in Tools Documentation.

Packaging Your KEXT for Distribution and Installation

This tutorial describes how to package a kernel extension (KEXT) for distribution and installation on Mac OS X. The KEXT can be any type: a device driver, file system stack, or Network Kernel Extension (NKE). The tutorial assumes that you are working in a Mac OS X development environment.

Important: The information in this document reflects versions of PackageMaker available in Mac OS X v10.3 and above.

Why Should You Create a Package?

As built, a kernel extension is not well-suited for distribution and installation. For example, some of the files in the bundle could be lost during distribution over the Web. The hierarchical tree format of a bundle leaves its contents open to accidental damage by mishandling. For example, if a KEXT were transferred to a DOS ("8.3") file system, filenames within the bundle directory could be truncated.

In addition, a KEXT does not normally contain the sort of useful information users expect when they install software, such as licensing restrictions, descriptive information, or default installation location. Before you distribute a KEXT for installation under Mac OS X, you should prepare it by creating a *package*.

Anatomy of a Package

The Mac OS X package format was created to be used with the Installer application. Installer takes care of providing users with many useful features such as descriptive information and default installation location.

Like KEXTs, packages are also implemented as bundles, folders that the Finder treats as single files. Although a package appears as a single entity from a Desktop Finder window, you can Control-click the package icon and choose Show Package Contents to view the files in the package. You can also use the Terminal application to enter a package directory and view the component files. The files that make up a package are named with suffixes that indicate the type of information stored in the file.

A package provides everything the Mac OS X Installer needs to install your software. A basic package contains:

- A Bill of Materials file—this is a file in binary (non-text) format that describes the contents of the archive file. PackageMaker names this file `Archive.bom`.
- An `Info.plist` file—this is a plist-formatted file that contains the information you entered in the PackageMaker application when you created the package project.
- An archive file—this is an archive of the complete set of files that will be installed. PackageMaker names this file `Archive.pax[.gz]`. If you choose to compress the files, the file name will include the `.gz` suffix. For this tutorial, the contents of your KEXT will comprise the `Archive.pax[.gz]` file.
- A package-description file—this is a plist-formatted file that contains localizable information such as the package title. PackageMaker names this file `Description.plist`.
- Resources (optional)—packages may contain additional (optional) supplementary resources, such as Read Me files, licensing information, pre- or postinstall scripts, and installation or volume check tools. These are package resources, used (or run) at installation time; they are not part of the KEXT's installation but are saved to disk as part of the package's receipt (located in `/Library/Receipts`). Resources can include any number of localized language projects (`.lproj` folders) that can contain localized versions of the Read Me, welcome, and license files, as well as a localized background image.

Because packages are implemented as bundles, their contents are also subject to possible modification during handling. To prepare a package for distribution, you should archive and (possibly) compress it into a form that can be distributed safely without changing the contents.

Preparation

Before you begin this tutorial, make sure you have met the following requirements.

1. Be sure you understand how to create a KEXT and what makes up a KEXT.

   You should have completed and understood the material presented in a previous tutorial, "Hello Kernel: Creating a Kernel Extension With Xcode" (page 15), before beginning this one. That tutorial demonstrates how to create a simple kernel extension project and explains the contents of a KEXT.

2. If your KEXT is a device driver, be sure you understand the differences between a device driver and other KEXTs.

   You should have completed and understood the material presented in "Hello I/O Kit: Creating a Device Driver With Xcode" (page 31). That tutorial demonstrates how to create a simple device driver (I/O Kit KEXT) project and explains the differences between device drivers and other kernel extensions.

3. Build a KEXT.

   Before you begin this tutorial, you will need to build a KEXT that you can package and install. It doesn't matter whether you plan to package a device driver or another type of KEXT.

4. This tutorial assumes you are logging in to Mac OS X as the user `admin` and that this user has administrative privileges on your computer. This tutorial also assumes that your KEXT project was named `Hello`. In the examples below, the project files can be found at the location

`/Users/admin/Projects/Hello/build`. The name of the KEXT is `Hello.kext`. If you are using another account, project name, or project location, substitute the correct information in the examples below.

Roadmap

Here's how you'll package your KEXT:

1. "Locate Your KEXT" (page 73)
2. "Create a Distribution Directory" (page 74)
3. "Gather Any Package Resources" (page 75)
4. "Create a Package with PackageMaker" (page 77)
5. "Examine the Package and Files" (page 82)
6. "Test Installing the Package" (page 83)

You'll use the Terminal application to type the commands to locate your KEXT and create a distribution tree. You'll use the PackageMaker application to build your package. Then you will test the package with the Mac OS X Installer application.

Locate Your KEXT

If you have not yet done so, you will need to create, build, and test a KEXT, either a device driver or another type of KEXT. If you do not already have a KEXT to package, follow the instructions in one of the previous tutorials, "Hello Kernel: Creating a Kernel Extension With Xcode" (page 15) or "Hello I/O Kit: Creating a Device Driver With Xcode" (page 31). It is not necessary (for this tutorial) that your KEXT actually do anything, only that it exists.

Locate your KEXT. Use the Terminal application to move to the directory that contains your KEXT.

1. Start the Terminal application. From a Desktop Finder window, locate and launch the Terminal application, found at `/Applications/Utilities/Terminal`.
2. Choose New from the Shell menu to start a new shell window.
3. In the Terminal window, move to the directory that contains your KEXT. Use the **cd** command to move to the appropriate directory. For example:

   ```
   % cd /Users/admin/Projects/Hello/build
   ```

   This directory contains your KEXT. You can use the **ls** command to view the contents of this directory. For example:

   ```
   % ls
   ```

```
Hello.build
Hello.kext
```

Your KEXT should have a name that ends with the suffix **.kext**.

Warning: For purposes of packaging, distribution, and installation, the filename of the KEXT (apart from the suffix) does not matter. However, the name of the KEXT binary and its class (if a driver), as stored in the KEXT's property list, should be unique. These should use the recommended "reverse-DNS" naming convention, such as `com.MySoftwareCompany.iokit.HelloIOKit`.

From a Desktop Finder window, a KEXT appears as a single file (look for it from the Desktop if you like). From the Terminal application, however, a KEXT appears as a directory.

Create a Distribution Directory

When your KEXT is installed, it will be installed into the Extensions folder, at `/System/Library/Extensions` under Mac OS X.

Because kernel extensions contain code and data that are loaded into the kernel, the most protected environment in the operating system, their file ownership and permissions must be set to prevent unauthorized tampering. All KEXT bundles (all files and folders in the KEXT, including the KEXT binary) must be owned by the user `root` and the group `wheel`. In addition, the folders and files of the KEXT bundle must have their permissions set so that they are not writable by any user other than the super user.

For development purposes, however, you can make a `root`-owned copy of your KEXT binary by using the **sudo** command to copy the KEXT binary (`Hello.kext`) to the `/tmp` directory and creating the distribution directory there. Using the **sudo** command to copy the KEXT binary to `/tmp` gives the KEXT the super user's ownership and permissions and leaves the original KEXT alone so you can revise and save it as you choose.

Note: You should not change the ownership and permissions of any of your KEXT files in your own directory, because you will no longer be able to save them after working on them. Thus, every time you make changes to your KEXT and rebuild it, you need to repeat the following steps to copy the new version to the `/tmp` directory and repackage it.

To create the package, you will first make a distribution folder, then copy your KEXT into it. When the package is installed (unpacked), Installer will place the KEXT at the correct point in the user's file system.

1. At the prompt, you'll use the **cp -R** command with the **sudo** command to copy your KEXT to `/tmp`. The **cp** command copies files from one place to another and the *-R* option tells **cp** to copy a directory and its entire subtree (like `Hello.kext`). When prompted for a password, enter your `admin` password. (Note that nothing is displayed as you type the password.)

   For example:

```
% sudo cp -R Hello.kext /tmp
Password:
```

Check the ownership and permissions of your driver by moving to the `/tmp` directory and using the **ls -l** command:

```
% cd /tmp
% ls -l
drwxr-xr-x 3 root wheel 102 Oct 20 10:30 Hello.kext
```

The *-l* makes the **ls** command display extra information about the files in a directory.

2. Now use the **mkdir** command with the **sudo** command to create the distribution directory. For example:

   ```
   % sudo mkdir dstroot
   ```

   This creates a directory folder, `dstroot` (distribution root), in the `/tmp` directory.

3. Copy your KEXT into the new `dstroot` folder using the **cp** command with the `-R` option. For example:

   ```
   % sudo cp -R Hello.kext dstroot
   ```

4. List the contents of the new directory hierarchy. Use the **find** command. For example:

   ```
   % find dstroot
   ```

   You should see output like this (although the name of your KEXT may differ):

   ```
   dstroot
   dstroot/Hello.kext
   dstroot/Hello.kext/Contents
   dstroot/Hello.kext/Contents/Info.plist
   dstroot/Hello.kext/Contents/MacOS
   dstroot/Hello.kext/Contents/MacOS/Hello
   dstroot/Hello.kext/Contents/Resources
   dstroot/Hello.kext/Contents/Resources/English.lproj
   dstroot/Hello.kext/Contents/Resources/English.lproj/InfoPlist.strings
   dstroot/Hello.kext/Contents/Resources/...
   ```

Gather Any Package Resources

As described in "Anatomy of a Package" (page 71), packages may contain additional (optional) supplementary resources used by the package itself. These are used (or run) at installation time but will not be installed on disk along with the software.

There are certain special filenames that Installer will recognize and display automatically if they are present in a package. These are: a Read Me file, a software license file, and a welcome file. These files can be in any of several forms: text (.txt), HTML (.html) or Rich Text Format (.rtf). In this tutorial, you will create three files: `ReadMe.rtf`, `License.rtf`, and `Welcome.rtf`.

The supplementary resources will not be installed along with your KEXT, so they should not be placed in the distribution directory you created in "Create a Distribution Directory" (page 74). Instead, use the **cd** command to go back to the `Projects` directory containing the original copy of your KEXT:

```
% cd /Users/admin/Projects
```

and create a resources directory there:

```
% mkdir MyResources
```

You can create your supplementary Installer resources elsewhere and copy them to the resources directory, or you can create them in the directory. In this tutorial, you will create three files, `ReadMe.rtf`, `License.rtf`, and `Welcome.rtf` in the resources directory. Move to your resources directory using the **cd** command. For example:

```
% cd MyResources
```

Create a Read Me File

If a Read Me file is present in a package, the Mac OS X Installer will display the contents of the file. A user dismisses the Read Me by clicking Continue. The presence of a `ReadMe.rtf` file is all that is needed to get this feature. Installer will automatically add a "Read Me" line to the bullet list presented in the left column of the application. For an example of a Read Me file displayed during installation, see Figure 6 (page 84).

1. Start the TextEdit application. From a Desktop Finder window, locate and launch the TextEdit application, located at `/Applications/TextEdit`.

2. Enter the text of your Read Me file. Generally, a Read Me file should describe the contents of your package, version information, and any additional information a user might need to see. For example, a Read Me file could provide contact information, special hardware instructions, any known incompatibilities with other extensions, and so forth.

3. Save your Read Me file. Use the name `ReadMe.rtf`. As the location (Where), choose the `MyResources` directory you created in "Gather Any Package Resources" (page 75).

4. Close the Read Me file.

Create a Software License File

If a license file is present in a package, Installer will display the contents of the file. A user dismisses the license pane by clicking Continue. Installer then displays a dialog that requires the user to Agree to the license terms in order to proceed with installation.

The presence of a `License.rtf` file is all that is needed to get this feature. Installer will automatically add a "License" line to the bullet list presented in the Left column of the application.

1. Use the TextEdit application. Choose New from the File menu to start a new document.
2. Enter the text of your license file. Generally, a license file should describe the terms of use for your package, any legal disclaimers, or any pre-release software warnings.
3. Save your license file. Use the name `License.rtf`. As the location (Where), choose the `MyResources` directory you created in "Gather Any Package Resources" (page 75).
4. Close the license file.

Create a Welcome File

If a welcome file is present in a package, Installer will display the contents of the file. After the user authenticates (if required), the welcome message is the first thing displayed when a package is opened. The presence of a `Welcome.rtf` file is all that is needed to get this feature.

1. Use the TextEdit application. Choose New from the File menu to start a new document.
2. Enter the text of your welcome file. Generally, a welcome file is a brief introduction to the software being installed, and may contain the name of the software, its creator, and a very brief description of the software. More extensive description belongs in the Read Me file.

   The text of the welcome file may not be larger than the Installer window it is displayed in because the welcome pane will not allow the user to scroll to see more text.
3. Save your welcome file. Use the name `Welcome.rtf`. As the location (Where), choose the `MyResources` directory you created in "Gather Any Package Resources" (page 75).
4. Close the welcome file.

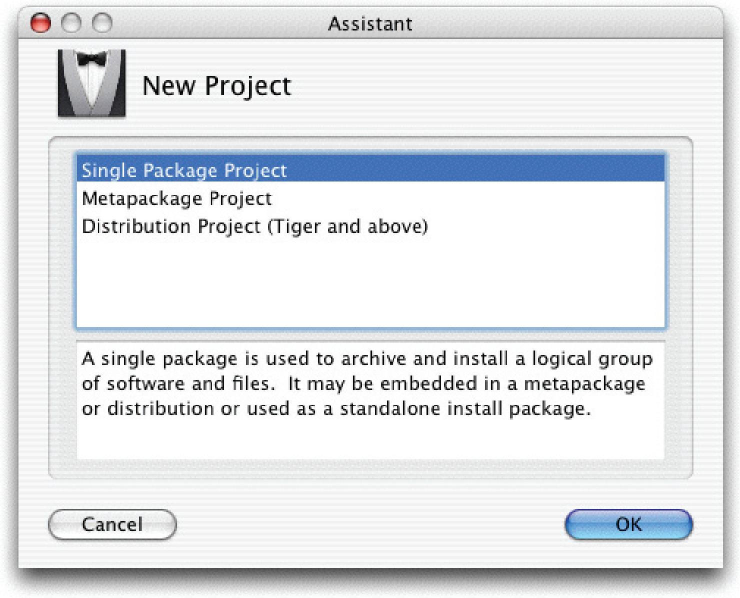
Create a Package with PackageMaker

Now you can use PackageMaker to create a package project and build from it an installable package, which includes all of the files in your distribution and resources directories.

1. Start the PackageMaker application. From a Desktop Finder window, locate and launch the PackageMaker application, located at `/Developer/Applications/Utilities`.

2. With PackageMaker, you can create simple packages, metapackages, and distributions (for more information on these types of installations, see *Software Delivery Guide*). For this tutorial, select Simple Package Project in the Assistant window and click OK, as shown in Figure 1.

Figure 1 Choose Simple Package Project in the PackageMaker Assistant window

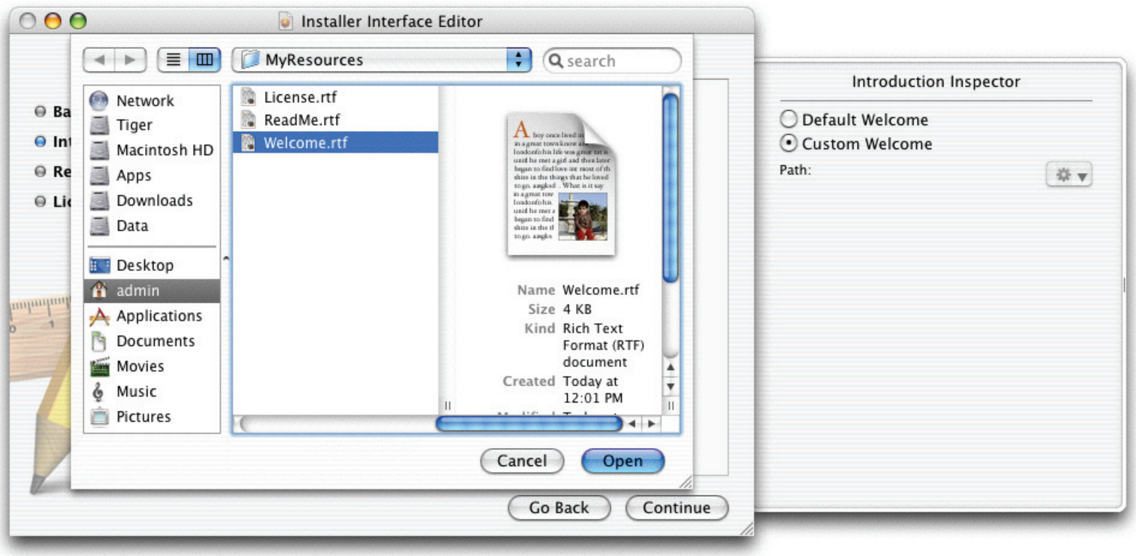


A single package is used to archive and install a logical group of software and files. It may be embedded in a metapackage or distribution or used as a standalone install package.

3. After you click OK to choose the package project type, the Assistant window disappears and is replaced by the PackageMaker window. Click the Show Installer Interface Editor button (if you don't see this button, be sure the Installer Interface tab is selected in the PackageMaker window). This displays a separate window in which you can customize the background image Installer displays and add to the package your welcome, Read Me, and license files (for this tutorial, you won't customize the background image).

   - First, add to the package the welcome file you created in "Create a Welcome File" (page 77). Click the Introduction radio button on the left side of the Installer Interface Editor window. You can either drag your welcome file into the center pane or you can select Custom Welcome in the Introduction Inspector drawer on the right side of the window. If you select Custom Welcome in the Introduction Inspector, you'll see a dialog drop down in which you navigate to your welcome file and then double-click it or select it and click Open, as shown in Figure 2.

Figure 2 Use the Installer Interface Editor to add the welcome file to the package



- Next, perform similar steps to add the Read Me file you created in "Create a Read Me File" (page 76): Click the Read Me radio button on the left side of the Installer Interface Editor window. You can either drag your Read Me file into the center pane or you can select Custom Read Me in the Read Me Inspector drawer on the right side of the window. If you select Custom Read Me, a dialog drops down in which you navigate to your Read Me file and then double-click it or select it and click Open.
- Finally, perform similar steps to add the license file you created in "Create a Software License File" (page 76): Click the License radio button on the left side of the Installer Interface Editor window. You can either drag your license file into the center pane or you can select Custom License in the License Inspector drawer on the right side of the window. If you select Custom License, a dialog drops down in which you navigate to your license file and then double-click it or select it and click Open.

4. Before you close the Installer Interface Editor window, save the changes you've made to the package project. In the menu bar, choose File and click Save. PackageMaker will ask you to name this project and choose a location in which to save it. For this tutorial, choose the name `Hello.pmproj`.

   As you provide additional information in the following steps, it's a good idea to save frequently. Also, if you want to make changes to any of this information, you can edit this project later and build a new package from it.

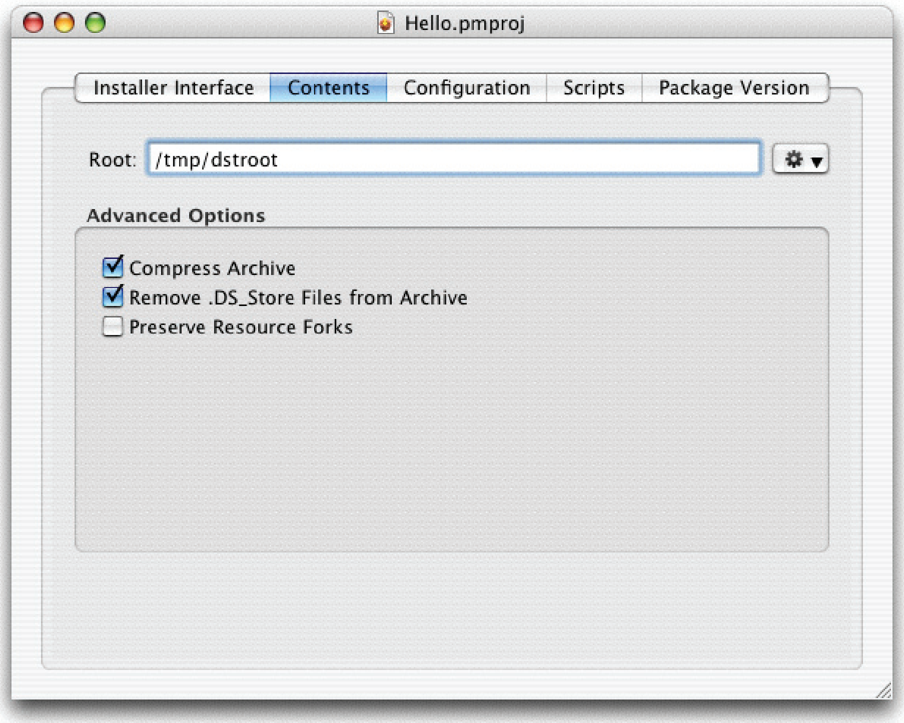
   Close the Installer Interface Editor window.

   Note: You do not need to make any changes to the Installer Interface pane in the main PackageMaker window, because the package you build in this tutorial is not contained in an old-style metapackage. For more information about metapackages, see *Software Delivery Guide*.

5. In the PackageMaker window, click Contents.

- For Root, type or choose the path to the copy of your KEXT in the `/tmp` directory. For example, `/tmp/dstroot`.
- For this tutorial, you can leave the checkboxes under Advanced Options in their default state. Specifically, you can leave the Compress Archive and the Remove .DS_Store Files from Archive checkboxes checked and the Preserve Resource Forks checkbox unchecked. Figure 3 shows the Contents pane of the PackageMaker window after you've completed this step. If you're interested in learning more about these options and how they can affect the packaging and installation of your software, see *Software Delivery Guide*.

Figure 3 Provide the location of your software



Root: /tmp/dstroot

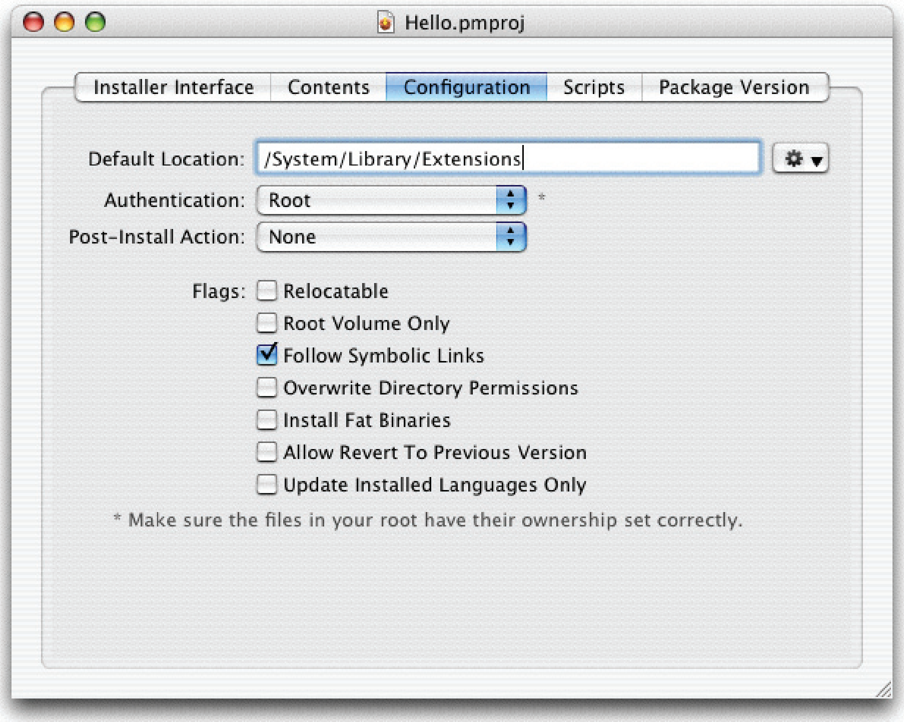
[✓] Compress Archive
[✓] Remove .DS_Store Files from Archive
[ ] Preserve Resource Forks

6. Click Configuration.
   - For Default Location, type or choose the path to the location in which you want your KEXT installed. For example, `/System/Library/Extensions`.
   - Select Root authentication for the authentication requirement. This allows one of Installer's tools to be run as `root`. If the user installing your KEXT is logged in as the `root` user, Installer will not request any authorization. If the user is logged in as the `admin` user, Installer will display a dialog that requests the user to supply his or her `admin` password. Only users with `root` or `admin` privileges should install KEXTs.
   - Select the appropriate restart action. For this tutorial, select None.
   - Set the appropriate flags. For this tutorial, check only the Follow Symbolic Links checkbox. This ensures that the user's symbolic links will be preserved.

Important: Be sure to leave the Overwrite Directory Permissions checkbox unchecked. This prevents the possibility of causing Installer to incorrectly change the permissions of the `/System/Library/Extensions` folder.

Figure 4 shows the Configuration pane of the PackageMaker window after you've completed this step.

Figure 4 Describe the installation location and actions



Default Location: /System/Library/Extensions
Authentication: Root *
Post-Install Action: None

Flags:
- [ ] Relocatable
- [ ] Root Volume Only
- [✓] Follow Symbolic Links
- [ ] Overwrite Directory Permissions
- [ ] Install Fat Binaries
- [ ] Allow Revert To Previous Version
- [ ] Update Installed Languages Only

* Make sure the files in your root have their ownership set correctly.

7. Click Package Version.

   - For Identifier, type your reverse-DNS name; for example, `com.MyTutorial.Hello`.
   - The Get Info String field contains a string the Finder displays in the Get Info window for the package. Choose a name that describes your KEXT, such as `Hello v1.0`.
   - The Version field contains the version string the Finder displays in column view. You can use the same version number you used in the Get Info string or a different one, depending on how you structure your development environment. For this tutorial, use the same version number (`1.0`).

When you are finished providing the package information, click File in the menu bar and select Save. If you haven't saved earlier, PackageMaker will ask you what name to save the package project under and where to store it.

Now PackageMaker can build an installable package from the information in the project you've created. In the menu bar, choose Project and click Build.

PackageMaker will ask you what name to save the package under and where to store it:

1. In the Save As field, type in a name for your package. The best name to choose is one that incorporates the name of your KEXT. For example, if your KEXT is named `Hello.kext`, name your package `Hello.pkg` (note that PackageMaker automatically supplies the `.pkg` extension in the Save As field).

2. Choose a location. The default location is your home folder, but you may want to create a new folder called Packages to use for all packages you create.

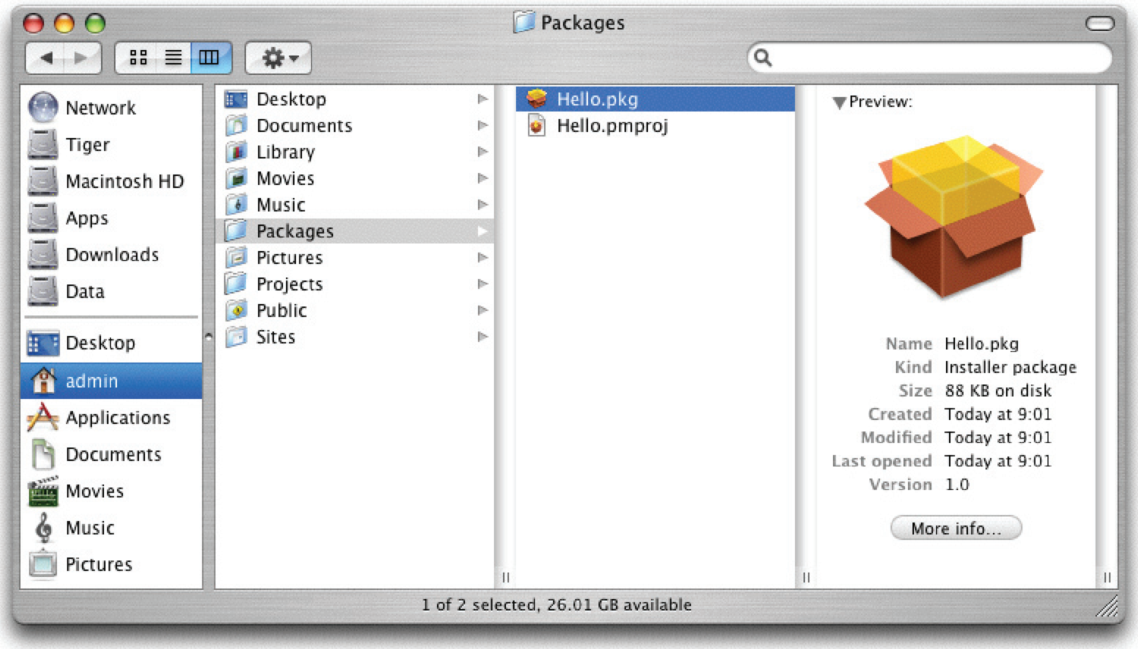
3. When you have chosen a name and location, click Save.

Examine the Package and Files

You can now quit PackageMaker and examine your new package.

1. From a Desktop Finder window, navigate to the location where you told PackageMaker to save your package. For example, if you named the package Hello.pkg and told PackageMaker to store it in the Packages folder in your home folder, move to the Packages folder as shown in Figure 5.

Figure 5 Locate your package on the Desktop



Name Hello.pkg
Kind Installer package
Size 88 KB on disk
Created Today at 9:01
Modified Today at 9:01
Last opened Today at 9:01
Version 1.0

1 of 2 selected, 26.01 GB available

The file `Hello.pmproj` is the project that contains the information you provided in "Create a Package with PackageMaker" (page 77).

2. From a shell window in the Terminal application, move to the directory that contains your package. For example:

   ```
   % cd /Users/admin/Packages
   ```

   From a Desktop Finder window, a package appears as a single file. From the Terminal application, however, a package appears as a directory.

   List the contents of your package directory using the **find** command. Some of the files you'll see are listed below:

   ```
   % find Hello.pkg
   Hello.pkg
   Hello.pkg/Contents
   Hello.pkg/Contents/Archive.bom
   Hello.pkg/Contents/Archive.pax.gz
   Hello.pkg/Contents/Info.plist
   Hello.pkg/Contents/PkgInfo
   Hello.pkg/Contents/Resources
   Hello.pkg/Contents/Resources/BundleVersions.plist
   Hello.pkg/Contents/Resources/English.lproj/License.rtf
   Hello.pkg/Contents/Resources/English.lproj/ReadMe.rtf
   Hello.pkg/Contents/Resources/English.lproj/Welcome.rtf
   ```

   Refer to "Anatomy of a Package" (page 71) and "Gather Any Package Resources" (page 75) for descriptions of the `Hello` and `.rtf` files and their contents.

Test Installing the Package

Now you can test your package. Mac OS X will run the Installer application if you double-click a package.

1. From a Desktop Finder window, navigate to the location where you told PackageMaker to save your package (repeat step #1 in "Examine the Package and Files" (page 82)).

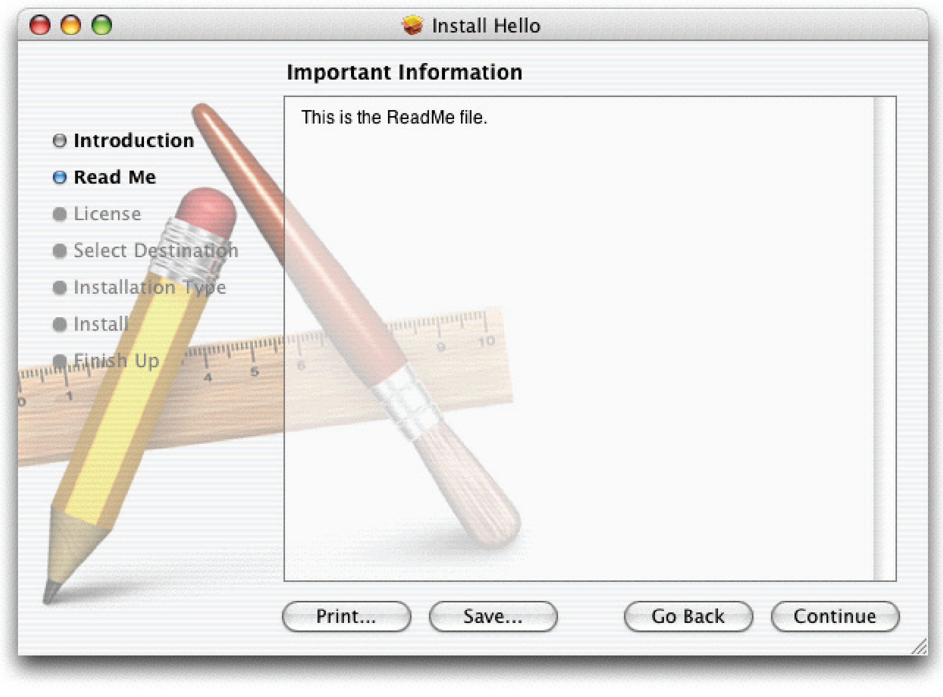
2. Select and double-click the package, for example, `Hello.pkg`.

3. The Mac OS X Installer launches.

4. The Introduction window will be displayed, along with the welcome file, if you created one. Click Continue.

5. If you created a Read Me file, you should see it displayed with the heading "Important Information" as shown in Figure 6. Click Continue.

Figure 6 Installer displays Read Me file



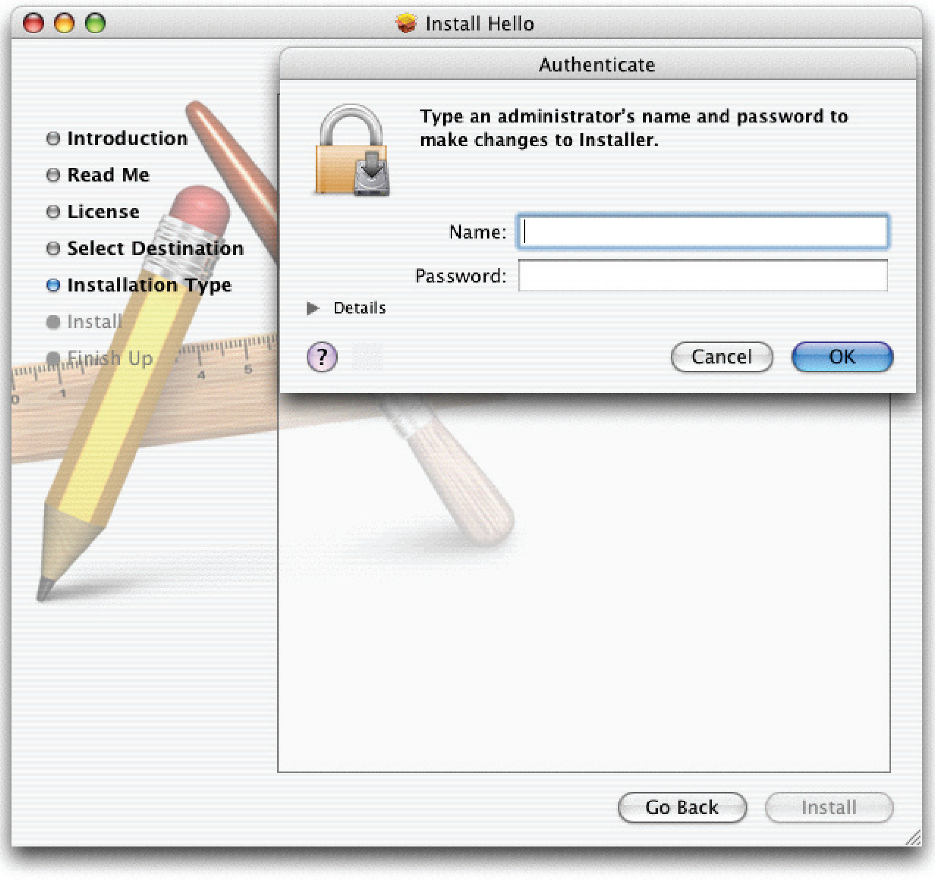
This is the ReadMe file.

6. If you created a license file, you should see it displayed with the heading "Software License Agreement". Click Continue. Installer will display a dialog requiring you to agree (or disagree) to the terms of the license. Click Agree.

7. If appropriate, select a destination volume and click Continue.

8. If you required a higher level of authorization than you are currently logged in at, Installer will display an authorization dialog and ask you to type your password, as in Figure 7.

Figure 7 Installer displays an authentication dialog



Type an administrator's name and password to make changes to Installer.

Name:
Password:

9. Click Install (or Upgrade if you've installed this package before) and allow Installer to proceed. Click Close when Installer has finished.

Now you can check that the package was installed. Navigate to `/System/Library/Extensions`. You should see your KEXT (`Hello.kext`). Remember that the KEXT is a bundle; you can view the contents of the KEXT from a Terminal shell. If you do, you can see that the Read Me, software license, and welcome files were not installed.

Where to Go Next

Congratulations! You've now packaged and installed a KEXT.

This tutorial is part of a series. An additional tutorial describes how to debug a kernel extension ("Hello Debugger: Debugging a Device Driver With GDB" (page 53)). This tutorial, as well as the earlier (prerequisite) tutorials, can be found in *Kernel Extension Programming Topics*.

Loading Kernel Extensions at Boot Time

This document describes the boot-time loading of kernel extensions (KEXTs) in Mac OS X. In particular, it focuses on the BootX booter and its use of the KEXT property `OSBundleRequired`. If you are not familiar with the boot sequence or if you would like more information on booting and system and kernel initialization, see *System Startup Programming Topics*. More information on KEXTs and their information property lists can be found in *I/O Kit Fundamentals*.

The BootX Booter

In order to mount the root file system, the kernel must contain the KEXTs that drive the hardware required to access the root volume. Rather than building these KEXTs into the kernel, Mac OS X provides for boot-time loading of KEXTs by the BootX booter. This is accomplished in part through use of the `OSBundleRequired` property in the root dictionary of the KEXT's information property list (`Info.plist` file). The BootX booter, the KEXT loading utility `kextd`, and the KEXT caching utility `kextcache` use the `OSBundleRequired` property to determine which KEXTs are necessary to load or cache.

Important: It is essential that only those KEXTs required for hardware in the boot process and the minimal set of KEXTs required to operate the system's user interface include the `OSBundleRequired` property in their `Info.plist` files. See "The OSBundleRequired Property" (page 89) for information on how to set this property to the correct value.

Near the beginning of the boot sequence for Mac OS X, the BootX booter receives control from BootROM which has initialized the system hardware and selected an operating system to boot. BootX's primary responsibility is to load the kernel environment. To do this, BootX first attempts to load a previously cached set of device drivers (called an mkext cache) for hardware that is in the boot process. If this cache is missing or corrupt, BootX searches `/System/Library/Extensions` for KEXTs whose `OSBundleRequired` property is set to a value appropriate to the type of system boot (see Table 1 (page 89) for a complete list of values used with this property). For example, if BootX is performing a network boot, it will look for KEXTs whose `OSBundleRequired` property is set to "Network-Root". When booting in single-user mode, BootX will load those KEXTs whose `OSBundleRequired` property is set to "Console", in addition to loading the KEXTs required to mount the root file system. BootX ignores all KEXTs whose `OSBundleRequired` property is set to "Safe Boot" and all KEXTs whose `Info.plist` files do not contain this property at all.

After the root file system is mounted, `kextd` starts and examines all the drivers available on the system. At this point, any unnecessary drivers get unloaded, freeing up memory; and `kextd` attempts to fulfill any KEXT loading requests it receives.

Safe Booting

The `OSBundleRequired` property is also used to facilitate safe-boot mode in which all unnecessary extensions are disabled. During a safe boot, BootX loads the KEXTs required to mount the root file system just as in other modes of booting. Then `kextd` starts and examines all the drivers on the system. If this is a safe boot, however, `kextd` will only consider loading a KEXT if the `OSBundleRequired` property is present in its `Info.plist` file. This preserves the integrity of the safe boot by ensuring that only drivers necessary for mounting root or operating the system's user interface are loaded.

Safe booting helps protect the system from the possible panic of a driver. If a driver's `OSBundleRequired` property is set to "Root", it will always be loaded by BootX, even in single-user or safe-boot modes. If this driver panics, the system itself may panic, requiring reinstallation of the operating system or booting off a CD. If the same driver's `OSBundleRequired` property is set to "Safe Boot", however, it will never be loaded by BootX. `kextd` will load it after the root file system has been mounted and if the driver panics it can be disabled by re-booting into single-user mode and then using the console to move it out of the way. The safe boot can then be restarted without the panicking driver.

Note: During the development phase of your driver, you may find it useful to set the `IOKitDebug` key in your driver's personality to a nonzero value. If you do so, that driver personality will not be sent to the kernel during a safe boot. If all of your driver's personalities have nonzero `IOKitDebug` values, the KEXT itself will not even be loaded into the kernel during a safe boot.

To initiate a safe-boot, hold down the Shift key while restarting the computer.

The Mkext Cache

The mkext cache is created by the `kextcache` utility which takes a directory of KEXTs and archives it into a compressed binary form suitable for putting in ROM or other confined space. The utility `kextcache` also makes use of the `OSBundleRequired` property when determining which KEXTs to cache. When being used to create a net-boot repository, `kextcache` includes all KEXTs with `OSBundleRequired` set to "Root", "Network-Root", or "Console". When being used to create a local-boot repository, `kextcache` will include all KEXTs with `OSBundleRequired` set to "Root", "Local-Root", or "Console". The default behavior of `kextcache` is to create a full cache of all KEXTs it is given. For a complete description of the use of this utility, access the Terminal application in Mac OS X (located in `/Applications/Utilities/`) and type `man kextcache`.

The OSBundleRequired Property

The `OSBundleRequired` property should appear in the root dictionary of a KEXT's `Info.plist` if that KEXT is required to mount the root file system (such as a platform driver) or is required to operate the system's user interface (such as a mouse or graphics driver). A driver that does not fall into either category (for example, an audio driver) should not include the `OSBundleRequired` property in its `Info.plist` at all. Such a driver will be loaded by `kextd` as needed unless booting in safe-boot mode (see "Safe Booting" (page 88) for an explanation of this process).

For KEXTs that do include this property, it is essential that it be set to the appropriate value. As explained in "Safe Booting" (page 88), a KEXT whose `OSBundleRequired` property is set to "Root" will always be loaded by BootX when it attempts to mount the root file system. Therefore, unless a KEXT is absolutely required to mount root, its `OSBundleRequired` property should not be set to "Root".

Table 1 (page 89) matches the values of the `OSBundleRequired` property with descriptions of the KEXTs that should use them. If a KEXT does not fit any of these descriptions, it should not include the `OSBundleRequired` property in its `Info.plist` file.

Table 1 Values of the OSBundleRequired property and their usages

Value	Usage
Root	This KEXT is required to mount root, regardless of where root comes from – for example, platform drivers and families, PCI, or USB.
Network-Root	This KEXT is required to mount root on a remote volume—for example, the network family, Ethernet drivers, or NFS.
Local-Root	This KEXT is required to mount root on a local volume – for example, the storage family, disk drivers, or file systems.
Console	This KEXT is required to provide character console support (single-user mode) – for example, keyboard drivers or the ADB family.
Safe Boot	This KEXT is required even during safe-boot (unnecessary extensions disabled)—for example, mouse drivers or graphics drivers.

Kernel Extension Dependencies

Before loading a kernel extension (KEXT), the kernel extension management facilities in Mac OS X check its compatibility with other loadable extensions it depends on and with the kernel itself. To make sure your KEXT loads, be sure you know what your KEXT depends on and how to declare those dependencies.

This document provides kernel subcomponent and loadable extension versions for Mac OS X versions 10.0 through 10.4.2. It also describes the kernel programming interface (KPI) collections introduced in Mac OS X version 10.4 and how to use them.

KEXT Versions and Dependencies

Every kernel extension has a version that is stored both in its information property list (`Info.plist` file) and in its executable. The version number is stored as a string property in the `'vers'` resource style named `CFBundleVersion` (described in http://developer.apple.com/documentation/mac/Toolbox/Toolbox-454.html and Technical Note TN1132). A KEXT or kernel component that can be depended on also declares the earliest version its current version is compatible with. This version (also in the `'vers'` resource style) is stored in the `OSBundleCompatibleVersion` property at the top level of its `Info.plist` file.

A KEXT declares dependencies on other loadable extensions or kernel subcomponents by listing them as elements in the `OSBundleLibraries` dictionary at the top level of its `Info.plist` file. Each element in the `OSBundleLibraries` dictionary consists of a key/value pair. The key is the `CFBundleIdentifier` of the dependency (such as `com.apple.kernel.mach`), and the value is the required version of the dependency expressed as a `'vers'` resource style string (such as `1.0b1`). When a KEXT is about to be loaded, the required version of each element in its `OSBundleLibraries` dictionary is compared to the current and compatible versions of the dependency. If the required version lies between the current version of the dependency and its `OSBundleCompatibleVersion` value, inclusive, the KEXT and its dependencies are deemed compatible and loading proceeds.

If you fail to declare the appropriate dependencies, your KEXT will probably compile, but it will not load. To help you catch such problems early, it's good practice to use the `kextload` tool to load your KEXT periodically thoughout the development cycle. The `kextload` tool has options that allow it to perform a wide range of tests; for more information, see Mac OS X Man Pages or type `man kextload` in a Terminal window.

KPIs and KEXT Dependencies in Mac OS X v10.4 and Later

In Mac OS X v10.4, Apple introduced sustainable kernel programming interfaces, or KPIs. In particular, the KPIs support the development of NKEs (network kernel extensions), file-system KEXTs, and other non-I/O Kit KEXTs. For more information on the KPIs themselves, see *KPI Reference*.

The KPIs are divided into collections which correspond in name to the kernel subcomponents available in earlier versions of Mac OS X. For example, the KPI collection that corresponds to the `com.apple.kernel.libkern` kernel subcomponent is `com.apple.kpi.libkern`. In Mac OS X v10.4 and later, you declare dependencies on KPI collections by creating key/value pairs in your KEXT's `OSBundleLibraries` dictionary, just as you did for dependencies on kernel subcomponents (as described in "KEXT Versions and Dependencies" (page 91)). The key is the KPI collection's `CFBundleIdentifier` (such as `com.apple.kpi.bsd`) and the value is the current version of the Darwin kernel expressed as a `'vers'` resource style string. Because you can only declare a dependency on a KPI collection in Mac OS X v10.4 and later, the minimum version number is `8.0`. To get the current version of the Darwin kernel, type `uname -a` in a Terminal window.

For pure I/O Kit KEXTs (those that use only I/O Kit-provided APIs), no changes in declared dependencies are necessary. This is because Mac OS X v10.4 provides backward compatibility with the I/O Kit kernel subcomponents available in earlier versions of Mac OS X. Therefore, you can continue to declare your pure I/O Kit KEXT's dependencies on the kernel subcomponent versions of the earliest version of Mac OS X you need to run in. For more information on how to do this, see "Declaring Dependencies on Kernel Subcomponents" (page 93).

Note: If you are developing a universal I/O Kit device driver, see Technical Note TN2163: Building Universal I/O Kit Drivers for information on how to configure your Xcode project and declare the correct dependencies for your situation.

You should consider, however, taking this opportunity to make sure all your pure I/O Kit KEXT's dependencies are explicitly declared. If, for example, your KEXT uses libkern symbols, declare an explicit dependency on `com.apple.kernel.libkern`. This may seem obvious, but some earlier versions of Mac OS X allowed implicit dependencies on some kernel subcomponents (such as `com.apple.kernel.libkern`) when you declared an explicit dependency on `com.apple.kernel.iokit`. The KPI collections, on the other hand, do not allow any implicit dependencies. Therefore, if your KEXT declares only explicit dependencies on kernel subcomponents, it will make it much easier to switch to dependencies on KPIs at some point in the future.

If your I/O Kit-based KEXT makes non-I/O Kit BSD or Mach calls, it may not load in Mac OS X v10.4 because some non-I/O Kit symbols are no longer available. If this describes your KEXT, you must develop a new version of your KEXT for Mac OS X v10.4 that depends on the appropriate KPI collections instead of the kernel subcomponents it depended on before.

Important: You cannot combine dependencies on KPI collections and dependencies on kernel subcomponents in the same KEXT; dependencies must be all of one type or the other. If your KEXT declares dependencies on both KPI collections and kernel subcomponents, it will fail to load.

Beginning in Mac OS X v10.4, there are three, mutually exclusive ways to declare KEXT dependencies, the first two of which are discussed above. The following list summarizes the three methods, explains how to get the correct version for each type of dependency, and describes the circumstances in which you can use each method.

- **Declare dependencies on KPI collections**. This is the preferred method because doing so ensures your KEXT links against the latest, sustainable interfaces for the kernel. When you use this method, the version of each KPI collection must match the version of the Darwin kernel (8.0 or greater).

  This method is required for NKEs, file-system KEXTs, and other non-I/O Kit KEXTs.

  This method is required for I/O Kit-based KEXTs that use non-I/O Kit functions (such as BSD or Mach functions) and that must run in Mac OS X v10.4 and later.

  This method is not required for pure I/O Kit KEXTs, but it is strongly recommended for pure I/O Kit KEXTs that will run only in Mac OS X v10.4 and later.

- **Declare dependencies on the kernel subcomponents available in earlier versions of Mac OS X**. This method provides backward compatibility for pure I/O Kit KEXTs, but some Mach and BSD symbols are no longer available. When you use this method, use the version of the kernel subcomponent that corresponds to the earliest version of Mac OS X you need to support (see the tables in "Declaring Dependencies on Kernel Subcomponents" (page 93) for these values).

  This method is suitable for pure I/O Kit KEXTs that must run in versions of Mac OS X prior to Mac OS X v10.4.

  This method will not work for NKEs, file-system KEXTs, or other non-I/O Kit KEXTs. In addition, this method will not work for I/O Kit-based KEXTs running in Mac OS X v10.4 and later that use non-I/O Kit functions.

- **Declare a dependency on a specific version of the entire kernel**. This method provides no binary compatibility between Mac OS X releases and is provided as a convenience for testing or educational purposes only. When you use this method, use the build version of the kernel displayed when you type `sw_vers` in a Terminal window.

  This method is recommended for in-house or educational environments in which access to internal kernel interfaces is desirable, but binary compatibility of a product is not required. You might also use this method in an open-source environment in which users of your product expect to rebuild it with every update.

Warning: This method is absolutely unsuitable for development of a consumer product. If you declare your KEXT's dependency on a specific version of the kernel, it will load on only that version and no other. If a user performs a software update or upgrades the kernel in any way, your KEXT will no longer load.

Declaring Dependencies on Kernel Subcomponents

The following tables show the `CFBundleVersion` value of each kernel subcomponent. Note that Table 1 (page 94) also includes the `CFBundleVersion` values for select kernel extensions. This is because in versions of Mac OS X prior to 10.1, these kernel extensions were included in the kernel. In Mac OS X version 10.1, however, most of these kernel extensions were removed from the kernel and made into loadable kernel extensions. The remaining four tables do not list these loadable kernel extensions.

Table 1 Kernel subcomponent CFBundleVersion values for Mac OS X, version 10.0 through 10.1

CFBundleIdentifier	10.0	10.0.1	10.0.2	10.0.3	10.0.4 Tier 2 Language Support	10.0.4 Software Update	10.1
com.apple.kernel	1.0b1	1.0b1	1.3.2	1.3.3	1.3.5	1.3.7	1.4
com.apple.kernel.mach	1.0b1	1.0b1	1.0.2	1.0.3	1.0.5	1.0.7	1.1
com.apple.kernel.bsd	1.0b1	1.0b1	1.0.2	1.0.3	1.0.5	1.0.7	1.1
com.apple.kernel.libkern	1.0b1	1.0b1	1.0.2	1.0.3	1.0.5	1.0.7	1.1
com.apple.kernel.iokit	1.0b1	1.0b1	1.0.2	1.0.3	1.0.5	1.0.7	1.1
com.apple.iokit.IOADBFamily	1.0b1	1.0b1	1.0.2	1.0.3	1.0.5	1.0.7	1.1(M)
com.apple.iokit.IOCDStorage-Family	1.0b1	1.0b1	1.0.2	1.0.3	1.0.5	1.0.7	1.1(L)
com.apple.iokit.IODVDStorage-Family	1.0b1	1.0b1	1.0.2	1.0.3	1.0.5	1.0.7	1.1(L)
com.apple.iokit.IOGraphics-Family	1.0b1	1.0b1	1.0.2	1.0.3	1.0.5	1.0.7	1.1(L)
com.apple.iokit.IOHIDSystem	1.0b1	1.0b1	1.0.2	1.0.3	1.0.5	1.0.7	1.1(L)
com.apple.iokit.IONDRVSupport	1.0b1	1.0b1	1.0.2	1.0.3	1.0.5	1.0.7	1.1(L)
com.apple.iokit.IONetworking-Family	1.0b1	1.0b1	1.0.2	1.0.3	1.0.5	1.0.7	1.1(L)
com.apple.iokit.IOPCIFamily	1.0b1	1.0b1	1.0.2	1.0.3	1.0.5	1.0.7	1.1(L)
com.apple.iokit.IOStorage-Family	1.0b1	1.0b1	1.0.2	1.0.3	1.0.5	1.0.7	1.1(L)
com.apple.iokit.IOSystem-ManagementFamily	1.0b1	1.0b1	1.0.2	1.0.3	1.0.5	1.0.7	1.1(M)

Note: Component versions marked (L) in Table 1 indicate that the component was removed from the kernel and made into a loadable kernel extension.

Component versions marked (M) in Table 1 indicate that the component was moved into the `System.kext` PlugIns folder.

The following tables (Table 2 (page 95), Table 3 (page 95), Table 4 (page 96), and Table 5 (page 96)) each list kernel subcomponent `CFBundleVersion` values for a range of operating system releases, beginning with Mac OS X version 10.1.1. The kernel subcomponents in these tables are:

- `com.apple.kernel`

- `com.apple.kernel.mach`
- `com.apple.kernel.bsd`
- `com.apple.kernel.libkern`
- `com.apple.kernel.iokit`

In the interests of space, the `com.apple.` prefix is not included in each kernel subcomponent `CFBundleIdentifier` value.

Table 2 Kernel subcomponent CFBundleVersion values for Mac OS X, versions 10.1.1 through 10.1.5

System release	`kernel`	`kernel.mach`	`kernel.bsd`	`kernel.libkern`	`kernel.iokit`
10.1.1 (Build 5M45)	5.1	5.1	5.1	5.1	5.1
10.1.2 (Build 5P68)	5.2.2	5.2	5.2	5.2	5.2.2
10.1.3 (Builds 5Q45, 5Q83)	5.3	5.3	5.3	5.3	5.3
10.1.4 (Build 5Q125)	5.4	5.4	5.4	5.4	5.4
10.1.4 (Builds 5R48, 5R60)	5.3.2	5.3.2	5.3.2	5.3.2	5.3.2
10.1.4 (Build 5R106)	5.4.2	5.4.2	5.4.2	5.4.2	5.4.2
10.1.5 (Build 5S60)	5.5	5.5	5.5	5.5	5.5

Table 3 Kernel subcomponent CFBundleVersion values for Mac OS X, versions 10.2 through 10.2.8

System release	`kernel`	`kernel.mach`	`kernel.bsd`	`kernel.libkern`	`kernel.iokit`
10.2 (Build 6C115)	6.0	6.0	6.0	6.0	6.0
10.2.1 (Builds 6DF52-6E62)	6.1	6.1	6.1	6.1	6.1
10.2.2 (Builds 6F21, 6F39)	6.2	6.2	6.2	6.2	6.2
10.2.3 (Builds 6G30-6H59)	6.3	6.3	6.3	6.3	6.3
10.2.4 (Builds 6I32, 6I34)	6.4	6.4	6.4	6.4	6.4
10.2.5 (Build 6L29)	6.5	6.5	6.5	6.5	6.5
10.2.6 (Build 6L60)	6.6	6.6	6.6	6.6	6.6
10.2.6 (Builds 6R106-6R132)	6.8	6.8	6.8	6.8	6.8
10.2.7 (Builds 6R42-6R55)	6.7	6.7	6.7	6.7	6.7
10.2.8 (Builds 6R65, 6R73)	6.8	6.8	6.8	6.8	6.8

Table 4 G5 kernel subcomponent CFBundleVersion values for Mac OS X versions 10.2.7 and 10.2.8

System release	`kernel`	`kernel.mach`	`kernel.bsd`	`kernel.libkern`	`kernel.iokit`
10.2.7 (Builds 6S74-6S80)	6.7.5	6.7.5	6.7.5	6.7.5	6.7.5
10.2.8 (Build 6S90)	6.8.5	6.8.5	6.8.5	6.8.5	6.8.5

Table 5 Kernel subcomponent CFBundleVersion values for Mac OS X versions 10.3 through 10.3.9

System release	`kernel`	`kernel.6.0`	`kernel.mach`	`kernel.bsd`	`kernel.libkern`	`kernel.iokit`
10.3 (Build 7B85)	7.0	6.9.9	6.9.9	6.9.9	6.9.9	6.9.9
10.3.1 (Builds 7C107-7C9)	7.0	6.9.9	6.9.9	6.9.9	6.9.9	6.9.9
10.3.2 (Builds 7D24-7D28)	7.2	6.9.9	6.9.9	6.9.9	6.9.9	6.9.9
10.3.2 (Builds 7E40, 7E46)	7.2.1	6.9.9	6.9.9	6.9.9	6.9.9	6.9.9
10.3.3 (Build 7F44)	7.3	6.9.9	6.9.9	6.9.9	6.9.9	6.9.9
10.3.3 (Build 7G21)	7.3.1	6.9.9	6.9.9	6.9.9	6.9.9	6.9.9
10.3.4 (Builds 7H63-7H142)	7.4	6.9.9	6.9.9	6.9.9	6.9.9	6.9.9
10.3.4 (Builds 7L32-7L46)	7.4.1	6.9.9	6.9.9	6.9.9	6.9.9	6.9.9
10.3.5 (Builds 7M34-7P36)	7.5	6.9.9	6.9.9	6.9.9	6.9.9	6.9.9
10.3.5 (Builds 7P124-7P220)	7.5.1	6.9.9	6.9.9	6.9.9	6.9.9	6.9.9
10.3.6 (Builds 7R28-7R107)	7.6	6.9.9	6.9.9	6.9.9	6.9.9	6.9.9
10.3.7 (Build 7S215)	7.7	6.9.9	6.9.9	6.9.9	6.9.9	6.9.9
10.3.7 (Build 7T21)	7.7.1	6.9.9	6.9.9	6.9.9	6.9.9	6.9.9
10.3.7 (Builds 7T51-7T62)	7.7.2	6.9.9	6.9.9	6.9.9	6.9.9	6.9.9

System release	`kernel`	`kernel.6.0`	`kernel.mach`	`kernel.bsd`	`kernel.libkern`	`kernel.iokit`
10.3.8 (Build 7U16)	7.8	6.9.9	6.9.9	6.9.9	6.9.9	6.9.9
10.3.9 (Build 7W98)	7.9	6.9.9	6.9.9	6.9.9	6.9.9	6.9.9

Table 6 Kernel subcomponent CFBundleVersion values for Mac OS X versions 10.4 through 10.4.10

System release	`kernel`	`kernel.6.0`	`kernel.mach`	`kernel.bsd`	`kernel.libkern`	`kernel.iokit`
10.4 (Build 8A428)	8.0	7.9.9	7.9.9	7.9.9	7.9.9	7.9.9
10.4.1 (Build 8B15)	8.1	7.9.9	7.9.9	7.9.9	7.9.9	7.9.9
10.4.2 (Builds 8C46, 8E45, 8E90, and 8E102)	8.2	7.9.9	7.9.9	7.9.9	7.9.9	7.9.9
10.4.3 (Build 8F46)	8.3	7.9.9	7.9.9	7.9.9	7.9.9	7.9.9
10.4.4 (Builds 8G32 and 8G1165)	8.4	7.9.9	7.9.9	7.9.9	7.9.9	7.9.9
10.4.5 (Builds 8H14 and 8G1454)	8.5	7.9.9	7.9.9	7.9.9	7.9.9	7.9.9
10.4.6 (Builds 8I127 and 8I1119)	8.6	7.9.9	7.9.9	7.9.9	7.9.9	7.9.9

System release	`kernel`	`kernel.6.0`	`kernel.mach`	`kernel.bsd`	`kernel.libkern`	`kernel.iokit`
10.4.7 (Builds 8J135, 8J2135a, and 8K1079)	8.7	7.9.9	7.9.9	7.9.9	7.9.9	7.9.9
10.4.8 (Builds 8L127 and 8L2127)	8.8	7.9.9	7.9.9	7.9.9	7.9.9	7.9.9
10.4.9 (Builds 8P135 and 8P2137)	8.9	7.9.9	7.9.9	7.9.9	7.9.9	7.9.9
10.4.10 (Builds 8R218 and 8R2218)	8.10	7.9.9	7.9.9	7.9.9	7.9.9	7.9.9

You can also get this information by using the **kextstat** utility to examine the version numbers of these components. (For more information on how to use **kextstat**, type `man kextstat` in a Terminal window.) To build a KEXT that works on two different versions of Mac OS X, build the KEXT on the oldest version you want to support.

If an I/O Kit kernel extension does not declare a dependency that defines a superclass but otherwise uses no symbols within that dependency, the KEXT will load successfully but the runtime metaclass system will fail to create an instance of the class, due to the absence of the superclass. In this case, an error message to this effect will be displayed.

The identifier `com.apple.kernel` represents the kernel as a whole. Because every kernel extension is linked against the kernel, it may seem easier to declare your KEXT's dependency on `com.apple.kernel` instead of the individual kernel subcomponents. However, if any part of the kernel changes in an incompatible way, the compatible version (the `OSBundleCompatibleVersion` value) for `com.apple.kernel` will be incremented. This may prevent your KEXT from loading even if the kernel subcomponents your KEXT actually depends on remain unchanged.

To determine which of the four kernel subcomponents your KEXT depends on, examine the `#include` directives in your KEXT's code. Table 7 lists header directories and their corresponding kernel subcomponents.

Table 7 Header directories and corresponding kernel subcomponents

Header Directory	Module Dependency
architecture	`com.apple.kernel.libkern`
bsd	`com.apple.kernel.bsd`, `com.apple.kernel.mach`
crypto	`com.apple.kernel.bsd`, `com.apple.kernel.mach`
ddb	`com.apple.kernel.mach`
default_pager	`com.apple.kernel.mach`
dev	`com.apple.kernel.bsd`, `com.apple.kernel.mach`
device	`com.apple.kernel.mach`
hfs	`com.apple.kernel.bsd`, `com.apple.kernel.mach`
i386	`com.apple.kernel.bsd`, `com.apple.kernel.mach`
IOKit	`com.apple.kernel.iokit`, `com.apple.kernel.libkern`
ipc	`com.apple.kernel.mach`
isofs	`com.apple.kernel.bsd`, `com.apple.kernel.mach`
kern	`com.apple.kernel.bsd`, `com.apple.kernel.mach`
libkern	`com.apple.kernel.libkern`
libsa	Kernel extensions should not use this API
mach	`com.apple.kernel.mach`
mach-o	Kernel extensions should not use this API
mach_debug	`com.apple.kernel.bsd`, `com.apple.kernel.mach`
machine	`com.apple.kernel.bsd`, `com.apple.kernel.mach`
miscfs	`com.apple.kernel.bsd`, `com.apple.kernel.mach`
net	`com.apple.kernel.bsd`, `com.apple.kernel.mach`
netat	`com.apple.kernel.bsd`, `com.apple.kernel.mach`
netccitt	`com.apple.kernel.bsd`, `com.apple.kernel.mach`
netinet	`com.apple.kernel.bsd`, `com.apple.kernel.mach`
netinet6	`com.apple.kernel.bsd`, `com.apple.kernel.mach`
netiso	`com.apple.kernel.bsd`, `com.apple.kernel.mach`

Header Directory	Module Dependency
netkey	`com.apple.kernel.bsd`, `com.apple.kernel.mach`
netns	`com.apple.kernel.bsd`, `com.apple.kernel.mach`
nfs	`com.apple.kernel.bsd`, `com.apple.kernel.mach`
pexpert	Kernel extensions should not use this API
ppc	`com.apple.kernel.bsd`, `com.apple.kernel.mach`
profile	Kernel extensions should not use this API
sys	`com.apple.kernel.bsd`, `com.apple.kernel.mach`
ufs	`com.apple.kernel.bsd`, `com.apple.kernel.mach`
UserNotification	`com.apple.kernel.mach`
vfs	`com.apple.kernel.bsd`, `com.apple.kernel.mach`
vm	`com.apple.kernel.mach`

If your KEXT can be depended on by other kernel extensions you must declare the earliest version that remains compatible with the current version in the `OSBundleCompatibleVersion` property at the top level of its `Info.plist` file. If this property is absent, no other kernel extension can declare a dependency on your KEXT. If you make changes to your KEXT that break binary compatibility with earlier versions (for example, the removal of a function or symbol) then you should change the value of the `OSBundleCompatibleVersion` property to equal the value of the `CFBundleVersion` property of the new version. It is not necessary to increment the `OSBundleCompatibleVersion` value every time you increment the `CFBundleVersion` value unless you've broken binary compatibility with a previously compatible version.

Installing and Removing Kernel Extensions

Software installers that install and remove kernel extensions can copy them to `/System/Library/Extensions` or remove them from that folder. To load a new, nondriver KEXT either the system must be restarted or the new KEXT must be explicitly loaded with the **kextload** utility. To load a driver KEXT the system must be restarted to ensure reliable matching and loading of the driver for all possible devices.

The system keeps a cache of installed KEXTs to speed up boot time. It updates this cache when it detects any change to the `/System/Library/Extensions` folder. If an installer installs an extension as a plug-in of another, however, only a subfolder of the `/System/Library/Extensions` folder is updated and the automatic cache update is not triggered. To force a cache update, you can change the modification date of the `/System/Library/Extensions` folder by using the UNIX command **touch**. For example, placing

```
touch /System/Library/Extensions
```

in a postprocessing shell script for your installer will guarantee that the cache gets updated.

Kernel Extension Ownership and Permissions

Because kernel extensions (KEXTs) contain code and data that are loaded into the kernel, the most protected environment in the operating system, their file ownership and permissions must be set to prevent unauthorized tampering. In fact, a KEXT will not load into the kernel unless its ownership and permissions are correct. Read this article to find out what the correct values are and how to make sure your KEXT has them.

For security reasons, no component of a KEXT should be writable by any user other than the superuser. Specifically, this means that:

- All files and folders in the KEXT, including the KEXT itself, must be owned by the root user (UID 0).
- All files and folders in the KEXT, including the KEXT itself, must be owned by the wheel group (GID 0).
- All folders in the KEXT, including the KEXT itself, must have the permissions 0755 (octal) or `rwxr-xr-x` (as shown by `ls -l`).
- All files in the KEXT must have permissions 0644 (octal) or `rw-r--r--` (as shown by `ls -l`). Note that a KEXT is *not* the place to store a user-space executable.

There are two common ways to ensure that your KEXT has the correct ownership and permissions:

- One way is to temporarily assume root-user privileges and copy the KEXT to a temporary location, as shown below:

  ```
  % sudo cp -R MyKEXT.kext /tmp
  Password:
  ```

  You use the `-R` option with the `cp` command to make sure that both the KEXT directory and its entire subtree are copied.

  Note: This method leaves the permissions and ownership of your original KEXT alone, so you can continue to revise and save it. However, this means that every time you make changes to the original KEXT and rebuild it, you must repeat the copy action shown above to make sure the new version has the correct ownership and permissions.

- Another way to give your KEXT the correct ownership and permissions is to include this task in post-build and post-install scripts. To do this, you can include the following shell script commands:

  ```
  /usr/sbin/chown -R root:wheel MyKEXT.kext
  ```

```
find MyKEXT.kext -type d -exec /bin/chmod 0755 {} \;
find MyKEXT.kext -type f -exec /bin/chmod 0644 {} \;
```

To find out more about packaging scripts with your KEXT, see "Anatomy of a Package" (page 71).

Document Revision History

This table describes the changes to *Kernel Extension Programming Topics*.

Date	Notes
2007-10-31	Changed title from "Kernel Extension Concepts." Updated debugging instructions to better explain how to generate symbol files on the host machine.
2007-06-08	Updated kernel debugging information for Mac OS X v10.5.
2007-04-03	Added links to KPI usage information and consolidated permissions and ownership information into a separate article.
2006-10-03	Updated for Xcode 2.4 and added guidelines for using IOMatchCategory.
2006-05-23	Updated the PackageMaker information and added information about the use of the NVRAM variable pmuflags while debugging.
2006-02-07	Updated instructions for enabling kernel debugging on Intel-based Macintosh computers and for editing property list files in Xcode.
2005-10-04	Corrected version information for alternate debugger keystroke.
2005-09-08	Added information about debugging KEXTs on Intel-based Macintosh computers.
2005-08-11	Added information about kernel programming interfaces.
2005-04-29	Added description of new way to break into kernel debugging mode in Mac OS X v. 10.4.
2005-03-03	Kernel subcomponent version information added for Mac OS X versions 10.3.4 through 10.3.7.
2004-02-25	Converted *KEXT Tutorials* HOWTO documents to *Kernel Extension Concepts* programming topic. Updated tutorials to use Xcode 1.1 on Mac OS X version 10.3.