Kernel Extension Programming Topics

Drivers, Kernel, & Hardware: Kernel Device Drivers

2010-03-19

Apple Inc.
© 2003, 2010 Apple Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

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

Apple, the Apple logo, Finder, FireWire, Leopard, Mac, Mac OS, Macintosh, and Xcode are trademarks of Apple Inc., registered in the United States and other countries.

Intel and Intel Core are registered trademarks of Intel Corportation or its subsidiaries in the United States and other countries.

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Introduction 9

Who Should Read This Document? 9
Organization of This Document 9
See Also 10

Deciding Whether to Create a Kernel Extension 11

Make Sure Your Code Needs to Run in Kernel Space 11
Proceed with Caution 11

The Anatomy of a Kernel Extension 13

A Kext Bundle Contains Two Main Components 13
- The Information Property List 13
- The Executable 13
- Additional Resources and Plug-ins 14

Kernel Extensions Have Strict Security Requirements 14
Kernel Extensions Should Reside in /System/Library/Extensions 15

Creating a Generic Kernel Extension with Xcode 17

Road Map 17
Create a New Project 17
Implement the Start and Stop Functions 18
- Implement the Start and Stop Functions 18
- Edit the Information Property List 20
- Build the Kernel Extension 21

Add Library Declarations 21
- Run kextlibs on the Kernel Extension 22
- Add the Library Declarations to the Information Property List 22

Prepare the Kernel Extension for Loading 24
- Set the Kernel Extension's Permissions 24
- Run kextutil 24

Where to Go Next 25

Creating a Device Driver With Xcode 27

Road Map 27
Familiarize Yourself with the I/O Kit Architecture 27
Create a New Project 28
Edit the Information Property List 28

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

Fill in the Header File 31
Implement the Driver's Entry Points 33
Add Library Declarations 34
Run kextlibs on the Driver 34
Add the Library Declarations to the Information Property List 35
Prepare the Driver for Loading 37
Set the Driver's Permissions 37
Run kextutil 37
Where to Go Next 38

Debugging a Kernel Extension With GDB 39

Road Map 39
Prepare the Machines 40
On the Development Machine, Sabotage the Kernel Extension 40
On the Target Machine, Enable Kernel Debugging 41
On the Target Machine, Get the Target Machine's IP Address 41
On the Development Machine, Start GDB 41
On the Target Machine, Load the Kernel Extension 42
On the Development Machine, Attach to the Target Machine 42
On the Development Machine, Get the Load Address of the Kernel Extension 42
On the Development Machine, Create and Load the Symbol File 42
On the Development Machine, Debug with GDB 43
On the Development Machine, Stop the Debugger 44
Where to Go Next 44

Command-Line Tools for Analyzing Kernel Extensions 45

Generate Debug Symbols and Prepare Kexts for Loading with kextutil 45
Output the Status of Loaded Kexts with kextstat 45
Determine Kext Dependencies with kextlibs 46
Locate Kexts with kextfind 46
Obtain Instance Counts with ioclasscount 46
View the I/O Kit Registry with IORegistryExplorer 47

Packaging a Kernel Extension for Distribution and Installation 49

Road Map 49
Set Permissions for your Kext 49
Create Custom Installer Information 50
The Welcome Message 50
The Read Me 50
The Software License Agreement 50
Create a Package with PackageMaker 51
Add Preinstall and Postinstall Actions (Optional) 53
Require Restart 53

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

Add Actions 53
Build the Package and Test Installation 53

Info.plist Properties for Kernel Extensions 55

Top-Level Properties 55
IOKitPersonalities Properties 57
Architecture-Specific Properties 58

Document Revision History 59

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

Figures, Tables, and Listings

The Anatomy of a Kernel Extension 13

Table 1 A comparison of the two Xcode templates for creating a kext 14

Creating a Generic Kernel Extension with Xcode 17

Figure 1 Viewing source code in Xcode 19
Figure 2 MyKext `Info.plist` 20
Figure 3 MyKext `Info.plist` as text 23
Listing 1 MyKext.c 19

Creating a Device Driver With Xcode 27

Figure 1 MyDriver Info.plist 29
Figure 2 `Info.plist` entries after additions 31
Figure 3 Viewing source code in Xcode 32
Figure 4 MyKext `Info.plist` file as text 36
Table 1 `MyDriver` personality dictionary values 30
Listing 1 `MyDriver.h` file contents 32
Listing 2 `MyDriver.cpp` file contents 33

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

Introduction

A **kernel extension** (or **kext**) is a dynamically loaded bundle of executable code that runs in kernel space. You can create a kext to perform low-level tasks that cannot be performed in user space. Kexts typically belong to one of three categories:

- Low-level device drivers
- Network filters
- File systems

This document is a primary resource for kext programming in Mac OS X. It describes the structure of a kext and demonstrates the process for developing a kext, from creating an Xcode project to packaging your kext for distribution.

Who Should Read This Document?

This document is intended for developers who are developing a kernel extension for Mac OS X. Because kext development has numerous pitfalls, you are encouraged to stay away from creating a kext unless you absolutely have to. Read "Deciding Whether to Create a Kernel Extension" (page 11) to make sure a kext is the correct solution for your needs.

If you are developing a driver for a USB or FireWire device, it can and should run in user space. See *USB Device Interface Guide* and *FireWire Device Interface Guide* for details.

Organization of This Document

This document contains the following chapters:

- "Deciding Whether to Create a Kernel Extension" (page 11) explains when it is absolutely necessary to create a kext, along with safer, simpler alternatives for common issues.
- "The Anatomy of a Kernel Extension" (page 13) describes the components of a kext bundle.
- "Creating a Generic Kernel Extension with Xcode" (page 17) guides you through creating a simple generic kext.
- "Creating a Device Driver With Xcode" (page 27) guides you through creating a simple I/O Kit device driver.
- "Debugging a Kernel Extension With GDB" (page 39) guides you through debugging a kernel extension with GDB.
- "Command-Line Tools for Analyzing Kernel Extensions" (page 45) describes command-line tools you can use when working with kexts.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

- "Packaging a Kernel Extension for Distribution and Installation" (page 49) guides you through using the Package Maker application to package your kext.
- "Info.plist Properties for Kernel Extensions" (page 55) describes kext-specific properties for your kext's information property list.

See Also

- *Kernel Programming Guide* provides fundamental high-level information about the Mac OS X core operating-system architecture.
- *I/O Kit Fundamentals* explains the terminology, concepts, architecture, and basic mechanisms of the I/O Kit.
- *I/O Kit Device Driver Design Guidelines* describes common tasks to perform when creating an I/O Kit driver.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

Deciding Whether to Create a Kernel Extension

There are often safer, easier alternatives to creating a kernel extension (kext). It is important to make sure creating a kext is absolutely necessary before doing so.

Make Sure Your Code Needs to Run in Kernel Space

The only reason to write a kext instead of a user-level application or plug-in is to use functionality that is unique to kernel space. The following cases require kernel-resident code:

- The primary client of your code resides in the kernel. File-system and networking device drivers fall into this category.
- Your code needs to handle a primary interrupt (a CPU interrupt generated by hardware). Many device drivers fall into this category: network controllers, graphics drivers, audio drivers, and so on. A USB or FireWire device driver does not require a kext unless its client resides in the kernel.
- A large number of applications require a resource that your code provides.

If your code does not meet any of the above criteria, do not write a kext. Use one of the following user-level solutions instead:

- If you are developing a USB or FireWire device driver, I/O Kit provides an interface for communicating with USB and FireWire devices from user space. See *USB Device Interface Guide* and *FireWire Device Interface Guide*.
- If you are developing a persistent background application that does not require kernel permissions, write a **daemon**. See *System Startup Programming Topics*.

Proceed with Caution

If you have determined that a kext is the proper solution for your issue, keep in mind that developing a kext is riskier and more difficult than developing a user-level application for many reasons, including the following:

- Kexts reduce the memory available to user programs, because kernel-space code requires wired memory (it cannot be paged out).
- The kernel runtime environment has many more restrictions than the user space runtime environment, and they must be followed carefully to avoid errors. See *Kernel Programming Guide* for details.
- Programming errors in a kext are far more severe than bugs in user-level code. Kernel-space code runs in supervisor mode, and it has no protection from memory errors. Consequently, a memory access error in a kext causes a kernel panic, which crashes the operating system.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

- Debugging kexts is more difficult than debugging user-level programs, because it requires two machines and additional steps to set up a debug session.
- For security reasons, some customers restrict the use of third-party kexts.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

The Anatomy of a Kernel Extension

Kexts are loadable bundles, and like all loadable bundles, they are loaded dynamically by another application. In the case of a kext, this application is the kernel itself. This has many implications for kexts, such as running in supervisor mode and the ability to load during early boot. Kexts have strict security and location requirements that you need to follow for your kext to work.

To understand the anatomy of a kext, you should have a basic understanding of bundles. For general information on the structure of a bundle, see *Bundle Programming Guide*.

A Kext Bundle Contains Two Main Components

In the simplest case, a kext bundle contains two components: an information property list and an executable. Along with these required components, a kext bundle may optionally include additional resources and plug-ins. Each of these components is described below.

The Information Property List

A kext's `Info.plist` file describes the kext's contents. Because a kext can be loaded during early boot when limited processing is available, this file must be in XML format and cannot include comments. The following keys are of particular importance in a kext's `Info.plist` file:

- `CFBundleIdentifier` is used to locate a kext both on disk and in the kernel. Multiple kexts with a given identifier can exist on disk, but only one such kext can be loaded in the kernel at a time.
- `CFBundleVersion` indicates the kext's version. Kext version numbers follow a strict pattern (see "Info.plist Properties for Kernel Extensions" (page 55)).
- `OSBundleLibraries` lists the libraries (which are kexts themselves) that the kext links against.
- `IOKitPersonalities` is used by an I/O Kit driver for automatically loading the driver when it is needed.

There are several more kext-specific `Info.plist` keys that allow you to further describe your kext. For a complete discussion of all kext `Info.plist` keys, including keys that refer to kernel-specific runtime facilities, see "Info.plist Properties for Kernel Extensions" (page 55).

The Executable

This is your kext's compiled, executable code. Your executable is responsible for defining entry points that allow the kernel to load and unload the kext. These entry points differ depending on the Xcode template you use when creating your kext. Table 1 describes the default differences between the two kext Xcode

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

templates. This table is intended to illustrate only the most common use of each template; the kernel does not differentiate between kexts created with different templates, and it is possible to incorporate elements of both templates into a kext.

Table 1 A comparison of the two Xcode templates for creating a kext

	Generic kernel extension template	IOKit driver template
Programming language	Usually C	C++
Implementation	C functions registered as callbacks with relevant subsystems	Subclasses of one or more I/O Kit driver classes, such as `IOGraphicsDevice`
Entry points	Start and stop functions with C linkage	C++ static constructors and destructors
Loading behavior	Must be loaded explicitly	Loaded automatically by the I/O Kit when needed
Unloading behavior	Must be unloaded explicitly	Unloaded automatically by the I/O Kit after a fixed interval when no longer needed
Tutorial	"Creating a Generic Kernel Extension with Xcode" (page 17)	"Creating a Device Driver With Xcode" (page 27)

Additional Resources and Plug-ins

Kexts sometimes require additional resources, such as firmware for a device. If your kext requires a resource, put it in the `Resources` folder of your kext's bundle. If you plan to localize your resources, keep in mind that kernel-space code does not detect localized resources. User-space code *does* detect localized resources in `.lproj` subfolders of the `Resources` folder, so if your resource is accessed *only* by user-space code, localization is straightforward.

In addition to general resources, kexts can contain plug-ins, including other kexts. If your kext uses a plug-in, put it in the `PlugIns` folder of your kext's bundle. Make sure that plug-in kexts do not contain plug-in kexts of their own; only one level of plug-ins is detected in order to limit file system traversal during early boot.

Kernel Extensions Have Strict Security Requirements

Kexts execute in kernel space and run in supervisor mode; consequently, files and folders in a kext bundle must be owned by the `root` user and the `wheel` group. Files must have the permissions `0644`, and folders must have the permissions `0755`. A kext that fails to meet these requirements will not load into the kernel.

During development, to ensure that your kext has the proper ownership and permissions, create a copy of your kext as the root user.

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


2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

This method requires creating a new copy of the kext every time you build it.

Kernel Extensions Should Reside in /System/Library/Extensions

Mac OS X looks up a kext by its `CFBundleIdentifier` information property list key. Kexts located in `/System/Library/Extensions`, and the plug-in kexts of those kexts, are searched by default. You can perform a custom search to locate kexts in other folders, but this approach is not recommended. If your kext needs to be loaded during boot loading, it *must* be installed in `/System/Library/Extensions` for the operating system to locate it.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

Creating a Generic Kernel Extension with Xcode

In this tutorial, you learn how to create a generic kernel extension (kext) for Mac OS X. You'll create a simple kext that prints messages when loading and unloading. This tutorial does not cover the process for loading or debugging your kext—see "Debugging a Kernel Extension With GDB" (page 39) after you have completed this tutorial for information on loading and debugging.

If you are unfamiliar with Xcode, first read *A Tour of Xcode*.

Road Map

These are the four major steps you will follow:

1. "Create a New Project" (page 17)
2. "Implement the Start and Stop Functions" (page 18)
3. "Add Library Declarations" (page 21)
4. "Prepare the Kernel Extension for Loading" (page 24)

This tutorial assumes that you are logged in as an administrator of your machine, which is necessary for using the `sudo` command.

Create a New Project

Creating a kext project in Xcode is as simple as selecting the appropriate project template and providing a name:

1. Launch Xcode.
2. Choose File > New Project. The New Project panel appears.
3. In the New Project panel, pick the appropriate project category and Xcode template.

   In the list of project categories on the left, select System Plug-in.

   In the list of templates on the right, select Generic Kernel Extension.

   Click the Choose button.
4. In the save sheet that appears, enter the project name and save the project.

   Enter MyKext as the project name.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

Choose a location for the project, and click the Save button.

When you click Save, Xcode creates the new project and displays its project window.

The new project contains several files, including a source file, `MyKext.c`, that contains templates for the kext's start and stop functions.

5. Make sure the kext is building for all architectures.

   Click the disclosure triangle next to Targets in the Groups and Files pane.

   Select the MyKext target.

   Choose File > Get Info. The Target "MyKext" Info window opens.

   In the list of settings, find Build Active Architecture Only and make sure the checkbox is unchecked.

   Close the Target "MyKext" Info window.

Implement the Start and Stop Functions

Now that you've created the project, it's time to make your kext do something when it gets loaded and unloaded. You'll do so by adding code to your kext's start and stop functions, which are called when your kext is loaded and unloaded.

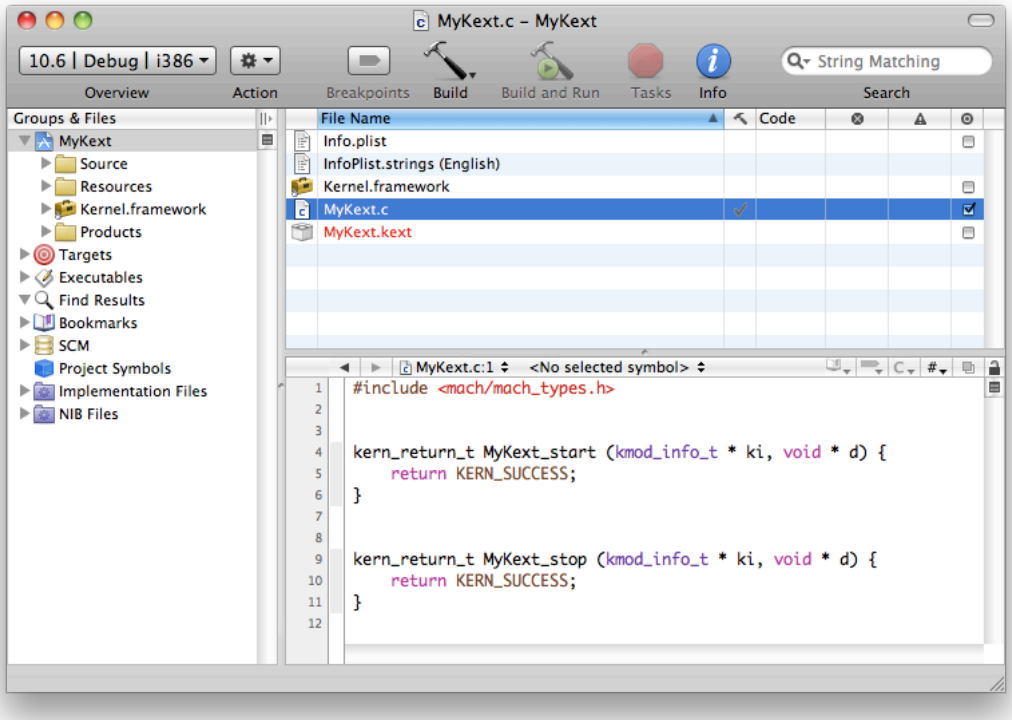
Implement the Start and Stop Functions

1. Open `MyKext.c` to edit the start and stop functions.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

Figure 1 (page 19) shows the unedited file.

Figure 1 Viewing source code in Xcode



The default start and stop functions do nothing but return a successful status. A real kext's start and stop functions typically register and unregister callbacks with kernel runtime systems, but for this tutorial, your kext simply prints messages so that you can confirm when your kext has been started and stopped.

2. Edit `MyKext.c` to match the code in Listing 1.

Listing 1 MyKext.c

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

kern_return_t MyKext_start (kmod_info_t * ki, void * d)
{
    printf("MyKext has started.\n");
    return KERN_SUCCESS;
}

kern_return_t MyKext_stop (kmod_info_t * ki, void * d)
{
    printf("MyKext has stopped.\n");
    return KERN_SUCCESS;
}
```


2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

Notice that `MyKext.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 to include only 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 kernel extension might compile, but it will probably fail to load or run because the functions and services those headers define are not available in the kernel.

3. Save your changes by choosing File > Save.

Edit the Information Property List

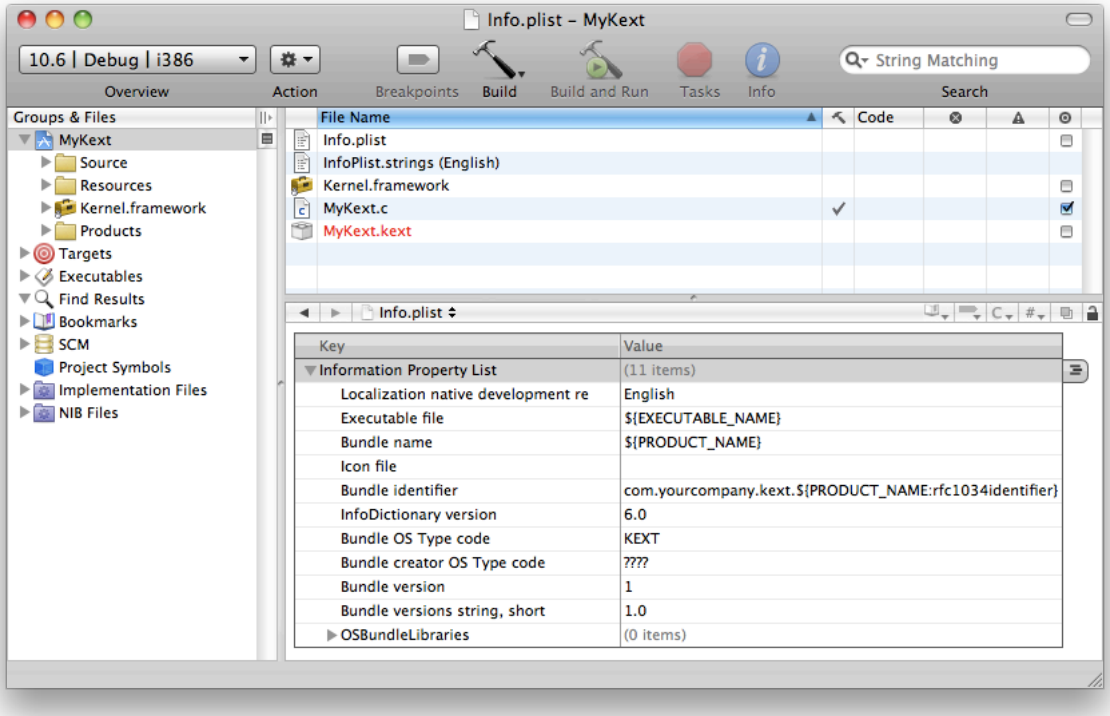
Like all bundles, a kext contains an information property list, which describes the kext. The default `Info.plist` file created by Xcode contains template values that you must edit to describe your kext.

A kext's `Info.plist` file is in XML format. Whenever possible, you should view and edit the file from within Xcode or within the Property List Editor application. In this way, you help ensure that you don't add elements (such as comments) that cannot be parsed by the kernel during early boot.

1. Click Info.plist in the Xcode project window.

   Xcode displays the `Info.plist` file in the editor pane. You should see the elements of the property list file, as shown in Figure 2.

Figure 2 MyKext `Info.plist`



2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

By default, Xcode's property list editor masks the actual keys and values of a property list. To see the actual keys and values, Control-click anywhere in the property list editor and choose Show Raw Keys/Values from the contextual menu.

2. Change the value of the `CFBundleIdentifier` property to use your unique namespace prefix.

   On the line for `CFBundleIdentifier`, double-click in the Value column to edit it. Select `com.yourcompany` and change it to `com.MyCompany` (or your company's DNS domain in reverse). The value should now be `com.MyCompany.kext.${PRODUCT_NAME:rfc1034identifier}`.

   Bundles in Mac OS X typically use a reverse-DNS naming convention to avoid namespace collisions. This is particularly important for kexts because all loaded kexts share a single namespace for bundle identifiers.

   The last portion of the default bundle identifier, `${PRODUCT_NAME:rfc1034identifier}`, is replaced with the Product Name build setting for the kext target when you build your project.

3. Save your changes by choosing File > Save.

Build the Kernel Extension

Now you're ready to configure your build settings and build your kext to make sure the source code compiles. First, configure your build settings to build the kext for every architecture, to make sure your kext will load regardless of the architecture of the kernel.

1. Click the disclosure triangle next to Targets in the Groups and Files pane.
2. Select the MyKext target.
3. Choose File > Get Info. The Target "MyKext" Info window opens.
4. In the list of settings, find Build Active Architecture Only and make sure the checkbox is unchecked.
5. Close the Target MyKext Info window.

Now that your kext is building against every architecture, choose Build > Build to build your project. If the build fails, correct all indicated errors and rebuild before proceeding.

Add Library Declarations

Because kexts are linked at load time, a kext must list its libraries in its information property list with the `OSBundleLibraries` property. At this stage of creating your kext, you need to find out what those libraries are. The best way to do so is to run the `kextlibs` tool on your built kext and copy its output into your kext's `Info.plist` file.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

Run kextlibs on the Kernel Extension

`kextlibs` is a command-line program that you run with the Terminal application. Its purpose is to identify libraries that your kext needs to link against.

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

1. Start the Terminal application, located in `/Applications/Utilities`.
2. In the Terminal window, move to the directory that contains your kext.

   Xcode stores your kext in the `Debug` folder of the `build` folder of your project (unless you've chosen a different build configuration or set a different location for build products using Xcode's Preferences dialog).

   ```
   $ cd MyKext/build/Debug
   ```

   This directory contains your kext. It should have the name `MyKext.kext`. This name is formed from the Product Name as set in your target's build settings, and a suffix, in this case `.kext`.
3. Run `kextlibs` on your kernel extension with the `-xml` command-line flag.

   This command looks for all unresolved symbols in your kernel extension's executable among the installed library extensions (in `/System/Library/Extensions/`) and prints an XML fragment suitable for pasting into an `Info.plist` file. For example:

   ```
   $ kextlibs -xml MyKext.kext
           <key>OSBundleLibraries</key>
           <dict>
                   <key>com.apple.kpi.libkern</key>
                   <string>10.2</string>
           </dict>
   ```
4. Make sure `kextlibs` exited with a successful status by checking the shell variable `$?`.

   ```
   $ echo $?
   0
   ```

   If `kextlibs` prints any errors or exits with a nonzero status, it may have been unable to locate some symbols. For this tutorial, the libraries are known, but in general usage you should use the `kextfind` tool to find libraries for any symbols that `kextlibs` cannot locate. See "Locate Kexts with kextfind" (page 46) for information on `kextfind`.
5. Select the XML output of `kextlibs` and choose Edit > Copy.

Add the Library Declarations to the Information Property List

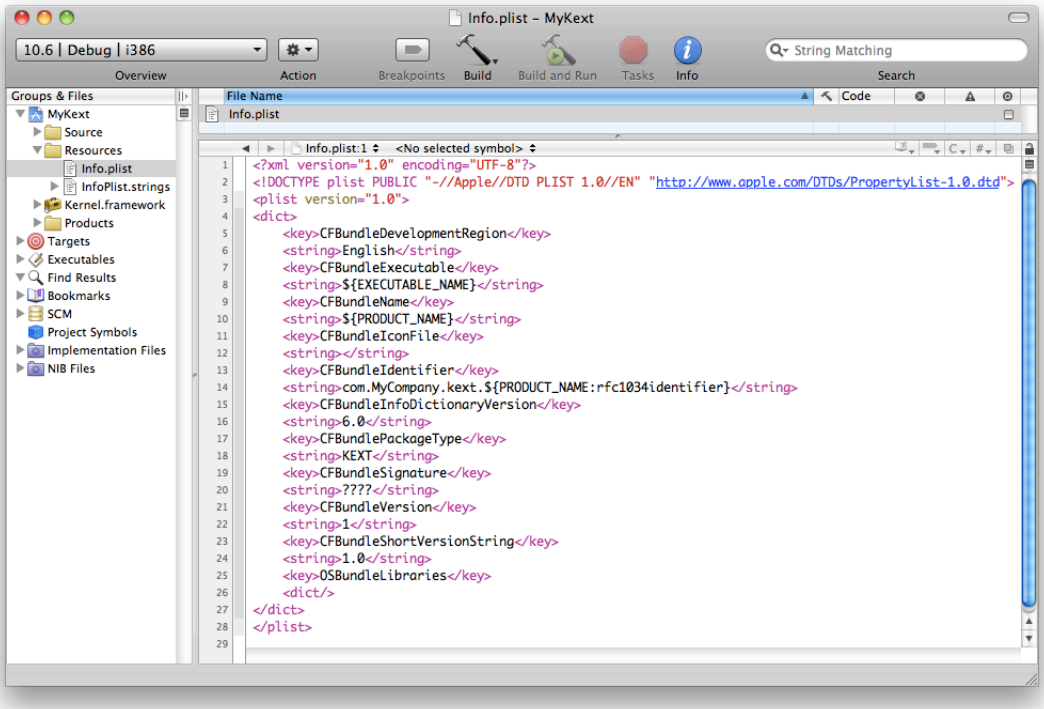
Earlier, you edited the information property list with Xcode's graphical property list editor. For this operation, however, you need to edit the information property list as text.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

1. Control-click Info.plist in the Xcode project window, then choose Open As > Source Code File from the contextual menu.

   Xcode displays the Info.plist file in the editor pane. You should see the XML contents of the property list file, as shown in Figure 3. Note that dictionary keys and values are listed sequentially.

Figure 3 MyKext `Info.plist` as text



2. Select all the lines defining the empty OSBundleLibraries dictionary:

```
<key>OSBundleLibraries</key>
<dict/>
```

3. Paste text into the info dictionary.

   If `kextlibs` ran successfully, choose Edit > Paste to paste the text you copied from Terminal.

   If `kextlibs` didn't run successfully, type or paste this text into the info dictionary:

```
<key>OSBundleLibraries</key>
<dict>
        <key>com.apple.kpi.libkern</key>
        <string>10.0</string>
</dict>
```

4. Save your changes by choosing File > Save.

5. Choose Build > Build a final time to rebuild your kext with the new information property list.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

Prepare the Kernel Extension for Loading

Now you are ready to prepare your kext for loading. You'll do this with the `kextutil` tool, which can examine a kext and determine whether it is able to be loaded. `kextutil` can also load a kext for development purposes, but that functionality is not covered in this tutorial.

Note: This tutorial does not cover actually loading a kext. For safety reasons, you should not load your kext on your development machine. For information on loading and debugging a kext with a two-machine setup, see "Debugging a Kernel Extension With GDB" (page 39).

Set the Kernel Extension's Permissions

Kexts have strict permissions requirements (see "Kernel Extensions Have Strict Security Requirements" (page 14) for details). The easiest way to set these permissions is to create a copy of your kext as the `root` user. Type the following into Terminal from the proper directory and provide your password when prompted:

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

Now that the permissions of the kext's temporary copy are correct, you are ready to run `kextutil`.

Run kextutil

Type the following into Terminal:

```
$ kextutil -n -print-diagnostics /tmp/MyKext.kext
```

The `-n` (or `-no-load`) option tells `kextutil` not to load the driver, and the `-t` (or `-print-diagnostics`) option tells `kextutil` to print the results of its analysis to Terminal. If you have followed the previous steps in this tutorial correctly, `kextutil` indicates that the kext is loadable and properly linked.

```
No kernel file specified; using running kernel for linking.
MyKext.kext appears to be loadable (including linkage for on-disk libraries).
```


2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

Note: You may encounter an error similar to the following:

```
Warnings:
    Executable does not contain code for architecture:
        i386
```

If you do, make sure you set your kext to build for all architectures, as described in "Create a New Project" (page 17).

Where to Go Next

Congratulations! You have now written, built, and prepared your own kext for loading. In the next tutorial in this series, "Creating a Device Driver With Xcode" (page 27), you'll learn how to create an I/O Kit driver, a kext that allows the kernel to interact with devices. If you are not creating an I/O Kit Driver, you can move directly on to "Debugging a Kernel Extension With GDB" (page 39), in which you learn how to load a kext, debug it, and unload it with a two-machine setup.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

Creating a Device Driver With Xcode

In this tutorial, you'll learn how to create an I/O Kit device driver for Mac OS X. You'll create a simple driver that prints text messages, but doesn't actually control a device. This tutorial does not cover the process for loading or debugging your driver—see "Debugging a Kernel Extension With GDB" (page 39) after you have completed this tutorial for information on loading and debugging.

If you are unfamiliar with Xcode, first read *A Tour of Xcode*.

Road Map

Here are the major steps you will follow:

1. "Familiarize Yourself with the I/O Kit Architecture" (page 27)
2. "Create a New Project" (page 28)
3. "Edit the Information Property List" (page 28)
4. "Fill in the Header File" (page 31)
5. "Implement the Driver's Entry Points" (page 33)
6. "Add Library Declarations" (page 34)
7. "Prepare the Driver for Loading" (page 37)

This tutorial assumes that you are logged in as an administrator of your machine, which is necessary for using the `sudo` command.

Familiarize Yourself with the I/O Kit Architecture

Every I/O Kit driver is based on an I/O Kit **family**, a collection of C++ classes that implement functionality that is common to all devices of a particular type. Examples of I/O Kit families include storage devices (disks), networking devices, and human-interface devices (such as keyboards).

An I/O Kit driver communicates with the device it controls through a **provider** object, which typically represents the bus connection for the device. Provider objects that do so are referred to as **nubs**.

An I/O Kit driver is loaded into the kernel automatically when it **matches** against a device that is represented by a nub. A driver matches against a device by defining one or more **personalities**, descriptions of the types of device the driver can control.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

After an I/O Kit driver matches against a device and loads into the kernel, it routes I/O for the device, as well as vending **services** related to the device, such as providing a firmware update mechanism.

Before you begin creating your own driver, you should make sure you understand the architecture of the I/O Kit by reading "Architectural Overview" in *I/O Kit Fundamentals*.

Create a New Project

Creating an I/O Kit driver project in Xcode is as simple as selecting the appropriate project template and providing a name.

1. Launch Xcode.
2. Choose File > New Project. The New Project panel appears.
3. In the New Project panel, pick the appropriate project category and Xcode template.

   In the list of project categories on the left, select System Plug-in.

   In the list of templates on the right, select IOKit Driver.

   Click the Choose button.
4. In the save sheet that appears, enter the project name and save the project.

   Enter MyDriver as the project name.

   Choose a location for the project, and click the Save button.

   When you click Save, Xcode creates the new project and displays its project window.

   The new project contains several files, including a source file, `MyDriver.cpp`, which contains no code.
5. Make sure the driver is building for all architectures.

   Click the disclosure triangle next to Targets in the Groups and Files pane.

   Select the MyDriver target.

   Choose File > Get Info. The Target "MyDriver" Info window opens.

   In the list of settings, find Build Active Architecture Only and make sure the checkbox is unchecked.

   Close the Target "MyDriver" Info window.

Edit the Information Property List

Like all bundles, a device driver contains an information property list, which describes the driver. The default `Info.plist` file created by Xcode contains template values that you must edit to describe your driver.

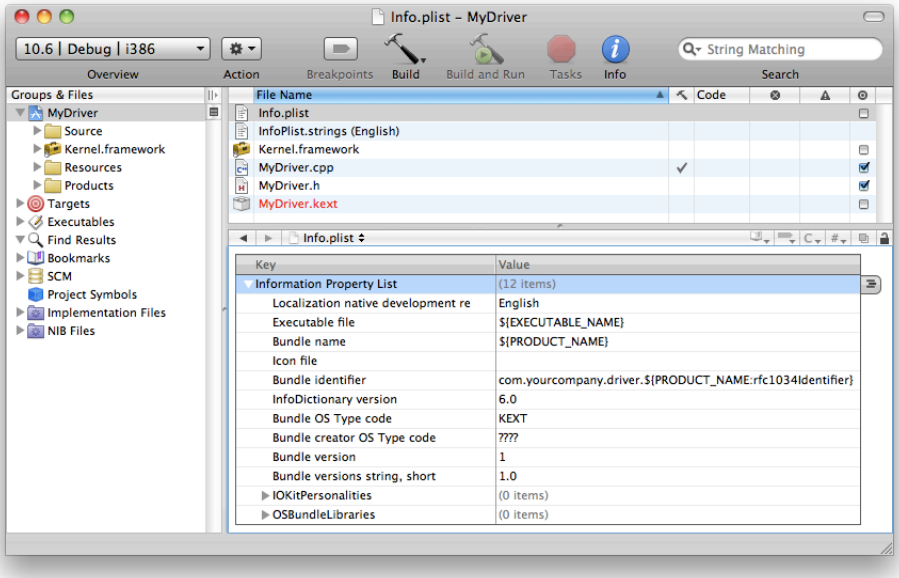
2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

A device driver's `Info.plist` file is in XML format. Whenever possible, you should view and edit the file from within Xcode or within the Property List Editor application. In this way, you help ensure that you don't add elements (such as comments) that cannot be parsed by the kernel during early boot.

1. Click Info.plist in the Xcode project window.

   Xcode displays the `Info.plist` file in the editor pane. You should see the elements of the property list file, as shown in Figure 1.

   Figure 1 MyDriver Info.plist



   By default, Xcode's property list editor masks the actual keys and values of a property list. To see the actual keys and values, Control-click anywhere in the property list editor and choose Show Raw Keys/Values from the contextual menu.

2. Change the value of the `CFBundleIdentifier` property to use your unique namespace prefix.

   On the line for `CFBundleIdentifier`, double-click in the Value column to edit it. Select `com.yourcompany` and change it to `com.MyCompany` (or your company's DNS domain in reverse). The value should now be `com.MyCompany.driver.${PRODUCT_NAME:rfc1034identifier}`.

   Bundles in Mac OS X typically use a reverse-DNS naming convention to avoid namespace collisions. This convention is particularly important for kexts, because all loaded kexts share a single namespace for bundle identifiers.

   The last portion of the default bundle identifier, `${PRODUCT_NAME:rfc1034identifier}`, is replaced with the Product Name build setting for the driver target when you build your project.

3. Add a personality to your driver's `IOKitPersonalities` dictionary.

   Click the `IOKitPersonalities` property to select it, then click its disclosure triangle so that it points down.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

Click the New Child symbol on the right side of the selected line. A property titled `New item` appears as a child of the `IOKitPersonalities` property. Change the name of `New item` to `MyDriver`.

Make the `MyDriver` item a dictionary. Control-click it and choose Value Type > Dictionary from the contextual menu.

Your device driver requires one or more entries in the `IOKitPersonalities` dictionary of its information property list. This dictionary defines properties used for matching your driver to a device and loading it.

4. Fill in the personality dictionary.

Create a child entry for the `MyDriver` dictionary. Rename the child from `New item` to `CFBundleIdentifier`. Copy and paste the value from the property list's top-level `CFBundleIdentifier` value (`com.MyCompany.driver.${PRODUCT_NAME:rfc1034identifier}`) as the value.

Create a second child for the `MyDriver` dictionary. Rename the child to `IOClass`. Enter `com_MyCompany_driver_MyDriver` as the value. Note that this is the same value as for the `CFBundleIdentifier`, except it separates its elements with underbars instead of dots. This value is used as the class name for your device driver.

Create a third child for the `MyDriver` dictionary. Rename the child to `IOKitDebug`. Enter `65535` as the value and change the value type from String to Number. If you specify a nonzero value for this property, your driver provides useful debugging information when it matches and loads. When you build your driver for public release, you should specify `0` as the value for this property or remove it entirely.

Create two more children for the `MyDriver` dictionary. Assign their names and values according to Table 1.

Table 1 `MyDriver` personality dictionary values

Name	Value
`IOProviderClass`	`IOResources`
`IOMatchCategory`	`com_MyCompany_driver_MyDriver`

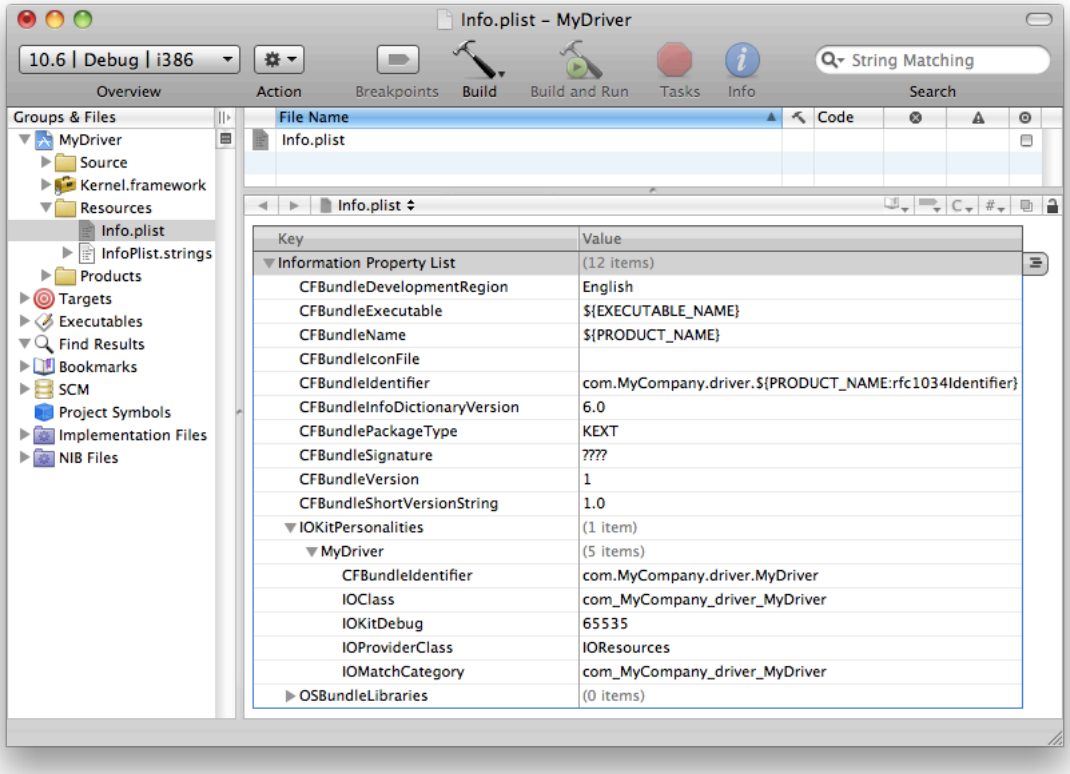
These elements together define a successful match for your driver, so that it can be loaded. They serve the following purposes:

- `IOProviderClass` indicates the class of the provider objects that your driver can match on. Normally a device driver matches on the nub that controls the port that your device is connected to. For example, if your driver connects to a PCI bus, you should specify `IOPCIDevice` as your driver's provider class. In this tutorial, you are creating a virtual driver with no device, so it matches on `IOResources`.
- `IOMatchCategory` allows other drivers to match on the same device as your driver, as long as the drivers' values for this property differ. This tutorial's driver matches on `IOResources`, a special provider class that provides system-wide resources, so it needs to include this property to allow other drivers to match on `IOResources` as well. When you develop your driver, you should not include this property unless your driver matches on a device that another driver may match on, such as a serial port with multiple devices attached to it.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

When you have finished adding property list elements, the list should look like the example shown in Figure 2.

Figure 2 `Info.plist` entries after additions



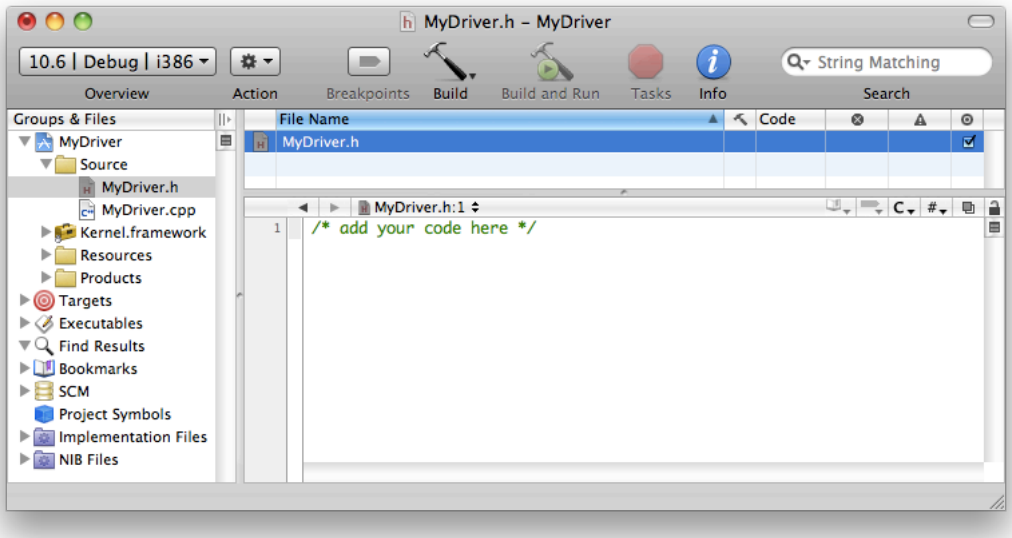
5. Choose File > Save to save your changes.

Fill in the Header File

Open `MyDriver.h` in your project's `Source` folder. The default header file contains no code. Figure 3 shows where to find the `MyDriver.h` file in the project window.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

Figure 3 Viewing source code in Xcode



Edit the contents of `MyDriver.h` to match the code in Listing 1.

Listing 1 `MyDriver.h` file contents

```
#include <IOKit/IOService.h>
class com_MyCompany_driver_MyDriver : public IOService
{
OSDeclareDefaultStructors(com_MyCompany_driver_MyDriver)
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);
};
```

Notice that the first line of `MyDriver.h` includes the header file `IOService.h`. This header file defines many of the methods and services that device drivers use. The header file is located in the `IOKit` folder of `Kernel.framework`. When you develop your own driver, be sure to include only 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 it fails to load because the functions and services defined in those header files are not available in the kernel.

Note that when you are developing your own driver, you should replace instances of `com_MyCompany_driver_MyDriver` with the name of your driver's class.

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

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

Implement the Driver's Entry Points

1. Open `MyDriver.cpp` in your project's `Source` folder. The default file contains no code.
2. Edit `HelloIOKit.cpp` to match the code in Listing 2.

Listing 2 `MyDriver.cpp` file contents

```
#include <IOKit/IOLib.h>
#include "MyDriver.h"

// This required macro defines the class's constructors, destructors,
// and several other methods I/O Kit requires.
OSDefineMetaClassAndStructors(com_MyCompany_driver_MyDriver, IOService)

// Define the driver's superclass.
#define super IOService

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

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

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

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

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


2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

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 required by the I/O Kit.

This listing includes the entry point methods that the I/O Kit uses to access your driver. These entry points serve the following purposes:

- The `init` method is the first instance method called on each instance of your driver class. It is called only once on each instance. The `free` method is the last method called on any object. Any outstanding resources allocated by the driver should be disposed of in `free`. Note that the `init` method operates on objects generically; it should be used only to prepare objects to receive calls. Actual driver functionality should be set up in the `start` method.
- The `probe` method is called if your driver needs to communicate with hardware to determine whether there is a match. This method must leave the hardware in a good state when it returns, because other drivers may probe the hardware as well.
- The `start` method tells the driver to start driving hardware. After `start` is called, the driver can begin routing I/O, publishing nubs, and vending services. The `stop` method is the first method to be called before your driver is unloaded. When `stop` is called, your driver should clean up any state it created in its `start` method. The `start` and `stop` methods talk to the hardware through your driver's provider class.

The `IOLog` function is the kernel equivalent of `printf` for an I/O Kit driver.

3. Save your changes by choosing File > Save.
4. Build your project by choosing Build > Build. Fix any compiler errors before continuing.

Add Library Declarations

Because kexts are linked at load time, a kext must list its libraries in its information property list with the `OSBundleLibraries` property. At this stage of creating your driver, you need to find out what those libraries are. The best way to do so is to run the `kextlibs` tool on your built kext, and copy its output into your kext's `Info.plist` file.

Run kextlibs on the Driver

`kextlibs` is a command-line program that you run with the Terminal application. Its purpose is to identify libraries that your kext needs to link against.

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

1. Start the Terminal application, located in `/Applications/Utilities`.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

2. In the Terminal window, move to the directory that contains your driver.

   Xcode stores your driver in the `Debug` folder of the `build` folder of your project (unless you've chosen a different build configuration or set a different location for build products using the Xcode Preferences dialog):

   ```
   $ cd MyDriver/build/Debug
   ```

   This directory contains your driver. It should have the name `MyDriver.kext`. This name is formed from the Product Name, as set in your target's build settings, and a suffix, in this case `.kext`.

3. Run `kextlibs` on your driver with the `-xml` command-line flag.

   This command looks for all unresolved symbols in your kernel extension's executable among the installed library extensions (in `/System/Library/Extensions/`) and prints an XML fragment suitable for pasting into an `Info.plist` file. For example:

   ```
   $ kextlibs -xml MyDriver.kext
           <key>OSBundleLibraries</key>
           <dict>
                   <key>com.apple.kpi.iokit</key>
                   <string>10.2</string>
                   <key>com.apple.kpi.libkern</key>
                   <string>10.2</string>
           </dict>
   ```

4. Make sure `kextlibs` exited with a successful status by checking the shell variable `$?`.

   ```
   $ echo $?
   0
   ```

   If `kextlibs` prints any errors or exits with a nonzero status, it may have been unable to locate some symbols. For this tutorial, the libraries are known, but in general usage you should use the `kextfind` tool to find libraries for any symbols that `kextlibs` cannot locate. See "Locate Kexts with kextfind" (page 46).

5. Select the XML output of `kextlibs` and choose Edit > Copy.

Add the Library Declarations to the Information Property List

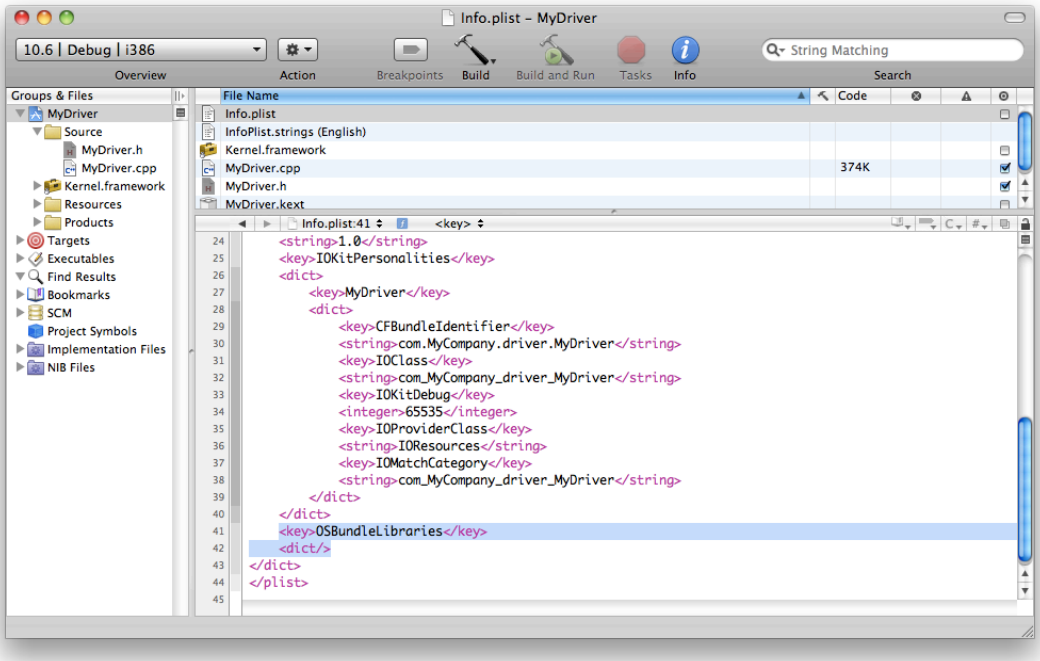
Earlier you edited the information property list with Xcode's graphical property list editor. For this operation, however, you need to edit the information property list as text.

1. Control-click Info.plist in the Xcode project window, then choose Open As > Source Code File from the contextual menu.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

Xcode displays the Info.plist file in the editor pane. You should see the XML contents of the property list file, as shown in Figure 3. Note that dictionary keys and values are listed sequentially.

Figure 4 MyKext `Info.plist` file as text



2. Select all the lines defining the empty OSBundleLibraries dictionary.

```
<key>OSBundleLibraries</key>
<dict/>
```

3. Paste text into the info dictionary.

   If `kextlibs` ran successfully, choose Edit > Paste to paste the text you copied from Terminal.

   If `kextlibs` didn't run successfully, type or paste this text into the info dictionary:

```
<key>OSBundleLibraries</key>
<dict>
        <key>com.apple.kpi.iokit</key>
        <string>10.2</string>
        <key>com.apple.kpi.libkern</key>
        <string>10.2</string>
</dict>
```

4. Save your changes by choosing File > Save.

5. Rebuild your driver (choose Build > Build) with the new information property list. Fix any compiler errors before continuing.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

Prepare the Driver for Loading

Now you are ready to prepare your driver for loading. You'll do this with the `kextutil` tool, which can examine a kext and determine whether it is able to be loaded. `kextutil` can also load a kext for development purposes, but that functionality is not covered in this tutorial.

Note: This tutorial does not cover loading your driver. For safety reasons, you should not load your driver on your development machine. For information on loading and debugging a kext with a two-machine setup, see "Debugging a Kernel Extension With GDB" (page 39).

Set the Driver's Permissions

Kexts have strict permissions requirements (see "Kernel Extensions Have Strict Security Requirements" (page 14) for details). The easiest way to set these permissions is to create a copy of your driver as the `root` user. Type the following into Terminal from the proper directory and provide your password when prompted:

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

Now that the permissions of the driver's temporary copy are correct, you are ready to run `kextutil`.

Run kextutil

Type the following into Terminal:

```
$ kextutil -n -t /tmp/MyDriver.kext
```

The `-n` (or `-no-load`) option tells `kextutil` not to load the driver, and the `-t` (or `-print-diagnostics`) option tells `kextutil` to print the results of its analysis to Terminal. If you have followed the previous steps in this tutorial correctly, `kextutil` indicates that the kext is loadable and properly linked.

```
No kernel file specified; using running kernel for linking.
Notice: /tmp/MyDriver.kext has debug properties set.
MyDriver.kext appears to be loadable (including linkage for on-disk libraries).
```

Note: You may encounter an error similar to the following:

```
Warnings:
    Executable does not contain code for architecture:
        i386
```

If you do, make sure you set your kext to build for all architectures, as described in "Create a New Project" (page 28).

The debug property notice is due to the nonzero value of the `IOKitDebug` property in the information property list. Make sure you set this property to zero or remove it when you build your driver for release.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

Where to Go Next

Congratulations! You have now written, built, and prepared your own driver for loading. In the next tutorial in this series, "Debugging a Kernel Extension With GDB" (page 39), you'll learn how to load your kext, debug it, and unload it with a two-machine setup.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

Debugging a Kernel Extension With GDB

In this tutorial, you learn how to debug a kext. You'll set up a two-machine debugging environment and use GDB to perform remote debugging. If you have not yet created a kext, complete "Creating a Generic Kernel Extension with Xcode" (page 17) or "Creating a Device Driver With Xcode" (page 27) before completing this tutorial. If you are unfamiliar with GDB, see *Debugging with GDB*.

Although this tutorial is written with a device driver as the example, the steps for debugging are similar for debugging any type of kext.

Road Map

You need two machines for remote debugging: a **target machine** and a **development machine**. You load and run the kext on the target machine and debug the kext on the development machine. It is important to keep the two machines clear in your mind as you work through this tutorial, because you will be moving back and forth between them many times. 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.

These are the major steps you will follow:

1. "Prepare the Machines" (page 40)
2. "On Development Machine, Sabotage the Kernel Extension" (page 40)
3. "On Target Machine, Enable Kernel Debugging" (page 41)

   After you have completed the first three steps, you do not need to repeat them, even if you need to start over part way through the tutorial. The remaining steps prepare the kext for debugging, set up GDB, and attach the two machines to begin debugging your kext. If you need to start over part way through the tutorial, you should repeat all of the remaining steps.
4. "On Target Machine, Get the Target Machine's IP Address" (page 41)
5. "On Development Machine, Start GDB" (page 41)
6. "On Target Machine, Load the Kernel Extension" (page 42)
7. "On Development Machine, Attach to the Target Machine" (page 42)
8. "On Development Machine, Get the Load Address of the Kernel Extension" (page 42)
9. "On Development Machine, Create and Load the Symbol File" (page 42)
10. "On Development Machine, Debug with GDB" (page 43)
11. "On Development Machine, Stop the Debugger" (page 44)

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

Prepare the Machines

Prepare the two machines by doing the following:

1. Ensure that the machines are running the same version of Mac OS X.
2. Ensure that the machines are connected to the same network with their built-in Ethernet ports.
3. Ensure that you are logged in as an administrator on both machines, which is necessary for using the `sudo` command.
4. Mount the Kernel Debug Kit disk image on the development machine. Download the Kernel Debug Kit from the Apple Developer website under the Mac OS X download category. Make sure the Kernel Debug Kit you download matches the version of Mac OS X installed on your target machine.

   For more information on the Kernel Debug Kit, see the Read Me file included in the disk image.
5. If your target machine is running Mac OS X Server, disable the Mac OS X Server watchdog timer.

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

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

On the Development Machine, Sabotage the Kernel Extension

To better simulate a real-world kext debugging scenario, you need your kext to produce a kernel panic. The easiest way to do this is to dereference a null pointer.

1. In Xcode, add the following code to your driver's `start` method (if you are debugging a generic kext, add it to the kext's `MyKext_start` function):

   ```
   char *kernel_panic = NULL;
   char message = *kernel_panic;
   ```
2. Rebuild your kext. In the Terminal application, create a copy of the kext as root:

   ```
   $ sudo cp -R MyDriver.kext /tmp
   ```
3. Copy your kext's `dSYM` file (in the Xcode project's build folder with your kext) to the same location you copy your kext.
4. Transfer the copy of your kext from the development machine to the target machine.

   If the transferred copy of your kext has an incorrect owner or group, correct it with the following command:

   ```
   $ sudo chown -R root:wheel MyDriver.kext
   ```

Warning: Make sure that you do *not* put the kext in `/System/Library/Extensions`. If you do, the kext is loaded every time you reboot.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

On the Target Machine, Enable Kernel Debugging

Before you can debug your kext, you must first enable kernel debugging. On the target machine, do the following:

1. Start the Terminal application.
2. Set the kernel debug flags.

   To enable kernel debugging, you must set an NVRAM (nonvolatile random access memory) variable:

   ```
   $ sudo nvram boot-args="debug=0x144 -v"
   Password:
   ```

   For more information on debugging flags, see "Building and Debugging Kernels" in *Kernel Programming Guide*.
3. Restart the computer for the debugging flags to take effect.

On the Target Machine, Get the Target Machine's IP Address

> **Note:** If you need to restart the tutorial at any point after this step, begin with this step.

To connect to the target machine from the development machine, you need the target machine's IP address. If you don't already know it, you can find it in the Network pane of the System Preferences application.

On the Development Machine, Start GDB

1. Start GDB with the following command, indicating the target machine's architecture and the location of your debug kernel:

   ```
   $ gdb -arch i386 /Volumes/KernelDebugKit/mach_kernel
   ```
2. Add kernel-specific macros from the Kernel Debug Kit to your GDB session.

   ```
   (gdb) source /Volumes/KernelDebugKit/kgmacros
   ```
3. Inform GDB that you are going to be debugging a kext remotely.

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


2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

On the Target Machine, Load the Kernel Extension

You are ready to load your kext and cause a kernel panic. Do so with the following command:

```
$ sudo kextutil MyDriver.kext
```

The kernel panic should occur immediately. Interactivity ceases and debugging text appears on the screen, including the text `Awaiting debugger connection`.

On the 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 `kdp-reattach` macro with the target machine's name or IP address:

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

   The target machine prints:

   ```
   Connected to remote debugger.
   ```

On the Development Machine, Get the Load Address of the Kernel Extension

You need your kext's load address in order to generate a symbol file for it. Enter the following at the GDB prompt:

```
(gdb) showallkmods
```

A list appears displaying information about every kext running on the target machine. Find your kext in the list and write down the value in the address column. Note that the values in the kmod and size columns look similar to the address value, so make sure you have the correct value.

On the Development Machine, Create and Load the Symbol File

You can create a symbol file for your kext on the development machine with the `kextutil` command. Once again, make sure that the version of the Kernel Debug Kit you provide matches the version of Mac OS X on the target machine.

1. Open a second Terminal window.
2. Create the symbol file.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

Adjust the path to the Kernel Debug Kit, the path to your kext, and the architecture of the kernel as appropriate. The path after the `-s` option specifies the output directory where the symbol file is written. This should be the directory you copied your kext and `dSYM` file to. The `-n` option prevents the command from loading the kext into the kernel.

```
sudo kextutil -s /tmp -n -arch i386 -k /Volumes/KernelDebugKit/mach_kernel -e
-r /Volumes/KernelDebugKit /tmp/MyDriver.kext
```

The `kextutil` tool prompts you for the load address of your kext. Provide the load address you obtained in the previous step.

When you finish, the symbol file is in the output directory you specified. The filename is the bundle identifier of the kext with the `.sym` extension.

3. At the GDB prompt, specify the location of the symbol file.

   Again, make sure the symbol file is in the same folder as the copy of your kext and `dSYM` file.

   ```
   (gdb) set kext-symbol-file-path /tmp
   ```

4. At the GDB prompt, add your kext to the debug environment with the following macro:

   ```
   (gdb) add-kext /tmp/MyDriver.kext
   ```

   GDB asks you if you want to add the kext's symbol file. When you confirm, it loads the symbols.

On the Development Machine, Debug with GDB

Now you are ready to begin debugging! Request a backtrace from GDB to locate the source of the panic:

```
(gdb) bt
```

A list of stack frames appears. Your kext's stack frame that caused the panic should be easily identifiable as about the fifth frame from the top. When you are debugging your own kernel panics and don't know the cause, you can now enter the offending stack frame and figure out what exactly caused the panic.

Note: Because driver debugging happens at such a low level, you can't use some GDB features, including the following:

- 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 cause the target kernel to panic or hang, forcing you to reboot the target machine.

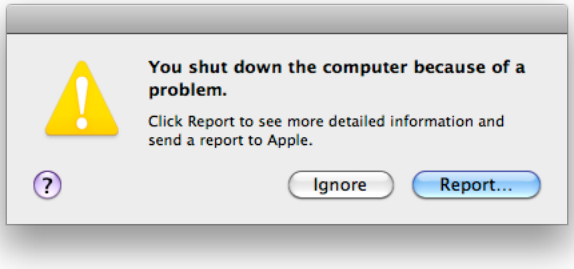
2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

On the Development Machine, Stop the Debugger

When you've finished debugging, stop the debugger by quitting GDB.

```
(gdb) quit
```

The debugging session ends. Because the target machine is still panicked, you need to reboot it. When you log back into the target machine, it displays the following message:



Click Ignore.

Where to Go Next

Congratulations! You've learned how to set up a two-machine debugging environment to debug a kext with GDB. To learn how to package your kext for installation by your customers, read "Packaging a Kernel Extension for Distribution and Installation" (page 49).

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

Command-Line Tools for Analyzing Kernel Extensions

You can simplify your kext development process with the following command-line tools. More information on these tools can be found in their respective man pages.

Generate Debug Symbols and Prepare Kexts for Loading with kextutil

Use the `kextutil` utility to generate debug symbols for your kext, and to test whether your kext can be loaded. While you are debugging your kext, you should use `kextutil` to load your kext instead of `kextload`.

Commonly used `kextutil` options include:

-n / -no-load
: Does not actually load the kext into the kernel. This option is useful when you only want to generate debug symbols or determine whether a kext can be loaded.

-s / -symbols
: Generates debug symbols for the kext in the directory specified after this option.

-t / -print-diagnostics
: Outputs whether or not the kext appears to be loadable, along with a diagnosis if the kext doesn't seem to be loadable.

-e / -no-system-extensions and -r / -repository
: Typically used together, these indicate that `System/Library/Extensions` should not be used as the default kext repository when resolving dependencies for your kext, and a specified folder should be used instead.

The `kextutil` utility includes additional options for simulating various load situations. See the `kextutil` man page for more information.

Output the Status of Loaded Kexts with kextstat

Use the `kextstat` utility to output the following information for each kext loaded in the kernel:

- The load index of the kext (used to track linkage references)
- The number of references to the kext from other kexts
- The kernel-space memory address of the kext
- The size, in bytes, of the kext
- The amount of wired memory, in bytes, occupied by the kext
- The bundle identifier of the kext
- The bundle version of the kext

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

- The load indices of other kexts that the kext has a reference to

See `kextstat` for more information.

Determine Kext Dependencies with kextlibs

Use the `kextlibs` utility to determine which library kexts your kext must link against in order to resolve its symbols. You must list the bundle identifiers of these library kexts in the `OSBundleLibraries` dictionary of your kext's information property list.

Commonly used `kextlibs` options include:

-xml
: Produces XML output you can copy and paste for the `OSBundleLibraries` dictionary of your kext's information property list.

-undef-symbols
: Displays symbols that `kextlibs` could not locate. You may be able to locate these symbols by using the `kextfind` utility (see "Locate Kexts with kextfind" (page 46)).

See `kextlibs` for more information.

Locate Kexts with kextfind

Use the `kextfind` utility to search for kexts with custom queries. In addition to its query predicates, `kextfind` includes predicates for generating tab-delimited reports for further processing.

Commonly used `kextfind` query predicates include:

-dsym / -defines-symbol
: Prints only kexts that define the symbol specified after this option. This predicate is useful for locating symbols in your kext that `kextlibs` can't locate.

-lib / -library
: Returns only library kexts that other kexts can link against.

The `kextfind` utility contains many more query predicates and report predicates you can use to fine-tune your search. See `kextfind(8)` for more information.

Obtain Instance Counts with ioclasscount

Use the `ioclasscount` utility to obtain the current number of instances of any given subclass of the `OSObject` C++ class (which includes virtually all built-in kernel classes). The instance count returned for a class includes the number of instances of that class's *direct* subclasses. You can use `ioclasscount` to discover leaked instances that you should have deallocated before your kext was unloaded.

See `ioclasscount` for more information.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

View the I/O Kit Registry with IORegistryExplorer

Use the IORegistryExplorer application (located in `/Developer/Applications/Utilities`) to view the current state of the I/O Kit registry. IORegistryExplorer also includes several searching and browsing options to help you navigate the registry.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

Packaging a Kernel Extension for Distribution and Installation

Before you distribute a kext for installation, you should prepare it by creating a **package**. A packaged kext provides users with the information they expect when they install software, such as licensing restrictions and a default installation location. If you have not yet created a kext, complete "Creating a Generic Kernel Extension with Xcode" (page 17) or "Creating a Device Driver With Xcode" (page 27) before completing this tutorial.

Road Map

Here are the major steps you will follow:

1. "Set Permissions for your Kext" (page 49)
2. "Create Custom Installer Information" (page 50)
3. "Create a Package with PackageMaker" (page 51)
4. "Build the Package and Test Installation" (page 53)

This tutorial assumes that you are logged in as an administrator of your machine, which is necessary for using the `sudo` command.

Set Permissions for your Kext

Before you package your kext, you need to make sure it has the proper permissions and that it resides in a directory with root permissions when it is packaged.

1. Create a directory for a copy of your kext in the `/tmp` directory as the root user.

```
% cd /tmp
% sudo mkdir mykextdir
Password:
```

2. Create a copy of your kext as the root user and place it in the folder you created.

```
% cd /KEXT_PROJECT_PATH/build/Release
% sudo cp -R MyKext.kext /tmp/mykextdir/
```

Do not change the permissions of the original kext in your Xcode project's build folder, or else you will encounter errors when you attempt to rebuild.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

Create Custom Installer Information

You can include custom installation information in your package to improve the installation process for your users. You will create a welcome message file, a Read Me file, and a software license agreement file for your package with the TextEdit application. These supplementary resources should not be placed in the directory you created in the previous step; instead, put them in your kext's Xcode project folder.

The Welcome Message

The welcome message is the first thing your customers read when they open your kext's package. It should be a brief introduction to the software your customer is installing.

1. Create a new file in TextEdit.
2. Enter the text of your welcome message.
3. Save your welcome message as `Welcome.rtf` in your kext's project folder.
4. Close the file.

The Read Me

The Read Me describes the contents of your package, version information, and any additional information your customer needs to know before installing.

1. Create a new file in TextEdit.
2. Enter the text of your Read Me.
3. Save your welcome message as `ReadMe.rtf` in your kext's project folder.
4. Close the file.

The Software License Agreement

The software license agreement describes the terms of use for your package, legal disclaimers, and any prerelease software warnings.

1. Create a new file in TextEdit.
2. Enter the text of your software license agreement.
3. Save your software license agreement as `License.rtf` in your kexts project folder.
4. Close the file.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

Once you have created all three files, make sure to add them to your Xcode project by choosing Project > Add to Project; this ensures that they are included in your project's SCM.

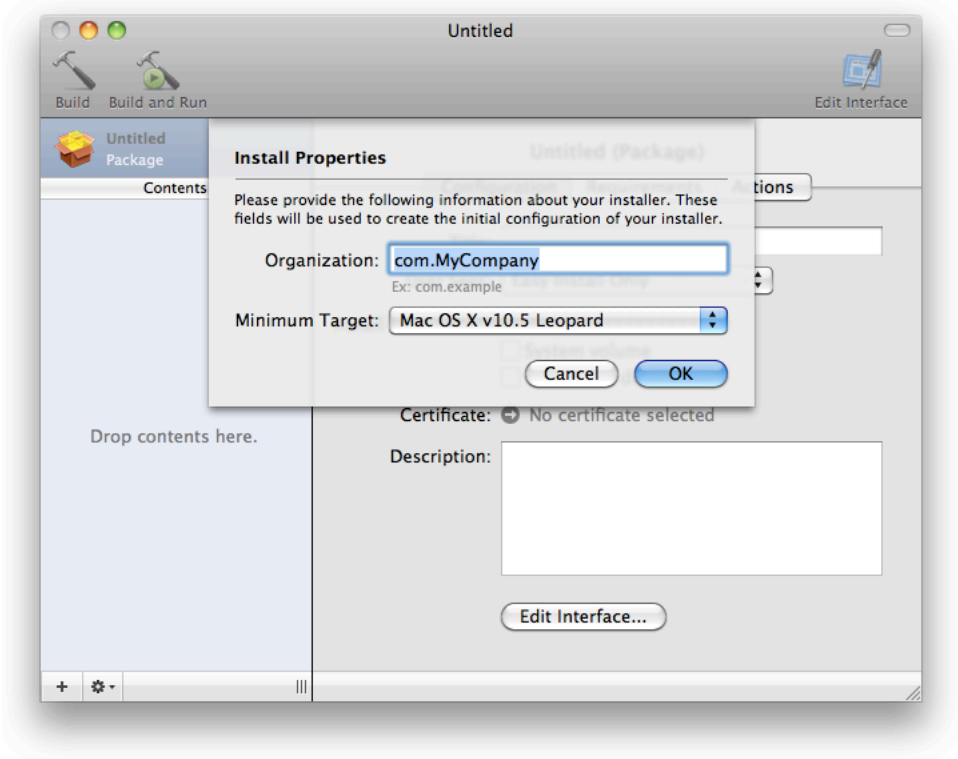
Create a Package with PackageMaker

Now you can use the PackageMaker application to build an installable package for your kext.

1. Open the PackageMaker application, located in `/Developer/Applications/Utilities`.

   The main window appears with an Install Properties sheet.

2. Enter `com.MyCompany` in the Organization field, and select Mac OS X v10.5 Leopard as the minimum target. Click OK.



3. Fill in the fields of the configuration tab of the main window as follows:

Title	`MyKext`
User Sees	Easy Install Only
Install Destination	System Volume (make sure all other install destinations are unchecked)

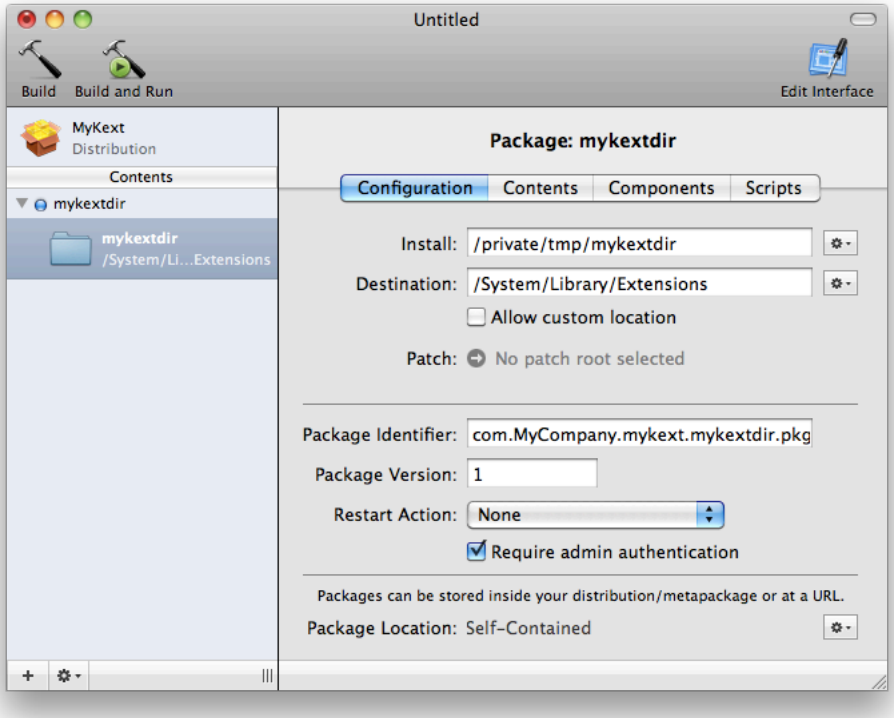
The Certificate and Description fields are not needed for this tutorial, but you need to specify a certificate for your package if you want it to be signed.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

4. Locate the copy of your kext you created in "Set Permissions for your Kext" by opening a Finder window. Choose Go > Go to Folder. Enter `/tmp` as the folder.

5. Drag the `mykextdir` folder from the Finder window and drop it into the Contents pane of the main PackageMaker window. The main view changes to show information about the `mykextdir` package.

6. Enter `/System/Library/Extensions` in the Destination field of the Configuration tab.



Now that the package has everything it needs for Installer to install your kext, you can customize the installation experience for your customers.

7. Click the Edit Interface button in the upper-right corner of the window.

   The Interface Editor window opens.

8. The first page of the Interface Editor allows you to provide a custom background image for your installation. You have not created one for this tutorial, so click Continue.

9. The second page allows you to provide custom welcome text. Choose the File radio button on the right side of the editor and provide the path for your welcome message file by clicking the gear menu next to the text box and choosing Choose.

10. The third page allows you to provide a Read Me. Repeat the same process you used for the welcome message, instead providing the path for your Read Me.

11. The fourth page allows you to provide a software license agreement. Repeat the same process you used for the welcome message, instead providing the path for your software license agreement.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

12. The fifth page allows you to provide a custom conclusion message. You have not created one for this tutorial, so close the Interface Editor window.

13. Save your progress by choosing File > Save. Specify a location of your choice and enter `MyKextPackage.pmdoc` as the filename.

Add Preinstall and Postinstall Actions (Optional)

You can further configure your kext's installation by specifying actions that run before and/or after your kext is installed. This tutorial doesn't require any such actions, so continue to the next step unless your kext has specific preinstall or postinstall requirements.

Require Restart

If your kext needs to load during early boot, or if your install actions require a restart, set the Restart Action option in the Configuration tab to Require Restart. Installer will prompt the user for a restart after executing any postinstall actions.

Add Actions

You can make sure certain actions are taken before and after your kext is installed. In the case of a kext, these actions most often involve loading or unloading other kexts.

1. Click the MyKext package in the upper left above the Contents view.

2. Click the Actions tab.

3. Click the Edit button for either Preinstall Actions or Postinstall Actions, depending on which you want to add. A sheet appears.

4. Drag the actions you want to add from the list on the left to the view on the right. Fill in any fields the actions require.

5. Order the actions in the view by dragging and dropping, such that the first action you want to perform appears at the top of the view.

Save your progress.

Build the Package and Test Installation

You are ready to build and test your package.

1. Choose Project > Build.

   Specify a location of your choice and enter `MyKext.pkg` as the filename.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

2. Double-click your package to run the Installer application.

   As you proceed through the installation process, the custom messages you included appear.

3. Check that the package was properly installed.

   Navigate to `/System/Library/Extensions`. You should see your kext.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

Info.plist Properties for Kernel Extensions

This appendix describes the properties you can use for your kext's `Info.plist` file.

Top-Level Properties

`CFBundleIdentifier`

The `CFBundleIdentifier` property uniquely identifies your kext. Two kexts with the same value for this property cannot both be loaded into the kernel. The value for this property should be in a reverse-DNS format, for example `com.MyCompany.driver.MyDriver` for an I/O Kit driver or `org.MyCompany.kext.MyKext` for a generic kext.

This property is required.

`CFBundleExecutable`

The `CFBundleExecutable` property specifies the name of your kext's executable code. Xcode automatically creates and populates this value correctly for all kext projects, so you should not need to change it.

This property is required.

`CFBundleVersion`

The `CFBundleVersion` property indicates your kext's version. Kext version numbers must adhere to a strict format:

- The version number is divided into three parts by periods, for example `3.1.2`.

  The first number represents the most recent major release, the second number represents the most recent significant revision, and the third number represents the most recent minor bug fix.

  The first number is limited to four digits; the second and third numbers are limited to two digits each.

  If the value of the third number is `0`, you can omit it and the second period.

- While developing a new version of your kext, include a suffix after the number that is being updated, for example `3.1.3a1`.

  The letter in the suffix represents the stage of development the new version is in (development, alpha, beta, or final candidate, represented by `d`, `a`, `b`, and `fc`), and the number in the suffix is the build version. The build version cannot be `0` or exceed `255`.

  When you release the new version of your kext, make sure to remove the suffix.

This property is required.

`OSBundleLibraries`

The `OSBundleLibraries` property is a dictionary that lists the library kexts that your kext links against.

Each element in the 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

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

dependency. 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, the kext and its dependencies are deemed compatible.

You determine the kexts to add with the `kextlibs` command-line tool (see "Determine Kext Dependencies with kextlibs" (page 46)).

This property is required.

This property can be architecture-specific (see "Architecture-Specific Properties" (page 58)).

`OSBundleRequired`

The `OSBundleRequired` property informs the system that your kext must be available for loading during early boot. Kexts that don't set this property can't load during early boot. You can specify one of the following values for this property:

`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.

This property can be architecture-specific (see "Architecture-Specific Properties" (page 58)).

`OSBundleCompatibleVersion`

The `OSBundleCompatibleVersion` property is used to enable linking against a kext as a library. It indicates the oldest version of your library kext that other kexts can link against and still use the current version successfully.

You should increment the value of this property when you remove a symbol from the library, or when an exported symbol's semantics change significantly enough to impact binary compatibility.

The format of this value is the same as that of `CFBundleVersion`.

This property can be architecture-specific (see "Architecture-Specific Properties" (page 58)).

`OSBundleAllowUserLoad`

The `OSBundleAllowUserLoad` property allows non-root users to load your kext. Using this property is not recommended.

I/O Kit drivers should never include this property, because they are loaded by the kernel when they are needed.

Specify a boolean value of true to enable this option.

This property can be architecture-specific (see "Architecture-Specific Properties" (page 58)).

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

`OSBundleEnableKextLogging`

The `OSBundleEnableKextLogging` property indicates that logging information specific to your kext should be logged in the kernel log (available at `/var/log/kernel.log`). The `kextutil` tool automatically enables this option to assist with debugging. Specify a boolean value of `true` to enable this option. See `kext_logging` for more information.

This property can be architecture-specific (see "Architecture-Specific Properties" (page 58)).

`IOKitPersonalities`

The `IOKitPersonalities` property is used by I/O Kit drivers. It is a nested dictionary of information describing hardware that the driver can operate.

See "IOKitPersonalities Properties" (page 57) for a list of properties to include in the `IOKitPersonalities` dictionary.

See "Driver Personalities and Matching Languages" in *I/O Kit Fundamentals* for more information on personalities.

This property is required for I/O Kit drivers.

This property can be architecture-specific (see "Architecture-Specific Properties" (page 58)).

IOKitPersonalities Properties

`IOClass`

The `IOClass` property names the C++ class to instantiate from your driver when it matches on a nub.

`IOKitDebug`

The `IOKitDebug` property indicates that I/O Kit-specific events such as attaching, matching, and probing should be logged in the kernel log (available at `/var/log/kernel.log`). The value of this property defines which events are logged. To log all relevant information, specify `65535` as the value. See `IOKitDebug.h` (available in `/System/Library/Frameworks/Kernel.framework/Headers/IOKit`) for itemized logging values.

`IOProviderClass`

The `IOProviderClass` property names the C++ class of the I/O Kit device object that your driver matches on. This is typically the nub that controls the port that your device connects to. For example, if your driver connects to a PCI bus, you should specify `IOPCIDevice` as your driver's provider class.

`IOMatchCategory`

The `IOMatchCategory` property allows multiple drivers with unique values for the property to match on the same provider class. Typically, only one driver can match on a given provider class. Include this property if you are matching on `IOResources` or on a port with multiple devices attached to it. The value for this property should be the same as the value for `CFBundleIdentifier`, with periods replaced with underscores (for example `com_MyCompany_driver_MyDriver`).

`IOResourceMatch`

The `IOResourceMatch` property allows you to declare a dependency between your driver and a specific resource, such as the BSD kernel or a particular resource on a device, like an audio-video jack. If you provide this property, your driver will not load into the kernel until the specified resource is available.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

Architecture-Specific Properties

Top-level kext Info.plist properties that begin with `OS` or `IO` have architecture-specific versions you can use to differentiate your kext's behavior on different architectures. To specify an architecture-specific property, add an underscore followed by the architecture name to a property name, for example `OSBundleCompatibleVersion_x86_64` or `OSBundleCompatibleVersion_i386`. Make sure to keep the base property in your Info.plist file for backwards compatibility.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

Document Revision History

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

Date	Notes
2010-03-19	Restructured and updated for Mac OS X v10.6.
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.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.

2010-03-19 | © 2003, 2010 Apple Inc. All Rights Reserved.