

INSIDE MAC OS X

Kernel Extensions Tutorials

Hello Kernel, Hello I/O Kit, Hello Debugger, How to Package a KEXT



Preliminary

October 2000

Apple Computer, Inc.
© 2000 Apple Computer, 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 Computer, 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 Computer, 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 book. Apple retains all intellectual property rights associated with the technology described in this book. This book is intended to assist application developers to develop applications only for Apple-labeled or Apple-licensed computers.

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

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

Apple, the Apple logo, AppleTalk, Mac, Macintosh, and QuickTime are trademarks of Apple Computer, Inc., registered in the United States and other countries.

Carbon, Cocoa, and Quartz are trademarks of Apple Computer, Inc.

NeXT and OpenStep are trademarks of NeXT Software, Inc., registered in the United States and other countries.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company, Ltd.

Simultaneously published in the United States and Canada

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, 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 MANUAL, 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

Chapter 1 **Hello Kernel: Creating a Kernel Extension With Project Builder** 7

Anatomy of a KEXT	7
Roadmap	8
Create a new Project using Project Builder	8
Create a Kernel Extension Project	9
Implement the Needed Methods	9
Edit the KEXT's Property List	11
Build the Kernel Extension	12
Build the Project	12
Fix Any Errors	13
If There Were No Errors	14
Build the Project Again	15
Test the Kernel Extension	15
Start the Terminal Application	16
Load the Kernel Extension Module	17
Using Console Mode	19
Where to Go Next	20

Chapter 2 **Hello IOKit: Creating a Device Driver With Project Builder** 21

Anatomy of a Device Driver	21
Roadmap	22
Create an I/O Kit Project using Project Builder	23
Create an I/O Kit Extension Project	24
Edit the Driver's Property List	24
Add the IOKitPersonalities Dictionary Entries	25
Implement the Header File	29
Implement the Driver's Entry Points	31
Build the Project	34
Fix Any Errors	35
Test the Device Driver	36

C O N T E N T S

Start the Terminal Application	36
Load the Driver	38
Using Console Mode	39
Where to Go Next	40

Chapter 3 Hello Debugger: Debugging a Device Driver With GDB 43

Preparation	43
Roadmap	44
On Target Machine, Enable Kernel Debugging	46
On Development Machine, Set Up a Permanent Network Connection	47
On Target Machine, Create a Symbol File	50
On Development Machine, Import the Symbol File	52
On Development Machine, Start GDB	53
On Target Machine, Break into Kernel Debugging Mode	54
On Development Machine, Attach to the Target Machine	55
On Development Machine, Set Breakpoints in GDB	55
On Target Machine, Start Running the Device Driver	56
On Development Machine, Control the Driver With GDB	57
Stop the Debugger	58
On Target Machine, Unload the Driver	59
Where to Go Next	59

Chapter 4 Packaging Your KEXT for Distribution and Installation 61

Anatomy of a Package	61
Preparation	62
Roadmap	63
Locate Your KEXT	64
Create a Distribution Directory	65
Gather Any Package Resources	66
Create a ReadMe File	67
Create a Software Licence File	68
Create a Package with PackageMaker	68
Examine the Package and Files	71
Test Installing the Package	73

C O N T E N T S

Test Uninstalling the Package	75
Archive Your Package for Distribution	76
Where to Go Next	77

C O N T E N T S

Hello Kernel: Creating a Kernel Extension With Project Builder

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.

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 **module**—the module or KMOD contains the kernel extension's binary code. This is what's actually loaded into the kernel and run. Generally, a KEXT has one module, but it can have none (locally). However, if it has none, its property list must reference a module in another KEXT and change its default settings. A KEXT must refer to at least one module.
- Optional **Resources**—Resources are useful if your KEXT needs to display a dialog or 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 Project Builder”](#) (page 8)
2. [“Build the Kernel Extension”](#) (page 12)
3. [“Test the Kernel Extension”](#) (page 15)

You'll use the Project Builder 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 Project Builder before, you may also wish to complete the tutorial, “HelloWorld: Creating a Project With Project Builder”. This tutorial can be found in the folder `/Developer/Documentation/DeveloperTools/ProjectBuilder` after you have installed the Developer tools for Mac OS X.

Create a new Project using Project Builder

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). Project Builder 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 Project Builder.

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

1. [“Create a Kernel Extension Project”](#) (page 9)

Hello Kernel: Creating a Kernel Extension With Project Builder

2. “Implement the Needed Methods” (page 9)
3. “Edit the KEXT’s Property List” (page 11)

The examples below assume that you will be logging in as the administrator of your machine. The account name in the examples is `admin`. If you use a different login account, be sure to substitute accordingly. This tutorial requires that you know the super user (`root`) password; by default, that password is the same as the administrator password.

Create a Kernel Extension Project

Kernel extensions are created and edited in Project Builder, Apple’s Integrated Development Environment (IDE).

From a Desktop Finder window, locate and launch the Project Builder application, found at `/Developer/Applications/Project Builder`. If this is the first time you’ve run Project Builder, 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 and choose 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, Project Builder creates the new project and displays its project window. The new project contains several files already, including a default source file, `HelloKernel.c`.

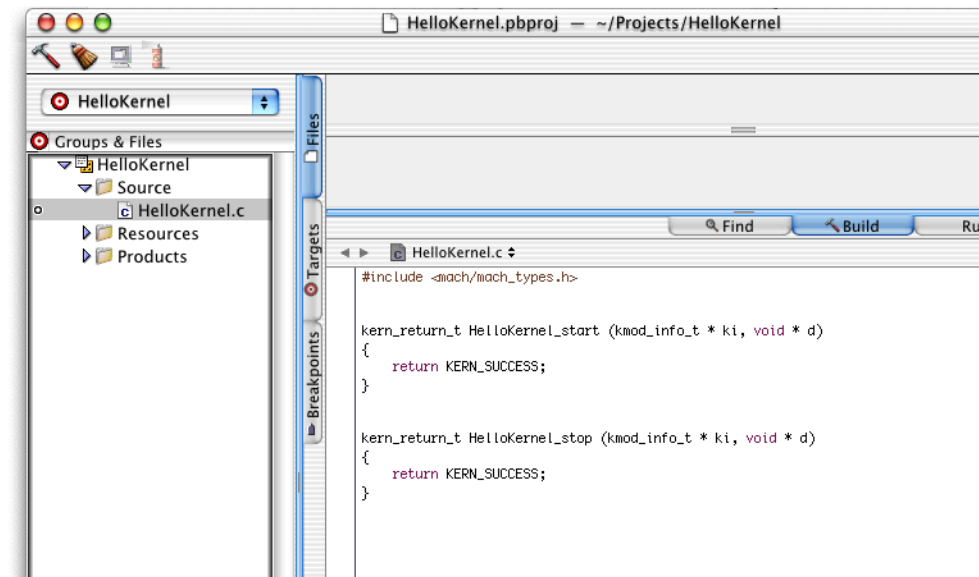
Implement the Needed Methods

Click on the Files tab, if necessary, to display the Groups & Files pane. Click on the disclosure triangle next to Source. Click on `HelloKernel.c` to display the source code. Figure 1-1 shows where you will find the `HelloKernel.c` file in the project window.

Hello Kernel: Creating a Kernel Extension With Project Builder

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 1-1 Viewing Source Code in Project Builder.



Change the contents of `HelloKernel.c` to match the code in [Listing 1-1](#). The lines you must add are in bold.

Listing 1-1 HelloKernel.c

```

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

kern_return_t HelloKernel_start (kmod_info_t * ki, void * d)
{

```

Hello Kernel: Creating a Kernel Extension With Project Builder

```

    printf("Module has loaded!\n");
    return KERN_SUCCESS;
}

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

```

Edit the KEXT'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 (Extended Markup Language) format. However, you will be viewing and editing the plist information from within Project Builder.

1. Click the Targets tab and select the HelloKernel target. Then click the Build Settings tab. Scroll down to the section entitled, Expert Build Settings. These settings define properties which will be stored in the plist.
2. The `MODULE_START` and `MODULE_STOP` properties contain the names of your KEXT's initialization and termination routines. Be sure that these properties have the values used in your `HelloKernel.c` file. That is, for this tutorial, be sure these properties 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 properties match the names you used! Otherwise, your KEXT will not run.
3. The value of the `MODULE_VERSION` properties is based on the 'vers' resource as used in Mac OS 8 and 9. You may change the `MODULE_VERSION` value if you wish, or leave it alone.
4. Change the name of the module from the default chosen by Project Builder (the value of the `MODULE_NAME` property).

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

Hello Kernel: Creating a Kernel Extension With Project Builder

By default, Project Builder names a new kernel module `com.MySoftwareCompany.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`.

Click on the line for `MODULE_NAME` and double-click on `MySoftwareCompany` in the Value field. Change this string to `MyTutorial`.

5. Click the Bundle Settings tab and change the Identifier string to match the value of `MODULE_NAME` from the previous step.

For this tutorial, change the Identifier to `com.MyTutorial.kext.HelloKernel`

Important

You must make sure that the `MODULE_NAME` value in the Build Settings panel matches the Identifier string in the Bundle Settings panel or your KEXT will not run.

Build the Kernel Extension

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

1. [“Build the Project”](#) (page 12)
2. [“Fix Any Errors”](#) (page 13)
3. [“If There Were No Errors”](#) (page 14)
4. [“Build the Project Again”](#) (page 15)

Build the Project

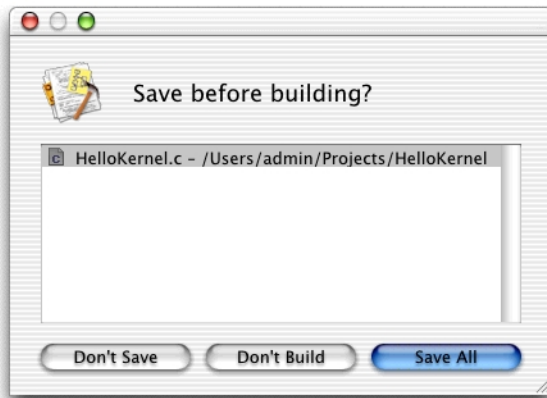
Click the Build button in the upper left corner of the Project Window or select Build from the Build menu. The Build button looks like a hammer and is illustrated in [Figure 1-2](#). (Be careful not to simply click the Build tab on the right side of the Project window.)

Figure 1-2 The Build Button Looks Like a Hammer



If Project Builder asks you whether to save some modified files, select all the files and click “Save All”. [Figure 1-3](#) shows the Save dialog.

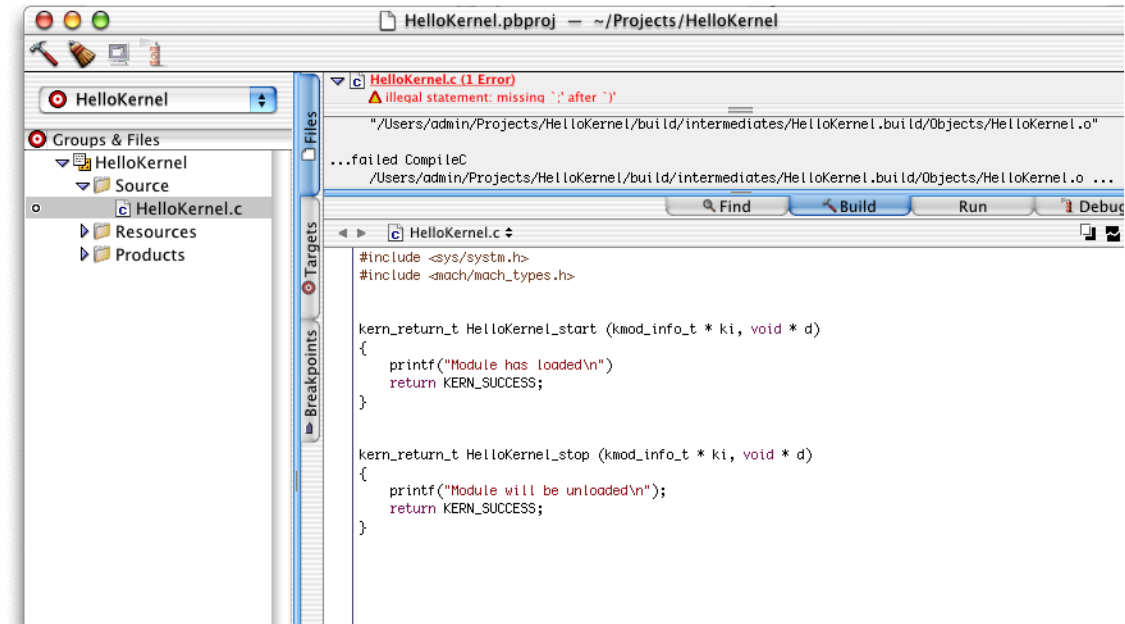
Figure 1-3 Save Before Building



Project Builder 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, Project Builder will stop and show you the errors. Click the build error that appears in the top view of the panel. The source code with the error will appear in the editing view below it, as shown in [Figure 1-4](#).

Figure 1-4 Project Builder Shows Code Errors

You can edit the source code inside the editing view. Correct the error 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 1-2](#).

Listing 1-2 HelloKernel.c

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

Hello Kernel: Creating a Kernel Extension With Project Builder

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

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

Build the Project Again

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

Project Builder starts building your project again. If there is an error, fix the error as described above in [“Fix Any Errors”](#) (page 13). 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. First you’ll load your KEXT’s module with the `kextload` command. Then you’ll use the `kmodstat` command to see that it’s loaded. Finally, you’ll unload your module from the kernel with 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 that you use `kextload` and `kextunload` only when testing a kernel extension. When a KEXT is fully installed under Mac OS X, the Kernel Extension Manager takes care of loading and running (and unloading) modules.

Here’s how you’ll test your KEXT:

Hello Kernel: Creating a Kernel Extension With Project Builder

1. “Start the Terminal Application” (page 16)
2. “Load the Kernel Extension Module” (page 17)

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. Choose New from the Shell menu to start a new shell window.
3. You’ll view the system log file in this window. At the prompt, enter the command:

```
tail -f /var/log/system.log
```
4. Open a second window in the Terminal application. Choose New from the Shell menu. Position this window so that you can view both windows easily. You will load your module from the second window.
5. Move to the directory that contains your KEXT. Project Builder stores your KEXT in the `build` directory of your project location (unless you’ve set a different location for build products using Project Builder’s Preferences dialog).

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

```
cd Projects/HelloKernel/build/
```

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

```
ls  
HelloKernel.kext intermediates
```

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 kernel module (stored in the KEXT’s property list) should be unique, using the recommended “reverse-DNS” naming convention.

Hello Kernel: Creating a Kernel Extension With Project Builder

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 kernel module (Contents/MacOS/HelloKernel).

6. 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/PkgInfo
HelloKernel.kext/Contents/Resources
HelloKernel.kext/Contents/Resources/English.lproj
HelloKernel.kext/Contents/Resources/English.lproj/InfoPlist.strings
HelloKernel.kext/Contents/version.plist
```

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

Load the Kernel Extension Module

1. You must assume special privileges as the super user (also known as `root`); only this account can load kernel extensions.

At the prompt, enter the `su` command to become the super user. When prompted for a password, enter the administrator password for your computer. Note that nothing is displayed as you type the password.

For example:

```
% su
Password:
root#
```

2. From the KEXT's Contents directory, use the `kextload` command; this loads your module and runs its initialization (start) function.

For example:

Hello Kernel: Creating a Kernel Extension With Project Builder

```
# kextload -v HelloKernel.kext
Examining: HelloKernel.kext
Loading module: com.MyTutorial.kext.HelloKernel.
Loaded module: com.MyTutorial.kext.HelloKernel.
Done.
```

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

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

```
kmod_create: com.MyTutorial.kext.HelloKernel (id 13), 3 pages ...
mach_kernel: Module has loaded
```

4. Use the `kmodstat` command to check the status of the KEXT. This command displays the status of all dynamically-loaded kernel modules. Enter the command:

```
kmodstat
```

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

```
Id  RefsAddressSize WiredName (Version) <Linked Against>
 1   0   0x4b940000x17000 0x16000ATIR128 (0.1)
 2   2   0x4bbb0000x10000 0xf000com.apple.IOAudioFamily (0.1a)
 3   1   0x4bcb0000x1e000 0x1d000com.apple.IOFireWireFamily (0.1a)
...
10   0   0x4d180000x7000 0x6000SIP-NKE (0.1a)
11   0   0x50470000x3000 0x2000com.MyTutorial.kext.HelloKernel...
```

5. Unload the module. Use the `kextunload` command. This command unloads your module and runs its termination (stop) function.

For example:

```
# kextunload HelloKernel.kext
IOCatalogueTerminate(Module com.MyTutorial.kext.HelloKernel) [0]
done.
```

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

```
mach_kernel: Module will be unloaded
mach_kernel: kmod_destroy: com.MyTutorial.kext.HelloKernel (id 11)...
```

Using Console Mode

As an alternative to the Terminal application, you can load and test your module from console mode. In console mode, all system messages (such as the kernel extension's message "Hello Kernel!") 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 ProjectBuilder 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 Desktop menu.

2. From the login screen, log into 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 as `root`. Only the root account (also known as the super user) may load kernel extensions. Use the password you have set for the administrator account.

For example:

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

```
login: root
```

```
Password for root:
```

```
root#
```

4. Move to the directory that contains your module. Use the `cd` command.

For example:

```
cd /Users/me/HelloKernel/build/HelloKernel.kext/Contents
```

5. Follow the instructions given in "Load the Kernel Extension Module" (page 17). Remember that the console messages will come directly to your screen; you do not need to view the system log file.

Hello Kernel: Creating a Kernel Extension With Project Builder

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 IOKit: Creating a Device Driver With Project Builder](#)” (page 21), 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`, `kmodstat`, and `kextunload`. For example, from a Terminal window, enter the command

```
man kmodstat
```

This tutorial is the first in a series. Additional tutorials describe how to build a device driver ([Hello IOKit: Creating a Device Driver With Project Builder](#)), how to debug a kernel extension ([Hello Debugger: Debugging a Device Driver With GDB](#)), how to package a KEXT ([Packaging Your KEXT for Distribution and Installation](#)), and how to debug a kernel panic (“[Don’t Panic: Debugging a Kernel Panic](#)”). These tutorials can be found in `/Developer/Documentation/Kernel/Tutorials`.

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/techpubs/mac/Toolbox/Toolbox-454.html#HEADING454-0>>.

Hello IOKit: Creating a Device Driver With Project Builder

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

Be sure you have completed the previous tutorial, [“Hello Kernel: Creating a Kernel Extension With Project Builder”](#) (page 7) before beginning this one. That tutorial demonstrates how to create a simple kernel extension project.

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 kernel **module**—the module or KMOD contains the driver's binary code. This is what's actually loaded into the kernel and run. Generally, a driver has one module, but it can have none. However, if it has none, its property list must reference a module in another KEXT and change that module's default settings. A kernel extension must refer to at least one module.
- Optional **Resources**—Resources are useful if your driver needs to display a dialog or an icon.

Hello IOKit: Creating a Device Driver With Project Builder

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 [“Add the IOKitPersonalities Dictionary Entries”](#) (page 25) 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 Bundle Settings.
- You must create a header file for a driver. See [“Implement the Header File”](#) (page 29).
- The I/O Kit requires several specific entry points, described in the section, [“Implement the Driver’s Entry Points”](#) (page 31). 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 29) and [“Implement the Driver’s Entry Points”](#) (page 31) for examples of the correct placement of these macros in your code.

- Device drivers are (frequently) written in C++.

Roadmap

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

1. [“Create an I/O Kit Project using Project Builder”](#) (page 23)
2. [“Build the Project”](#) (page 34)
3. [“Test the Device Driver”](#) (page 36)

Hello IOKit: Creating a Device Driver With Project Builder

You'll use the Project Builder 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 Project Builder much, you may also wish to complete the tutorial, "HelloWorld: Creating a Project With Project Builder". This tutorial can be found in the folder `/Developer/Documentation/DeveloperTools/ProjectBuilder` after you have installed the Developer tools for Mac OS X.

The examples below assume that you will be logging in as the administrator of your machine. The account name in the examples is `admin`. If you use a different login account, be sure to substitute accordingly. This tutorial requires that you know the super user (`root`) password; by default, that password is the same as the administrator password.

Create an I/O Kit Project using Project Builder

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). Project Builder 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 Project Builder.

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

1. "Create an I/O Kit Extension Project" (page 24)
2. "Edit the Driver's Property List" (page 24)
3. "Add the IOKitPersonalities Dictionary Entries" (page 25)
4. "Implement the Header File" (page 29)

Hello IOKit: Creating a Device Driver With Project Builder

The examples below assume that you will be logging in as the administrator of your machine. The account name in the examples is `admin`. If you use a different login account, be sure to substitute accordingly.

Create an I/O Kit Extension Project

Device drivers are created and edited in Project Builder, Apple's Integrated Development Environment (IDE).

From a Desktop Finder window, locate and launch the Project Builder application, found at `/Developer/Applications/ProjectBuilder`. If this is the first time Project Builder 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 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, Project Builder creates the new project and displays its project window. Notice that the new project contains several files already, including two source files — `HelloIOKit.h` and `HelloIOKit.cpp`.

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 Project Builder, so you don't need to worry about the underlying format.

1. Click the Targets tab and select the `HelloIOKit` target. Then click the Build Settings tab. Scroll down to the section, Expert Build Settings. These settings define macros which will be stored in the plist.

Hello IOKit: Creating a Device Driver With Project Builder

2. Note that the `MODULE_START` and `MODULE_STOP` properties 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 Bundle Settings.
3. The value of the `MODULE_VERSION` property is based on the 'vers' resource as used in Mac OS 8 and 9. You may change the `MODULE_VERSION` value if you wish, or leave it alone.
4. Change the name of the module, from the default chosen by Project Builder (the value of the `MODULE_NAME` property).

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, Project Builder names a new driver module `com.MySoftwareCompany.iokit.<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`.

Click on the line for `MODULE_NAME` and double-click on `MySoftwareCompany` in the Value field. Change this string to `MyTutorial`.

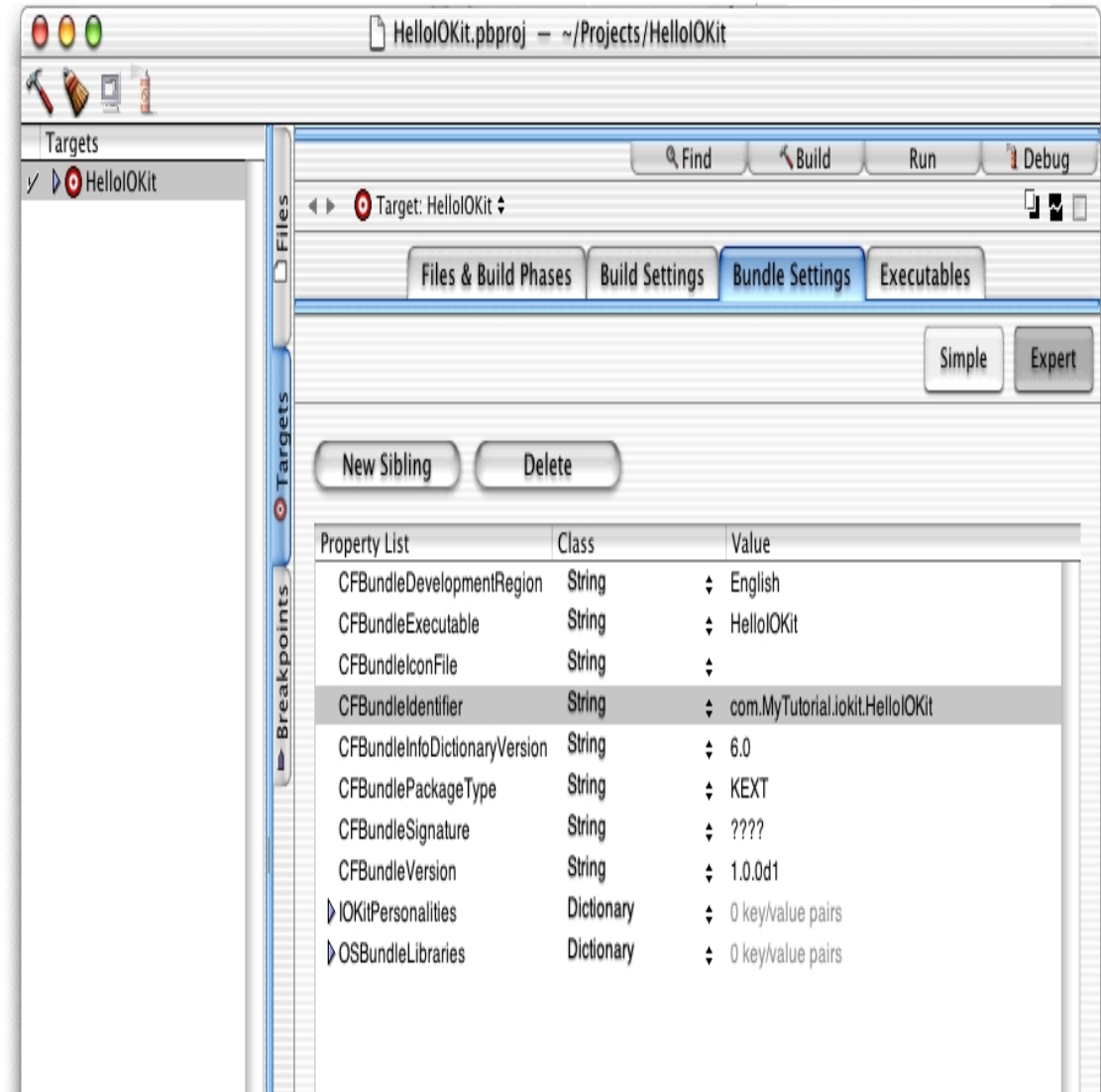
5. Click the Bundle Settings tab and, in the next panel, change the Identifier string to match the value of `MODULE_NAME` from the previous step. For this tutorial, change the Identifier to `com.MyTutorial.iokit.HelloIOKit`.

Important

You must make sure that the `MODULE_NAME` value in the Build Settings panel matches the Identifier string in the Bundle Settings panel or your driver will not run.

Add the IOKitPersonalities Dictionary Entries

In the Bundle Settings panel, click the Expert button in the upper-righthand corner. In the panel, you'll see several elements of the property list in more detail, as shown in [Figure 2-1](#).

Figure 2-1 Expert Build Settings

Hello IOKit: Creating a Device Driver With Project Builder

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.

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. 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.

Click on the line for `HelloIOKit` and 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. 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.iokit>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_iokit>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.

Hello IOKit: Creating a Device Driver With Project Builder

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 an element to help in debugging driver matching.

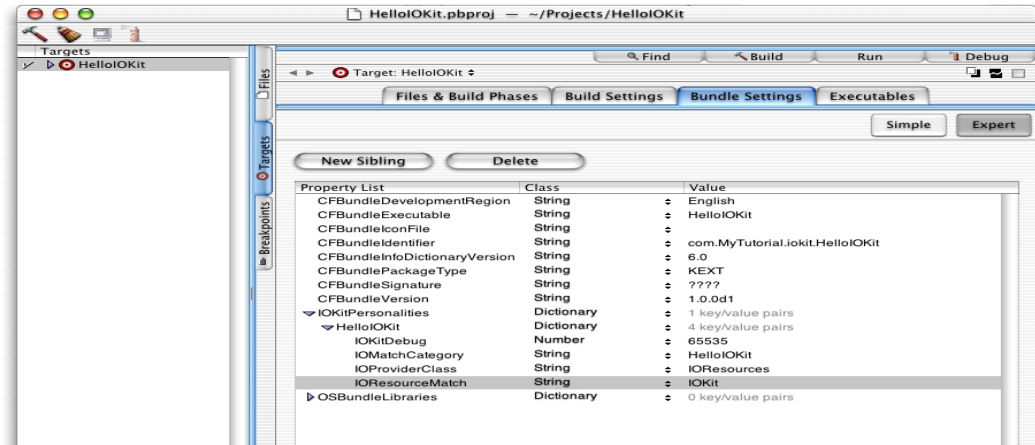
Clicking the button that says New Sibling each time, add the following property list elements. Be sure to change the Class of the first, `IOKitDebug`, from a string to a number using the popup menu. Be sure to enter the names and values exactly as shown below.

```
IOKitDebugNumber65535
IOMatchCategoryStringHelloIOKit
IOProviderClassStringIOResources
IOResourceMatchStringIOKit
```

Important

If you insert these elements in a different order, Project Builder 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!

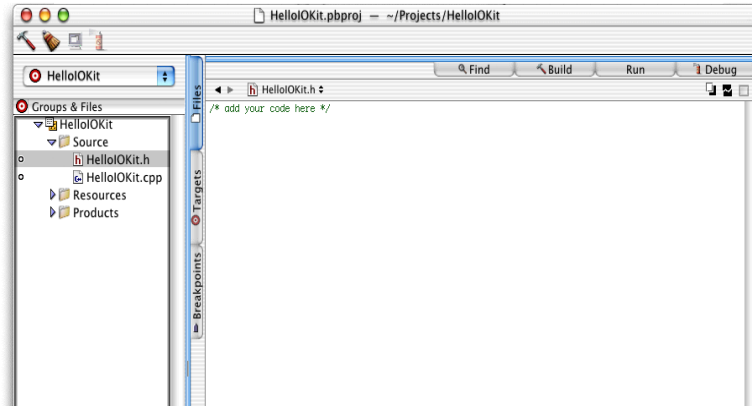
When you have finished adding property list elements, the screen should look like the example shown in [Figure 2-2](#).

Figure 2-2 Expert Build Settings after Additions

Before leaving this screen, copy the string `com_MyTutorial_iokit_HelloIOKit`; you'll need this string in the next step. To select and copy, triple-click on `com_MyTutorial_iokit_HelloIOKit` in the value field for `IOClass`, then choose Copy from the Edit menu.

Implement the Header File

Click on the Files tab, then on the disclosure triangle next to Source. 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 2-3 shows where you will find the `HelloKernel.h` file in the project window.

Figure 2-3 Viewing Source Code in Project Builder

Paste the string you copied from the Build Settings panel into `HelloIOKit.h`. Then edit the contents of `HelloIOKit.h` to match the code in [Listing 2-1](#).

Pay special attention to the lines that contain the string `com_MyTutorial_iokit_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 property list (Expert Bundle Settings panel). Other than changing dots to underbars, this name should also be identical to the Identifier defined in the Bundle Settings panel and to the `MODULE_NAME` in the Build 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 2-1 HelloIOKit.h

```
#include <IOKit/IOService.h>
class com_MyTutorial_iokit_HelloIOKit : public IOService
{
    OSDeclareDefaultStructors(com_MyTutorial_iokit_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_iokit_HelloIOKit`) again; you will need it in the next step. Then click on `HelloIOKit.cpp` in the Groups & Files pane of the Project Builder 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 those actions 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 subsequently be probed.

Hello IOKit: Creating a Device Driver With Project Builder

- 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.

Edit the contents of `HelloIOKit.cpp` to match the code in [Listing 2-2](#).

Important

Be very sure that the class name is exactly the same string you used in `HelloIOKit.h` and entered for the value of `IOClass` in the property list (Expert Bundle Settings panel). Other than changing dots to underbars, this name should also be identical to the Identifier defined in the Bundle Settings panel and to the `MODULE_NAME` in the Build 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 macros 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 macros 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.

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

Listing 2-2 HelloIOKit.cpp

```
#include <IOKit/IOLib.h>
#include "HelloIOKit.h"
extern "C" {
#include <pexpert/pexpert.h>
}

// Define my superclass
#define super IOService
```


C H A P T E R 2

Hello IOKit: Creating a Device Driver With Project Builder

```
// 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_iokit>HelloIOKit, IOService)

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

void com_MyTutorial_iokit>HelloIOKit::free(void)
{
    IOLog("Freeing\n");
    super::free();
}

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

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

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

Hello IOKit: Creating a Device Driver With Project Builder

The `IOLog` method is similar to `printf`, but runs faster and flushes its output more frequently. When you know more about drivers and C++, you can use `IOLog` to provide additional information. For example, the `IOLog` call in your driver's `init` method above could be enhanced as:

```
IOLog("%s(0x%08lx)->init(%s(0x%08lx)) = %s\n",
      instClassName(this), (UInt32) this,
      instClassName(dict), (UInt32) dict,
      (res) ? "true" : "false");
```

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.

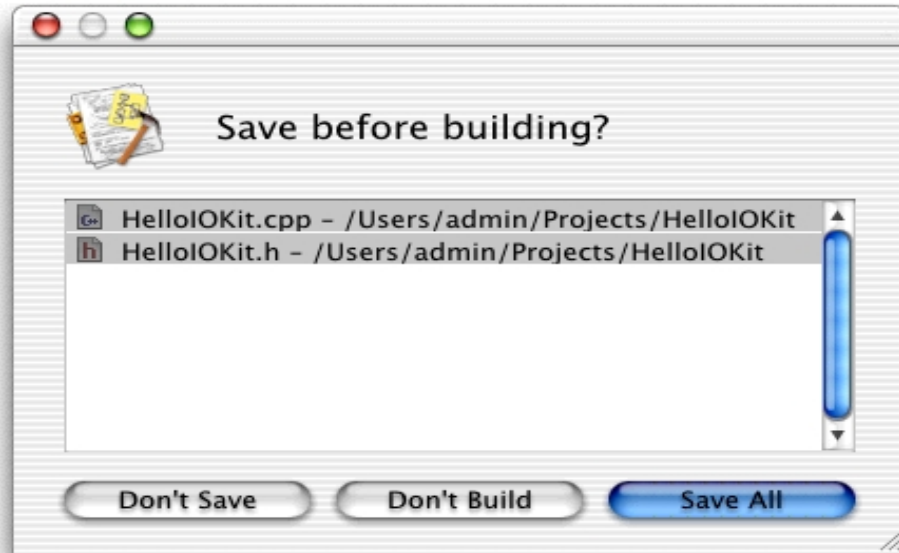
Build the Project

Click the Build button in the upper left corner of the Project Window or select Build from the Build menu. The Build button looks like a hammer and is illustrated in [Figure 2-4](#). (Be careful not to simply click the Build tab on the right side of the Project window.)

Figure 2-4 The Build Button Looks Like a Hammer



If Project Builder asks you whether to save some modified files, select all the files and click "Save All". [Figure 2-5](#) shows the Save dialog.

Figure 2-5 Save Before Building

Project Builder 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, Project Builder will stop and show you the errors. Click the build error that appears in the top view of the panel. The source code with the error will appear in the editing view below it. You can edit the source code inside the editing view. Correct any errors and build the project again.

Test the Device Driver

This section shows how to test the driver. First you'll load your driver, using the `kextload` command. Then you'll use the `kmodstat` command to see that the driver has been loaded. 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 that you use `kextload` and `kextunload` only when testing a driver. When a KEXT is fully installed under Mac OS X, the Kernel Extension Manager takes care of loading and running (and unloading) drivers.

Here's how you'll test your driver:

1. [“Start the Terminal Application”](#) (page 36)
2. [“Load the Driver”](#) (page 38)

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. Choose New from the Shell menu to start a new shell window.
3. You'll view the system log file in this window. At the prompt, enter the command:

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

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

Hello IOKit: Creating a Device Driver With Project Builder

5. Move to the directory that contains your driver. Project Builder stores your driver in the `build` directory of your project location (unless you've set a different location for build products using Project Builder's Preferences dialog).

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

```
cd Projects/HelloIOKit/build
```

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

```
ls
HelloIOKit.kext intermediates
```

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 kernel module and the module'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 kernel module (`Contents/MacOS/HelloKernel`).

6. 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/HelloKernel
HelloIOKit.kext/Contents/PkgInfo
HelloIOKit.kext/Contents/Resources
HelloIOKit.kext/Contents/Resources/English.lproj
HelloIOKit.kext/Contents/Resources/English.lproj/InfoPlist.strings
HelloIOKit.kext/Contents/version.plist
```

Hello IOKit: Creating a Device Driver With Project Builder

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

Load the Driver

1. You must assume special privileges as the super user (also known as `root`); only this account can load kernel extensions.

At the prompt, enter the `su` command to become the super user. When prompted for a password, enter the administrator password for your computer. Note that nothing is displayed as you type the password.

For example:

```
% su
Password:
root#
```

2. Use the `kextload` command; this loads your driver, runs its initialization (start) function, and registers it with the kernel.

For example:

```
# kextload -v HelloIOKit.kext
Examining: HelloIOKit.kext
Loading module: com.MyTutorial.iokit.HelloIOKit.
Loaded module: com.MyTutorial.iokit.HelloIOKit.
Done.
```

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

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

```
kmod_create: com.MyTutorial.iokit.HelloIOKit (id 3), 4 pages loaded...
Matching service count = 1
Initializing
com.MyTutorial_iokit_HelloIOKit::probe(IOResources)
Probing
com.MyTutorial_iokit_HelloIOKit::start(IOResources) <1>
Starting
```

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

```
# kmodstat
```

Hello IOKit: Creating a Device Driver With Project Builder

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

```
Id  Refs      Address SizeWiredName (Version) <Linked Against>
  1   0    0x4b94000 0x170000x16000ATIR128 (0.1)
  2   2    0x4bbb000 0x100000xf000com.apple.IOAudioFamily (0.1a)
10   0    0x4d18000 0x70000x6000SIP-NKE (0.1a)
11   0    0x50c0000 0x40000x3000com.MyTutorial.iokit.HelloIOKit ...
```

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

For example:

```
# kextunload HelloIOKit.kext
IOCatalogueTerminate(Module com.MyTutorial.iokit.HelloIOKit) [0]
done.
```

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

```
Stopping
Freeing
kmod_destroy: com.MyTutorial.iokit.HelloIOKit (id 11),...
```

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 the kernel extension's message "Hello Kernel!") 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 ProjectBuilder 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 Desktop menu.
2. From the login screen, log into console mode.

Hello IOKit: Creating a Device Driver With Project Builder

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 as `root`. Only the root account (also known as the super user) may load kernel extensions. Use the password you have set for the administrator account.

For example:

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

login: root
Password for root:
root#
```

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

For example:

```
cd /Users/me/HelloKernel/build
```

5. Follow the instructions given in “[Load the Driver](#)” (page 38). 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 a device driver. In the next tutorial in this series, “[Hello Debugger: Debugging a Device Driver With GDB](#)” (page 43), 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`, `kmodstat`, and `kextunload`. For example, from a Terminal window, enter the command

```
man kmodstat
```


Hello IOKit: Creating a Device Driver With Project Builder

This tutorial is the second in a series. Additional tutorials describe how to debug a kernel extension ([Hello Debugger: Debugging a Device Driver With GDB](#)), how to package a KEXT ([Packaging Your KEXT for Distribution and Installation](#)), and how to debug a kernel panic (“Don’t Panic: Debugging a Kernel Panic”). These tutorials can be found in `/Developer/Documentation/Kernel/Tutorials`.

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/techpubs/mac/Toolbox/Toolbox-454.html#HEADING454-0>>.

Additional useful reference material can be found under `/Developer/Documentation/CoreFoundation`. Look here for documentation about Bundles, Property Lists, Collections (such as Dictionaries) and more.

C H A P T E R 2

Hello IOKit: Creating a Device Driver With Project Builder

Hello Debugger: Debugging a Device Driver With GDB

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 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 previous tutorials.

Two previous tutorials, [Hello Kernel: Creating a Kernel Extension With Project Builder](#) and [Hello IOKit: Creating a Device Driver With Project Builder](#) describe how to create a kernel extension's project, correcting 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 the previous tutorials 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.

Hello Debugger: Debugging a Device Driver With GDB

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 TCP/IP.
- Both machines must be on the same subnet.
- You must have login access to both machines; this tutorial assumes you are logging in as the Administrative account (user `admin`). If you are using another account, substitute that account name in the examples that follow.
- You must have the super user (root) account password for both machines; by default, the `root` password is the same as the administrator password.
- Make sure that File Sharing and FTP Access are enabled for the Target machine. Use the Sharing panel in the System Preferences (Desktop > System Preferences) to enable FTP Access and File Sharing.

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

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

- If you are on a NetInfo network, you may only need to log in to both machines to see and use your KEXT.
- 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.

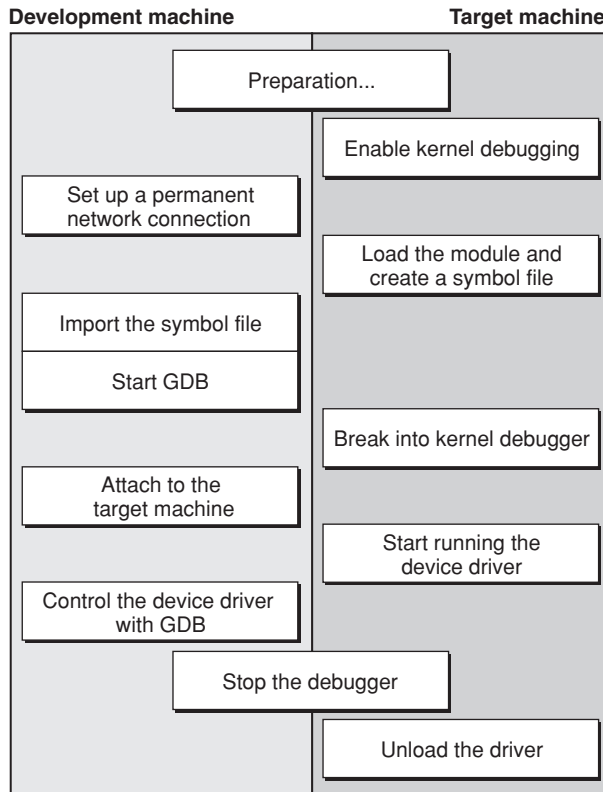
Roadmap

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.

Hello Debugger: Debugging a Device Driver With GDB

You will be moving back and forth between the two machines many times. [Figure 3-1](#) illustrates the steps you will take for each machine.

Figure 3-1 Steps to setup a debugging environment



After reading “[Preparation](#)” (page 43), 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 46)
2. “[On Development Machine, Set Up a Permanent Network Connection](#)” (page 47)

Hello Debugger: Debugging a Device Driver With GDB

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. [“On Target Machine, Create a Symbol File”](#) (page 50)
2. [“On Development Machine, Import the Symbol File”](#) (page 52)
3. [“On Development Machine, Start GDB”](#) (page 53)
4. [“On Target Machine, Break into Kernel Debugging Mode”](#) (page 54)
5. [“On Development Machine, Attach to the Target Machine”](#) (page 55)
6. [“On Target Machine, Start Running the Device Driver”](#) (page 56)
7. [“On Development Machine, Control the Driver With GDB”](#) (page 57)

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 [“On Target Machine, Create a Symbol File”](#) (page 50).

1. [“Stop the Debugger”](#) (page 58)
2. [“On Target Machine, Unload the Driver”](#) (page 59)

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

Hello Debugger: Debugging a Device Driver With GDB

2. You must assume special privileges as the super user (also known as `root`); only this account can enable kernel debugging.

At the prompt, enter the `su` command to become the super user. When prompted for a password, enter the administrator password for your computer. Note that nothing is displayed as you type the password.

For example:

```
% su
Password:
root#
```

3. Set the kernel debug flag.

If you're using a a Power Macintosh G4, blue & white Power Macintosh G3, or PowerBook G3 (FireWire, 400MHz or higher), enter this command:

```
# nvram boot-args="debug=0xe"
```

If you're using an older Power Macintosh, PowerMac G3 desktop or Mini Tower, or PowerBook G3 (without FireWire), enter this command:

```
# nvram boot-command="0 bootr debug=0xe"
```

If you are unsure which command to enter, enter both.

4. 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.

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

Hello Debugger: Debugging a Device Driver With GDB

1. Start the Terminal application. From a Desktop Finder window, locate and launch the Terminal application, found at `/Applications/Utilities/Terminal`.
2. You must assume special privileges as the super user (also known as `root`); only this account can change network connections.

At the prompt, enter the `su` command to enter super user mode. When prompted for a password, enter the administrator password for your computer. Note that nothing is displayed as you type the password.

For example:

```
% su
Password:
root#
```

3. Make sure your development machine can connect to your target machine:

Enter this command:

```
ping -c 1 target-machine
```

where *target-machine* is 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.

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
```

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

Enter this command:

```
arp -a
```


Hello Debugger: Debugging a Device Driver With GDB

This command 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.

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.

5. Remove the current, temporary, connection between your development and target machines.

Enter this command

```
arp -d target-machine
```

For example:

```
#a rp -d target.apple.com
target.apple.com (192.102.207.119) deleted
```

6. Create a new, permanent, connection between your development and target machines.

Enter this command:

```
arp -s target-machine ethernet-address
```

For example:

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

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

7. 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:

Hello Debugger: Debugging a Device Driver With GDB

```

nii133 (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

```

8. Exit from super user mode.

Enter this command:

```
exit
```

On Target Machine, 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.

Important

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

On the target machine, do the following:

1. From the login screen, log in as console.

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.

2. At the prompt, log in as `root`. Only the root account (also known as the super user) may load kernel extensions. Use the password you have set for the administrator account.

For example:

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

```

login: root
Password for root:
root#

```

3. Move to the directory that contains your KEXT *contents*.

Hello Debugger: Debugging a Device Driver With GDB

For example:

```
# cd /Users/admin/Projects/HelloIOKit/build/HelloIOKit.kext/Contents
```

4. For debugging purposes, you first load the driver (with the `kmodload` command) and create a symbol file for it; however you don't start the driver yet. You'll register the driver and start it running (using `kextload`) in a later step, “On Target Machine, Start Running the Device Driver” (page 56).

Important

In previous tutorials, [Hello Kernel: Creating a Kernel Extension With Project Builder](#) and [Hello IOKit: Creating a Device Driver With Project Builder](#), you used `kextload` to load your KEXT and start it running. The `kmodload` command is similar to `kextload`, but has less functionality. In particular, for a driver, the `kmodload` command merely loads the driver. It does not register the driver with the kernel nor does it start running the driver's code.

Load the driver (without starting it) and create a symbol file for it. Use the `kmodload` command.

For example, enter the command:

```
# kmodload -o /tmp/HelloIOKit.sym MacOS/HelloIOKit
```

The argument to `kmodload` must be the name of the *module*, not the KEXT bundle. The `-o` option tells `kmodload` to create a symbol file in `/tmp/HelloIOKit.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.

5. Move back to the directory that contains your KEXT.

For example:

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

or, as a shortcut, use this command to move up two levels in the directory hierarchy

```
# cd ../../..
```

On Development Machine, Import the Symbol File

On the development machine, import the symbol file from the target machine, for use with GDB. Use the `ftp` (file transfer protocol) command. A sample `ftp` session is shown below. Enter the `ftp` commands shown in boldface.

Important

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/HelloIOKit.sym
local: /tmp/HelloIOKit.sym remote: /tmp/HelloIOKit.sym
200 PORT command successful.
150 Opening BINARY mode data connection for '/tmp/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 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 /mach_kernel
```

For example:

```
% gdb /mach_kernel
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.

Enter this command at the GDB prompt:

```
add-symbol-file symbol-file-name
```

For example:

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

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

Enter this command at the GDB prompt:

```
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 two ways to break into kernel debugging mode.

- You can force a break into the debugger from the keyboard. You will use this method in this tutorial.
- 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')
```

To break into the debugger from the keyboard, 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”.

Important

You could reboot the machine if you hold the keys down for too long.

On the target machine, do the following:

1. Break into the debugger. Choose the appropriate key combination and hold the keys down for no more than three seconds:

On a **USB** keyboard, hold down the **Command** (or Apple) key and the **Power** button.

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

2. After three seconds, release the keys. The machine prints the following and waits for a remote debugger connection:

```
Debugger(Programmer Key)
```

```
Waiting for remote debugger connection.
Options..... Type
```

Hello Debugger: Debugging a Device Driver With GDB

```

-----
continue....  'c'
reboot.....  'r'

```

3. If you don't see the "Waiting..." message, hold down the keys for another count of three and try 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. Enter the following command at the GDB prompt, replacing *target-machine* with the actual machine name:

```
attach target-machine
```

For example

```
attach 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.

Hello Debugger: Debugging a Device Driver With GDB

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

For example:

```
(gdb) break 'com_MyTutorial_iokit_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_iokit_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:
3. The target machine is again able to accept keyboard input. Now you can run the driver.

```
(gdb) continue
```

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 directory containing your KEXT, using the `cd` command. For example

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


Hello Debugger: Debugging a Device Driver With GDB

2. Start the driver running, using the `kextload` command. Recall that you previously loaded the driver using `kmodload`, but `kmodload` does not register the driver with the kernel nor does it start running the driver. You must use `kextload` to finish the process and run the driver. For example:

```
# /kextload 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_iokit_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 some things you can try to do.

Hello Debugger: Debugging a Device Driver With GDB

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. Hold down the appropriate keys (see [“On Target Machine, Break into Kernel Debugging Mode”](#) (page 54)) for three seconds, then release the keys.

The target machine prints

```
Debugger(Programmer Key)
```

If you don't see the “Debugger...” message, hold down the keys for another count of three and try again.

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

```
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.

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:

```
# kextunload HelloIOKit.kext
IOCatalogueTerminate(Module com.MyTutorial.iokit.HelloIOKit) [0]
done.
```

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` command to read the manual pages for `kextload`, `kmodload`, `kmodstat`, `kextunload`, and `kmodunload`. For example, from a Terminal window, enter the command

```
man kmodstat
```

You may also want to familiarize yourself with GDB. Further information is available in `/Developer/Documentation/DeveloperTools/gdb`.

This tutorial is part of a series. Additional tutorials describe how to package a KEXT ([Packaging Your KEXT for Distribution and Installation](#)) and how to debug a kernel panic (“Don’t Panic: Debugging a Kernel Panic”). These tutorials, as well as the earlier (prerequisite) tutorials, can be found in `/Developer/Documentation/Kernel/Tutorials`.

C H A P T E R 3

Hello Debugger: Debugging a Device Driver With GDB

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).

Important

The information in this document is preliminary and may be subject to change.

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 an Installer program. The Installer takes care of providing users with many useful features such as descriptive information, default installation location, the ability to uninstall the software, and so forth.

Packaging Your KEXT for Distribution and Installation

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 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 installer needs to install (and optionally uninstall) your software. A basic package contains:

- A Bill of Materials (.bom)—this is a file in binary (non-text) format that describes the contents of the package.
- An information file (.info)—this text file contains the information entered in the Package Maker form when the package was created.
- An archive file (.pax)—this is an archive of the complete set of files which will be installed by the package. If the archive has been compressed, it will have an additional suffix (.gz). For this tutorial, the contents of your KEXT will compose the archive.
- A size calculation file (.sizes)—this is a text file that contains the compressed (and uncompressed) size of the unarchived software, to allow the installer to calculate required space.
- resources (optional)—packages may contain additional (optional) supplementary resources, such as icons, ReadMe files, licensing information, pre- or post-install scripts, and so forth. These are package resources, used (or run) at installation time; however, they will not be installed on disk along with the software.

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.

Packaging Your KEXT for Distribution and Installation

You should have completed and understood the material presented in a previous tutorial, [Hello Kernel: Creating a Kernel Extension With Project Builder](#) 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 IOKit: Creating a Device Driver With Project Builder](#). 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 64)
2. ["Create a Distribution Directory"](#) (page 65)
3. ["Gather Any Package Resources"](#) (page 66)
4. ["Create a Package with PackageMaker"](#) (page 68)
5. ["Examine the Package and Files"](#) (page 71)
6. ["Test Installing the Package"](#) (page 73)

Packaging Your KEXT for Distribution and Installation

7. “Test Uninstalling the Package” (page 75)
8. “Archive Your Package for Distribution” (page 76)

You’ll use the Terminal application to type the commands to locate your KEXT and create a distribution tree. You’ll use the Package Maker 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 Project Builder](#) or [Hello IOKit: Creating a Device Driver With Project Builder](#). 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.kext intermediates
```

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

Packaging Your KEXT for Distribution and Installation

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 kernel module and the module's class (if a driver), as stored in the KEXT's property list, should be unique. These should use the recommended "reverse-DNS" naming convention.

When your KEXT is installed under Mac OS X, any potential filename conflicts with other installed KEXTs will be resolved.

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.

Move up one directory, so that you are at the level that contains the `build` directory. Use the `cd` command. For example:

```
% cd ..
```

This command is a shortcut for "move back one level". In a BSD shell, `..` is synonymous for your parent 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. To create the package, you will first make a local "mirror" of this directory hierarchy, then copy your KEXT into the appropriate location. When the package is installed (unpacked), the installer will place the KEXT at the correct point in the user's file system.

1. Use the `mkdir` command to create the distribution directory hierarchy. For example:

```
% mkdir -p dstroot/System/Library/Extensions
```

This creates a directory tree, "rooted" at the directory `dstroot` (distribution root) in the current location. The `-p` option causes `mkdir` to create intermediate directories on the path as required. The new directory tree mirrors the structure (`System/Library/Extensions`) into which the KEXT will be installed.

Packaging Your KEXT for Distribution and Installation

2. Copy your KEXT into the new directory structure using the `cp` command. For example:

```
% cp -R build/Hello.kext dstroot/System/Library/Extensions
```

The `-R` option causes `cp` to copy a directory recursively, copying all files contained in that directory as well.

3. 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/System
dstroot/System/Library
dstroot/System/Library/Extensions
dstroot/System/Library/Extensions/Hello.kext
dstroot/System/Library/Extensions/Hello.kext/Contents
dstroot/System/Library/Extensions/Hello.kext/Contents/Info.plist
dstroot/System/Library/Extensions/Hello.kext/Contents/MacOS
dstroot/System/Library/Extensions/Hello.kext/Contents/MacOS/Hello
dstroot/System/Library/Extensions/Hello.kext/Contents/PkgInfo
dstroot/System/Library/Extensions/Hello.kext/Contents/Resources
dstroot/System/Library/Extensions/Hello.kext/Contents/Resources/...
dstroot/System/Library/Extensions/Hello.kext/Contents/Resources/...
dstroot/System/Library/Extensions/Hello.kext/Contents/version.plist
```

You can see how the destination directory hierarchy mirrors the system hierarchy where the KEXT will be installed. Instead of `dstroot/System/Library...`, the installer will install into `/System/Library...`

Gather Any Package Resources

As described in “[Anatomy of a Package](#)” (page 61), 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.

Packaging Your KEXT for Distribution and Installation

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

The supplementary resources will not be installed along with your KEXT, so they should not be placed in the distribution directory. Instead, create a resources directory at the same level as `dstroot`. Use the `mkdir` command. You don't need the `-p` option this time because you are only creating one level of directory. For example:

```
% mkdir resources
```

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 two files, `ReadMe.rtf` and `License.rtf`, in the resources directory. Move to the resources directory; use the `cd` command. For example:

```
% cd resources
```

Create a ReadMe File

If a ReadMe file is present in a package, the Mac OS X Installer will display the contents of the file. A user dismisses the ReadMe by pressing "Continue". The presence of a `ReadMe.rtf` file is all that is needed to get this feature. The Installer will automatically add a "Read Me" line to the bullet list presented in the Left column of the application.

1. Start the Text Edit application. From a Desktop Finder window, locate and launch the Text Edit application, located at `/Applications/TextEdit`.
2. Enter the text of your ReadMe file. Generally, a ReadMe file should describe the contents of your package, version information, and any additional information a user might need to see. For example, a ReadMe file could provide contact information, special hardware instructions, any known incompatibilities with other extensions, and so forth.
3. Save your ReadMe file. Use the name `ReadMe.rtf`. As the location (Where:), choose the `resources` directory you created in the previous step.
4. Close the ReadMe file.

Create a Software Licence File

If a License file is present in a package, the Mac OS X Installer will display the contents of the file. A user dismisses the License panel by pressing "Continue". The Installer then displays a panel 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. The Installer will automatically add a "License" line to the bullet list presented in the Left column of the application.

1. Use the Text Edit 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 `resources` directory you chose in the previous step.
4. Close the License file.

Create a Package with PackageMaker

Now you can use Package Maker to create a package, including all of the files in your distribution and resources directories.

1. Start the Package Maker application. From a Desktop Finder window, locate and launch the Package Maker application, located at `/Developer/Applications`.
2. For "Package Root Directory", click Change Root and choose the destination directory you created in "[Create a Distribution Directory](#)" (page 65). For example:

```
/Users/admin/Projects/Hello/dstroot
```

3. For "Package Resources Directory" click Set and choose the resources directory you created in "[Gather Any Package Resources](#)" (page 66). For example:

```
/Users/admin/Projects/Hello/resources
```

Packaging Your KEXT for Distribution and Installation

4. Fill in the “Package Information” fields as you like, choosing an appropriate title (approximately 30 characters or less) and description. The “Disk Name” field is used if your package will be distributed on disk; for this tutorial, you can use the package title here as well. The “Delete Warning” is a string that will be printed if a user tries to uninstall this package.
5. Leave the “Default Location” set to `/`.
6. In the “Package Flags” section, check the box for “Needs Authorization”. You must be the authorized administrator of the computer to install a kernel extension. You may optionally wish to check the box for “Install Only” if your package will not have an uninstall option. Leave the rest of the boxes unchecked.
7. Check the box labeled “Compress package”.

Figure 4-1 illustrates a sample Package Maker dialog.

Figure 4-1 Create a package using Package Maker

When you are finished, click the button labeled “Create Package” to create the package.

Packaging Your KEXT for Distribution and Installation

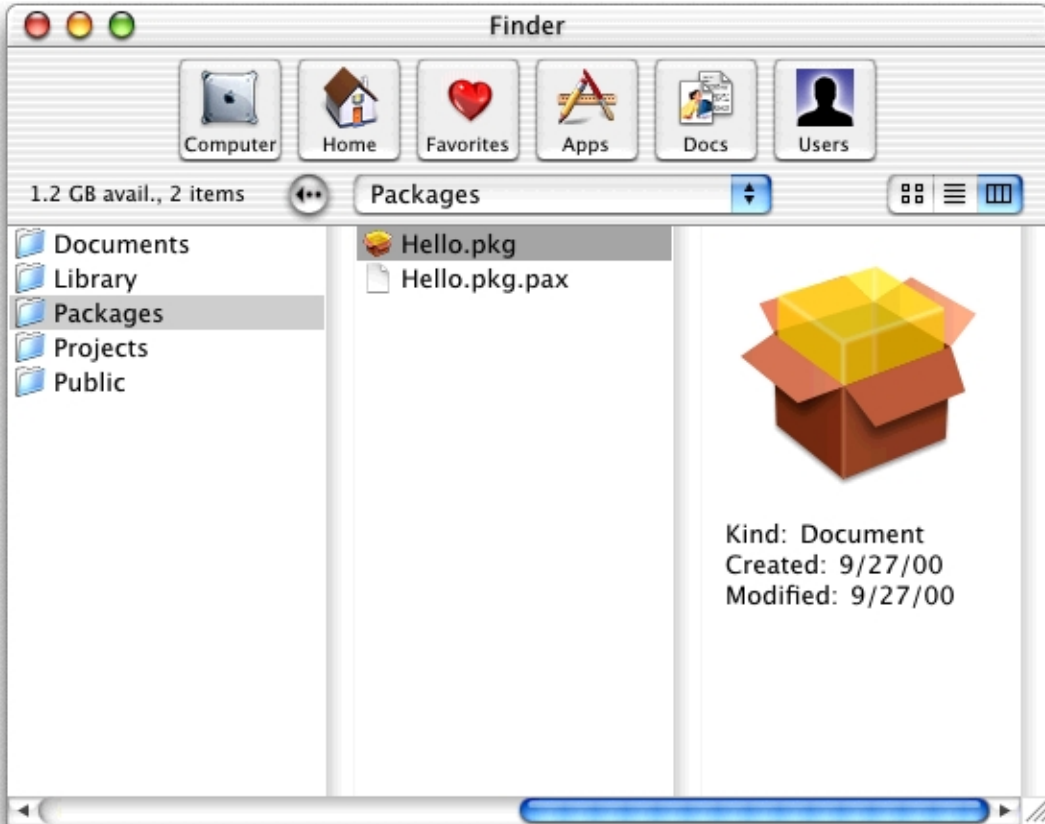
Package Maker will ask you what name to save the package under and where to store it.

1. Choose a name for your package; the best name is one which incorporates the name of your KEXT. That is, if your KEXT was `Hello.kext`, name your package `Hello.pkg`.
2. Choose a location. The default location is your home folder. You may want to make a new folder called Packages to use for all packages you create.
3. When you have chosen a name and location, click the Package button. Package Maker prints some status. When it has finished, click Continue. You may now close the Package Maker dialog.

Examine the Package and Files

You can now examine your new package.

1. From a Desktop Finder window, navigate to the location where you told Package Maker to save your package. For example, if you named the package `Hello.pkg` and told Package Maker to store it in the Packages folder in your home folder, move to the Packages folder as shown in [Figure 4-2](#).

Figure 4-2 Locate your package on the Desktop

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.

Move into the package directory and list its contents with the `ls` command. For example:

Packaging Your KEXT for Distribution and Installation

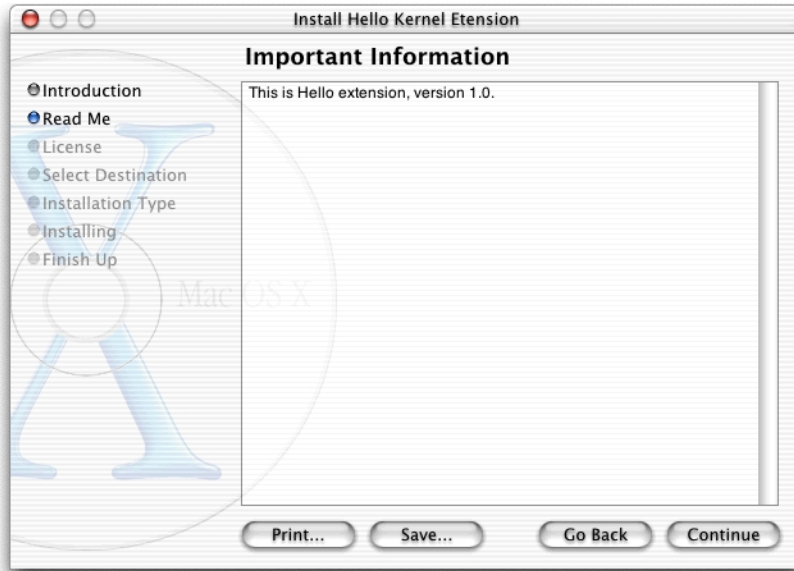
```
% cd Hello.pkg
% ls
Hello.bom  Hello.info  Hello.pax.gz  Hello.sizes  License.rtf  ReadMe.rtf
```

Refer to [“Anatomy of a Package”](#) (page 61) and [“Gather Any Package Resources”](#) (page 66) for descriptions of these files and their contents.

Test Installing the Package

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

1. From a Desktop Finder window, navigate to the location where you told Package Maker to save your package (repeat step #1 in [“Examine the Package and Files”](#) (page 71)).
2. Select and double-click the package file.
3. The Mac OS installer launches.
4. Click the lock and enter the Administrator password to install the software.
5. The Introduction panel will be displayed. Click Continue.
6. If you created a ReadMe file, you should see it displayed with the heading “Important Information” as shown in [Figure 4-3](#). Click Continue.

Figure 4-3 Installer Displays ReadMe

7. If you created a License file, you should see it displayed with the heading "Software License Agreement". Click Continue. The Installer will display a panel requiring you to agree (or disagree) to the terms of the license. Click Agree.
8. If appropriate, select a destination volume and click Continue.
9. Click Install and allow the Installer to proceed. Click Close when the 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 ReadMe and software license files were not installed.

Test Uninstalling the Package

When the Mac OS installer installs a package, it leaves a **receipt**. Receipts are implemented as packages as well. A receipt contains several of the files the installed package contained, but they do not contain the data archive.

Receipts can be used later to determine what was installed. When double-clicked, a receipt document launches the installer to uninstall the appropriate package.

1. From a Desktop Finder window, navigate to `/Library/Receipts`. This location contains a package, `Hello.pkg`.
2. From a shell window in the Terminal application, move to the directory that contains the receipt. For example:

```
% cd /Library/Receipts
```

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

Move into the `Hello.pkg` directory and list its contents with the `ls` command. For example:

```
% cd Hello.pkg
% ls
Hello.bom      Hello.info  Hello.sizes
```

Important

The Installer is unable to uninstall packages properly in the Beta release of Mac OS X. The steps below may fail.

3. From the Finder window, select and double-click the receipt package, `Hello.pkg`.
4. The Mac OS installer launches.
5. Click the lock and enter the Administrator password to uninstall the software.
6. Click Continue and allow the installer to uninstall the package.

Archive Your Package for Distribution

Because packages are implemented as bundles, their contents are subject to possible modification or damage 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.

You should feel free to use any archiving and compression tools you prefer. This tutorial suggests one possibility.

Important

Future versions of the Package Maker application may include the option to archive a package.

The `pax` command, included with Mac OS X, creates archive files in a format suitable for expansion with tools such as Aladdin's StuffIt Expander (included with Mac OS X). Recall that the package created with Package Maker contained a file named `Hello.pax.gz`. Package Maker uses the `pax` format internally to archive the data stored in a package.

The following steps demonstrate how to archive your package using `pax`.

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

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

Use the `pax` command to create an archive of the package. For example:

```
% pax -wf Hello.pkg.pax Hello.pkg
```

The `w` option tells `pax` to write an archive. The `f` option names the file which will contain the archive.

2. When the `pax` command completes, you will see a file named `Hello.pkg.pax`. This archive can be compressed (or not) and can be posted to the Web (for example) for downloading, or distributed by other means.
3. Test opening the archive by using Aladdin StuffIt Expander, or another utility of your choice. StuffIt Expander can be found in `/Applications/Utilities/Aladdin`.

Where to Go Next

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

This tutorial is part of a series. Additional tutorials describe how to debug a kernel extension ([Hello Debugger: Debugging a Device Driver With GDB](#)) and how to debug a kernel panic (“Don’t Panic: Debugging a Kernel Panic”). These tutorials, as well as the earlier (prerequisite) tutorials, can be found in `/Developer/Documentation/Kernel/Tutorials`.

If you’re interested, you can use the `man` command to read the manual pages for `pax`. For example, from a Terminal window, enter the command

```
man pax
```

C H A P T E R 4

Packaging Your KEXT for Distribution and Installation