Hello Kernel: Creating a Kernel Extension With Project Builder

This document describes how to write and build an kernel extension for Mac OS X. The extension you'll create is a simple HelloKernel extension that prints text messages when loading and unloading.

The next tutorial "Hello IOKit: Creating a Device Driver With Project Builder" (page 15) describes how to build and debug a device driver, a special kind of kernel extension that tells the kernel how to handle a device. It emphasizes how to write the driver's code and perform remote debugging with GDB.

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

1. "Create the Kernel Extension Project" (page 1)
2. "Build the Kernel Extension" (page 10)
3. "Running the Kernel Extension" (page 11)

Create the Kernel Extension Project

This section describes how to create the project that will create your kernel extension.

To better understand what you're doing in this section, it helps to know what's inside an kernel extension. In Mac OS X, a kernel extension is implemented as a bundle, a folder that the Finder treats as a single file. It can contain the following:

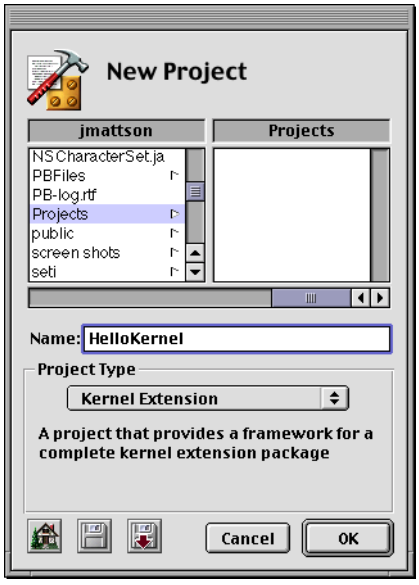
- Info-macos.xml describes the kernel extension's contents, settings, and requirements. It's a text file in XML format. An extension can contain nothing more than an Info-macos.xml file.
- Modules are the extension's binary code. This is what's loaded into the kernel. Generally, a kernel extension has only one, but it can have more or none. If it has none, its Info-macos.xml file would reference a module in another extension and change its default settings.
- Resources are useful if your extension needs to display a dialog box or menu.

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

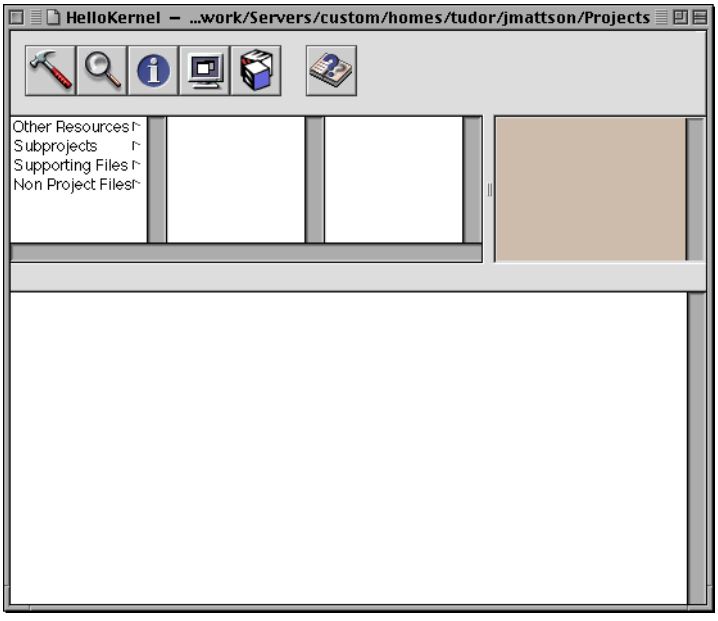
1. "Create a Kernel Extension Project" (page 3)
2. "Create a Kernel Module Subproject" (page 4)
3. "Edit the Module's CustomInfo.xml file" (page 5)
4. "Create a Source File" (page 8)
5. "Implement the Needed Methods" (page 8)

Create a Kernel Extension Project

Choose Project > New. For the name, enter “HelloKernel”. For the Project Type, choose Kernel Extension.



Project Builder creates the new project and displays its project window.

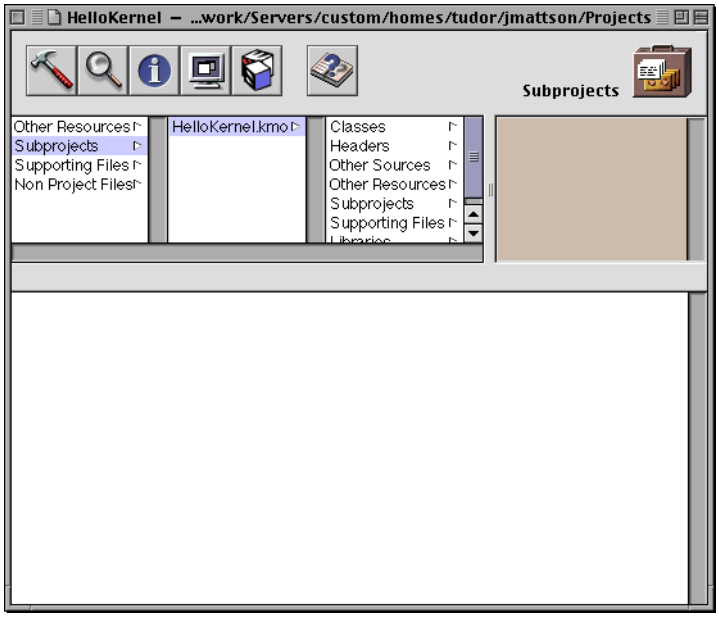


Create a Kernel Module Subproject

The kernel module is the binary code the extension executes. Usually, a extension has only one, but can have more.

Choose Project > New Subproject. For the name, enter “HelloKernel”. For the Project Type, choose Kernel Module.

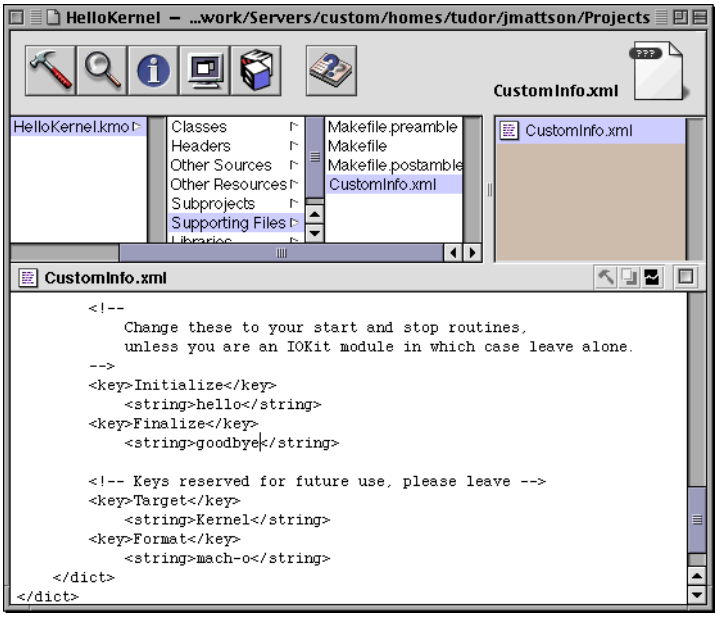
Project Builder creates the new subproject. To see what's in the subproject, click the word Subprojects in the project window, then click HelloKernel.kmodproj.



Edit the Module's CustomInfo.xml file

Your project contains two files named CustomInfo.xml, which tell the operating system what your driver contains and what it needs. In the next tutorial, you'll learn more about them. In this tutorial, you'll just edit the module's CustomInfo.xml.

To find the module's CustomInfo.xml file, go to the Project Window's browser and follow this path: Subprojects / HelloKernel / Other Resources / CustomInfo.xml.



Edit two of the lines in this file. Change the Initialize entry so its value is `<string>hello</string>` and change the Finalize entry so its value is `<string>goodbye</string>` . The listing below shows the modified file, with the changed lines highlighted.

Important: Don't change the text between the `<key>` and `</key>` markers. Instead, change the text between `<string>` and `</string>`.

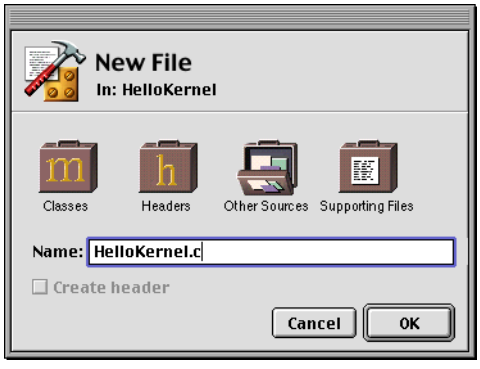
Listing 1-1 The module's CustomInfo.xml file, after editing

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist
 SYSTEM "file://localhost/System/Library/DTDs/PropertyList.dtd">
<plist version="0.9">
<!-- Welcome to XML.  Anything you see surrounded like this is a comment. -->
<dict>
```

```
<key>Module</key>
    <dict>
        <!-- You probably should leave the next two keys alone -->
        <key>Name</key>
            <string>HelloKernel</string>
        <key>File</key>
            <string>HelloKernel</string>

       <!-- Delete this key if this module doesn't require another module -->
        <key>Requires</key>
            <array></array>

        <!-- Change these to your start and stop routines,
        unless you are an IOKit module in which case leave alone.
        -->
        <key>Initialize</key>
            <string>hello</string>
        <key>Finalize</key>
            <string>goodbye</string>

        <!-- Keys reserved for future use, please leave -->
        <key>Target</key>
            <string>Kernel</string>
        <key>Format</key>
            <string>mach-o</string>
    </dict>
</dict>
</plist>
```

Create a Source File

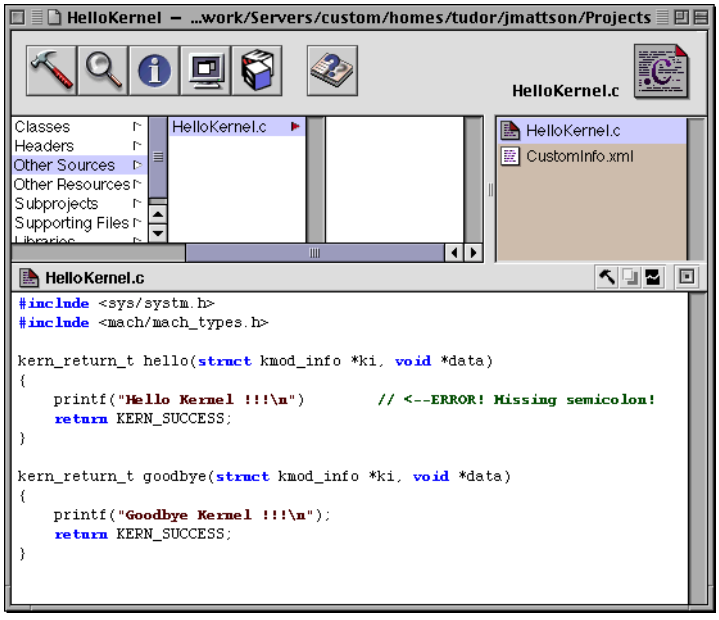
Choose File > New in Project. Choose the Other Sources suitcase and entry HelloKernel.c as the file's name.



Implement the Needed Methods

Replace the contents of HelloKernel.c with the following code. You can copy and paste the code from this document if you're reading it online.

This is where you'll find the HelloKernel.c file in the project window:



And this is the code for HelloKernel.c. Note that it contains a small typo. You'll correct the typo in the next section.

Listing 1-2 HelloKernel.c

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

kern_return_t hello(struct kmod_info *ki, void *data)
{
    printf("Hello Kernel !!!\n")      // <--ERROR! Missing semicolon!
    return KERN_SUCCESS;
}

kern_return_t goodbye(struct kmod_info *ki, void *data)
{
    printf("Goodbye Kernel !!!\n");
```

```
    return KERN_SUCCESS;
}
```

Build the Kernel Extension

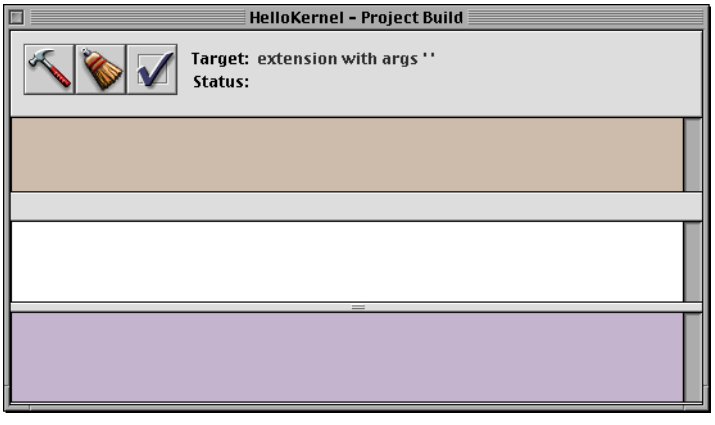
Here's how to build the kernel extension:

1. "Build the Project" (page 10)
2. "Fix Any Errors" (page 11)
3. "Build the Project Again" (page 11)

Build the Project

Click the Build button in the Project Window, which looks like the one below:

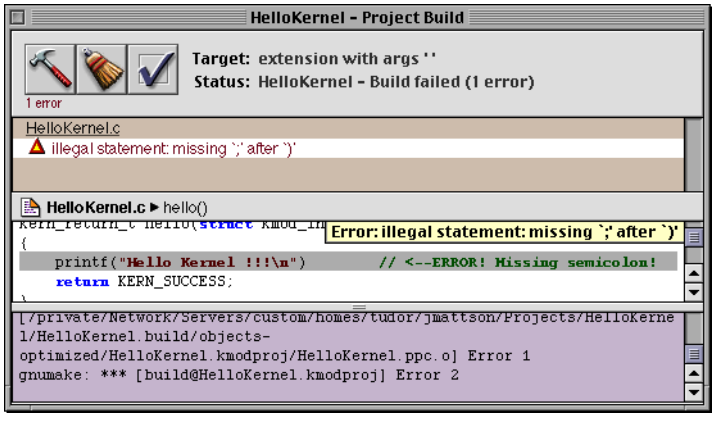
After the Build panel appears, click its Build button. If Project Builder asks you whether to save some modified files, select all the files and click "Save and build".



Project Builder starts building your project and stops when it reaches the error.

Fix Any Errors

Click the build error that appears in the top view of the Build panel. The source code for the error appears in the editing view below it.



You can edit the source code right inside the Build panel. Correct the error now by inserting a semicolon at the end of the statement.

Build the Project Again

Click the Build button in the Build panel. When Project Builder asks you whether to save some modified files, select all the files and click “Save and build”.

Project Builder starts building your project again, and this time it succeeds.

Running the Kernel Extension

This section shows how to run the kernel extension with the `kextload` command. Here’s how you’ll do it:

1. Log in as Root in Console Mode

2. Load and Unload the Kernel Extension

Log in as Root in Console Mode

You'll log in as root, since only the root account can load kernel extensions. And you'll run in console mode, since console mode writes all system messages (like the kernel extension's message "Hello Kernel!!!") directly to your monitor, instead of to the file /var/log/system log.

1. Log out of your account.

   From the Workspace, choose Log Out from the File menu.

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

   Type "console" as the user name, leave the password blank, and press Return. The screen turns black and looks like an old ASCII terminal. This is the console mode.

3. Log in as root.

   Only the root account may load kernel extensions.

   For example:

```
4BSD (dev) (console)

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

Load and Unload the Kernel Extension

1. Move to the directory that contains your module.

   For example:

```
dev# cd ~me/Projects/HelloKernel/HelloKernel.kext
```

2. Load the extension.

   Enter this command:

   `kmodload` ***module-name***

This command loads your extension by running its initialization function.

For example:

```
dev# kmodload HelloKernel
Hello Kernel !!!
```

3. Unload the extension

   Enter this command:

   `kmodunload` ***module-name***

   This command unloads your extension by running its finalization function.

   For example:

   ```
   dev# kmodunload HelloKernel
   Good-bye Kernel !!!
   ```

   You've now written, built, loaded, and unloaded your own kernel extension. Congratulations!