CHAPTER 2

Hello IOKit: Creating a Device Driver With Project Builder

This document describes how to write and debug an I/O Kit device driver for Mac OS X. It emphasizes how to edit the driver's XML file, write a its code, and use GDB to perform remote debugging. The driver you'll create is a simple HelloIOKit driver that prints text messages but doesn't actually control any hardware.

For remote debugging, you need two machines. On the development machine, you'll write and build the driver, and then run the debugger. On the target-machine, you'll load and run the driver. Both machines must be running the same version of Mac OS X, must be connected via TCP/IP, and must be on the same subnet.

The previous tutorial "Hello Kernel: Creating a Kernel Extension With Project Builder" (page 1) emphasizes how to create the kernel extension's project and correct bugs.

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

1. "Create and Build the Device Driver Project" (page 15)
2. "Debug the Device Driver" (page 29)

Create and Build the Device Driver Project

This section describes how to create the project that will create your device driver.

An I/O Kit device driver is a special type of kernel extension 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. A driver can contain the following:

- Info-macos.xml describes the driver's contents, settings, and requirements. It's a text file in XML format. A driver can contain nothing more than an Info-macos.xml file.
- Modules are the driver's binary code. This is what's loaded into the kernel. Generally, a driver 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 driver and change its default settings.
- Resources are useful if your driver needs to display a dialog box or menu.

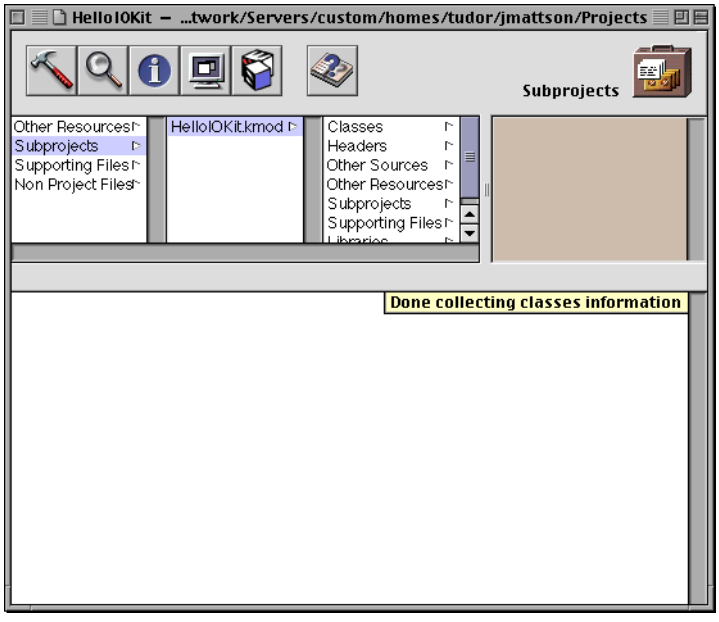
Here's how you'll create the device driver project:

1. "Create a Device Driver Project and a Module Subproject" (page 16)
2. "Edit the Driver's CustomInfo.xml file" (page 17)
3. "Edit the Module's CustomInfo.xml file" (page 20)
4. "Create a Source File and Header File" (page 23)
5. "Implement the Needed Methods" (page 23)
6. "Build the Project" (page 28)

Create a Device Driver Project and a Module Subproject

To create the device driver project, choose Project > New, enter "HelloIOKit" as the name, and Kernel Extension as the Project Type.

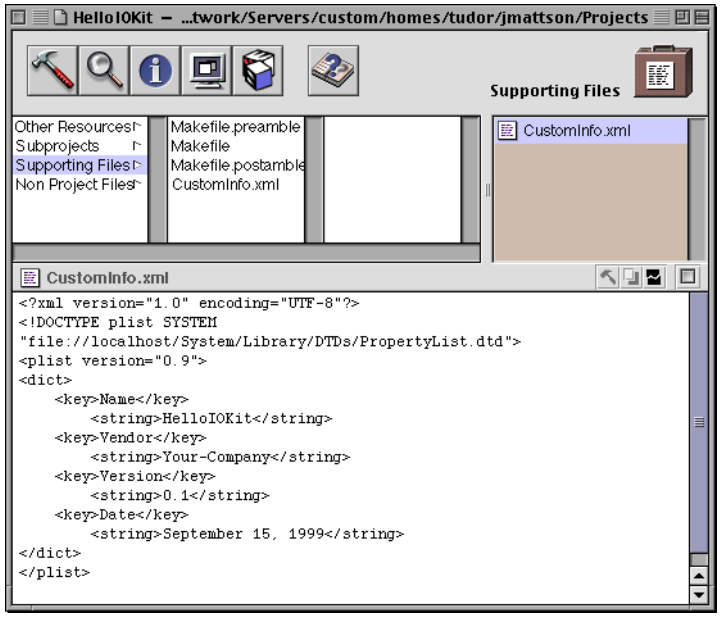
To create the module's subproject, choose Project > New Subproject, enter "HelloIOKit" as the name, and choose Kernel Module as the Project Type.



Edit the Driver's CustomInfo.xml file

Your project contains two files named CustomInfo.xml: one in the driver project and the other in the module subproject. These files tell the operating system what your driver contains and what it needs. When Project Builder builds your project, it will combine all its CustomInfo.xml files into one file named Info-macos.xml.

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



This CustomInfo.xml file contains a dictionary with four entries.

- Name: The name used for the driver's bundle. By default it's the name of the project.
- Vendor: Your company's name.
- Version: The driver's version number. It can contain any combination of numbers or characters.
- Date: The date the driver was created.

Here's the CustomInfo.xml file for this driver:

Listing 2-1 The Driver's CustomInfo.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist
 SYSTEM "file://localhost/System/Library/DTDs/PropertyList.dtd">
```

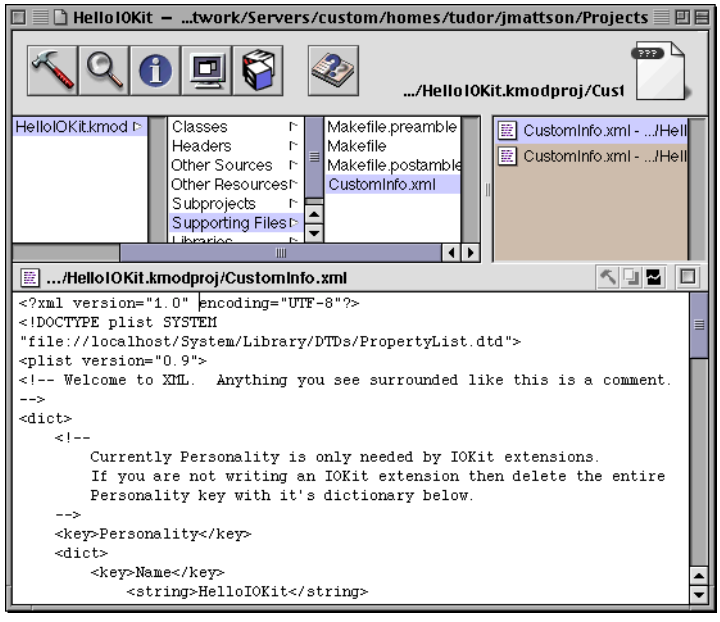
```
<plist version="0.9">
<dict>
    <key>Name</key>
        <string>HelloIOKit</string>
    <key>Vendor</key>
        <string>Your-Company</string>
    <key>Version</key>
        <string>0.1</string>
    <key>Date</key>
        <string>September 15, 1999</string>
</dict>
</plist>
```

For this tutorial, you don't need to change any of the entries. However, you can change the vendor, version, and date, if you like. Leave the name as it is.

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

Edit the Module's CustomInfo.xml file

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



This CustomInfo.xml file contains two dictionaries. For now, don't concern yourself with the first one, called "personality." Instead, concentrate on the second one, called "module," which contains six entries:

- Name: The official name for this module. The operating system will use this to identify the module.
- Target: What the module will be linked against. In this version, the only option is `kernel`.
- Initialize: What the module will call when it's loaded. Don't change this if you're writing an IOKit module
- Finalize: What the module will call when it's unloaded. Don't change this if you're writing an IOKit module
- File: The name of the module's file. It does not have to be the same as the above name, although it generally is.

- Format: The binary format for the module. In this version, the only option is `mach-o`.
- Requires: An array of other drivers that this one depends upon. It can have zero or more members.

For this tutorial, leave the name, file, initialize, finalize, target, and format entries as they are.

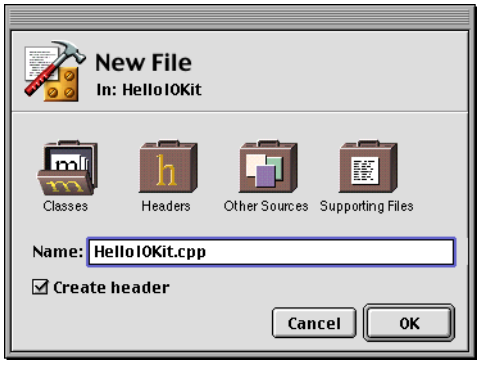
Listing 2-2 The module's CustomInfo.xml file, after editing

```
<?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>
    <!--
        Currently Personality is only needed by IOKit extensions.
        If you are not writing an IOKit extension then delete the entire
        Personality key with it's dictionary below.
    -->
    <key>Personality</key>
    <dict>
        <key>Name</key>
            <string>HelloIOKit</string>
        <key>Module</key>
            <string>HelloIOKit</string>

        <!-- The name of the class for registry to 'alloc' when probing. -->
        <key>IOClass Names</key>
            <string>HelloIOKit</string>

       <!-- IOKit matching properties, setup for a debug node by default. -->
        <key>IOImports</key>
            <string>IOResources</string>
        <key>IONeededResources</key>
            <string>IOKit</string>
        <key>IOMatchCategory</key>
            <string>HelloIOKit</string>
    </dict>
```

```
    <key>Module</key>
    <dict>
        <!-- You probably should leave the next two keys alone -->
        <key>Name</key>
            <string>HelloIOKit</string>
        <key>File</key>
            <string>HelloIOKit</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>IOKitRelocStart</string>
        <key>Finalize</key>
            <string>IOKitRelocStop</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 and Header File

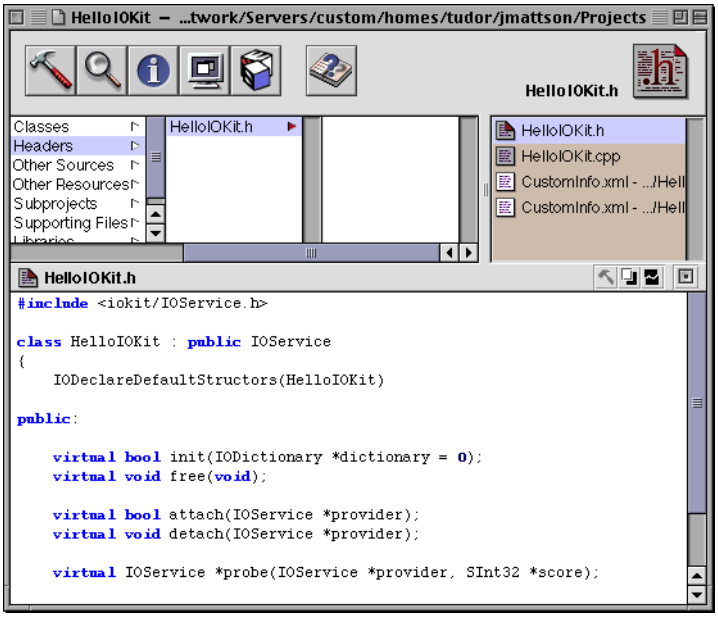
Choose File > New in Project. For the file's name, enter HelloIOKit.cpp. Choose the Classes suitcase, and turn on the "Create header" option.



Implement the Needed Methods

Replace the contents of HelloIOKit.h and HelloIOKit.cpp 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 HelloIOKit.h file in the project window:



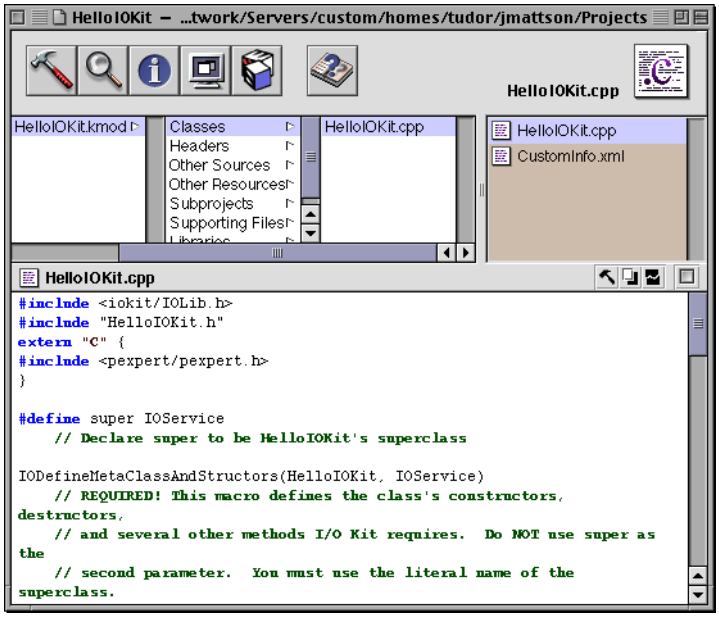
And this is the code for HelloIOKit.h:

Listing 2-3 HelloIOKit.h

```
#include <IOKit/IOService.h>

class HelloIOKit : public IOService
{
    OSDeclareDefaultStructors(HelloIOKit)

public:

    virtual bool init(OSDictionary *dictionary = 0);
    virtual void free(void);

    virtual bool attach(IOService *provider);
    virtual void detach(IOService *provider);
```

```
    virtual IOService *probe(IOService *provider, SInt32 *score);

    virtual bool start(IOService *provider);
    virtual void stop(IOService *provider);
};
```

This is where you'll find the HelloIOKit.cpp file in the project window:



And this is the code for HelloIOKit.cpp. Note that the `init` method contains a call to `PE_enter_debugger`. When you're finished debugging your driver, you can remove this call

Listing 2-4 HelloIOKit.cpp

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

```
#define super IOService
    // Declare super to be HelloIOKit's superclass

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

bool HelloIOKit::init(OSDictionary *dictionary)
{
    PE_enter_debugger("Debug");  // Remove this when you're done debugging.
    bool res = super::init(dictionary);
    IOLog("Initializing\n");

    return res;
}

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

bool HelloIOKit::attach(IOService *provider)
{
    bool res = super::attach(provider);
    IOLog("Attaching\n");

    return res;
}

void HelloIOKit::detach(IOService *provider)
{
    IOLog("Detaching\n");
    super::detach(provider);
}

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

```
    return res;
}

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

    return res;
}

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

There are a couple macros used in these files that are very important. If you don't use these macros in the proper places, your driver won't work properly:

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

The following list describes the entry points that the I/O Kit uses. 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 defines only the `start` and `stop` methods, and uses its superclass's definitions for the rest.

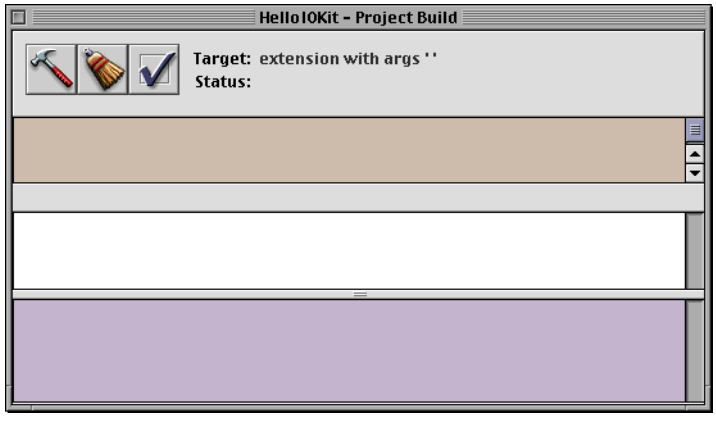
- `init` is the first method called when your driver is loaded (or unloaded); `free` is the last method called when your driver is unloaded. Note that neither method can talk to the hardware. They can only create or release data structures.
- `attach` is called when your driver is inserted in the registry; `detach` is called when it's removed.

- `probe` 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.
- `start` is the last method called when your driver is loaded; `stop` is the first method called when your driver is unloaded. They can talk to the hardware and create or release data structures.

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.

Debug the Device Driver

This section describes how to debug your device driver. The debugger you'll use is GDB, a command-line debugger.

Here's how you'll debug the driver:

1. "On Target Machine, Enable Kernel Debugging" (page 29)
2. "On Development Machine, Connect to the Target Machine" (page 30)
3. "On Target Machine, Load the Device Driver" (page 32)
4. "On Development Machine, Load the Debugger" (page 33)
5. "On Target Machine, Start Running the Device Driver" (page 34)
6. "On Development Machine, Start Debugging the Device Driver" (page 35)
7. "Stop the Debugger" (page 36)

The steps in this section include terminal output from an sample session in which the development machine is called dev.my-company.com and the target machine is called target.my-company.com.

On Target Machine, Enable Kernel Debugging

By default, the System does not let you debug the kernel. So before you can debug your driver, you must enable kernel debugging.

On the target machine, do the following:

1. From the login screen, log in as root.
2. Start the Terminal application.

   It's in the Apple menu, under Server Administration.
3. Set the kernel debug flag.

   If you're using a Blue & White Power Macintosh G3 or a Power Macintosh G4, enter this command:

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

If you're using any other Macintosh, enter this command:

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

4. Reboot the computer.

   Enter this command:

   ```
   target# reboot
   ```

   The computer reboots and displays the login screen.

On Development Machine, Connect to the Target Machine

Your development and target machines must be continuously connected by a reliable network connection. This section creates such a connection.

On the development machine, do the following:

1. Start the Terminal application.

   It's in the Apple menu, under Server Administration.

2. From the command line, log in as root.

   Enter this command:

   ```
   su
   ```

   Enter the root password when it asks.

   For example:

   ```
   dev> su
   Password:
   dev#
   ```

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, such as `target.my-company.com`.

This command makes sure your development machine can reach your target machine and creates a temporary connection between them.

For example:

```
dev# ping -c 1 target.my-company.com
PING target.my-company.com (17.203.35.103): 56 data bytes
64 bytes from 17.203.35.103: icmp_seq=0 ttl=255 time=2.099 ms

--- target.my-company.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. Note the hardware Ethernet address of the target machine.

   Enter this command:

   ```
   arp -a
   ```

   This command lists all the Ethernet addresses of all machines your development machine has recently connected to. Note the entry that contains your target machine's Ethernet address.

   For example:

   ```
   dev# arp -a
   niil33 (17.203.34.250) at 0:0:f:0:86:c3
   target (17.203.35.103) at 0:5:2:b0:b3:20
   dev (17.203.35.104) at 0:5:2:59:2f:20 permanent
   ```

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

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

   Enter this command

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

   For example:

   ```
   dev# arp -d target.my-company.com
   target.my-company.com (17.203.35.103) deleted
   ```

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

   Enter this command:

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

For example:

```
dev# arp -s target.my-company.com 0:5:2:b0:b3:20
```

7. Make sure the connection was made correctly.

   Enter this command:

   ```
   arp -a
   ```

   Make sure the entry for your target machine contains the word “permanent.”

   For example:

   ```
   root# arp -a
   niil33 (17.203.34.250) at 0:0:f:0:86:c3
   target (17.203.35.103) at 0:5:2:b0:b3:20 permanent
   dev (17.203.35.104) at 0:5:2:59:2f:20 permanent
   ```

8. Log out as root.

   Enter this command:

   ```
   exit
   ```

   For example:

   ```
   dev# exit
   dev>
   ```

On Target Machine, Load the Device Driver

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. The screen turns black and looks like an old ASCII terminal.

2. Log in as root.

   For example:

   ```
   4BSD (target) (console)

   login: root
   ```

```
Password for root:
target#
```

3. Move to the directory that contains your module.

   For example:

   ```
   target# cd ~me/Projects/HelloIOKit/HelloIOKit.kext
   ```

4. Load the driver and create a sym-file for it.

   Enter this command:

   `kmodload -o /`***module-name***`.sym` ***module-name***

   This command loads your driver and then creates a sym-file named *module-name*`.sym`, which contains information that the debugger needs to debug your driver.

   For example:

   ```
   target# kmodload -o /HelloIOKit.sym HelloIOKit
   ```

5. Copy the sym-file to your driver's directory.

   Log in to your user account with `su`, then copy the symbol file to your driver's directory.

   For example:

   ```
   target# su me
   Password:
   target> cp /HelloIOKit.sym ~me/Projects/HelloIOKit/HelloIOKit.kext/
   target> exit
   target#
   ```

On Development Machine, Load the Debugger

On the development machine, do the following from the Terminal application:

1. Move to the directory that contains your module.

   For example:

   ```
   dev> cd ~me/Projects/HelloIOKit/HelloIOKit.kext
   ```

2. Start the debugger on the kernel.

   Enter this command:

```
gdb /mach_kernel
```

This command can take as long as five minutes.

For example:

```
dev> gdb /mach_kernel
GNU gdb 4.18-19990707 (Apple version gdb-141.4)
Copyright 1998 Free Software Foundation, Inc.
...
Reading symbols from "/mach_kernel"...(no debugging symbols found)...done.
(gdb)
```

3. Load the symbol file you created.

   Enter this command at the GDB prompt:

   `add-symbol-file` ***module-name*****`.sym`**

   For example:

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

4. Help GDB find your source files.

   Enter this command at the GDB prompt:

   `path` ***path-name***

   Where *path-name* is the directory where your module's source files are.

   For example:

   ```
   (gdb) path ~me/Projects/HelloIOKit/HelloIOKit.kmodproj/
   ```

5. Let the debugger know you're remotely debugging a device driver.

   Enter this command at the GDB prompt:

   ```
   target remote-kdp
   ```

On Target Machine, Start Running the Device Driver

Now you can start the driver running. On the target machine, start the driver running. Enter this command:

`kextload` ***driver-name***

For example:

```
target# kextload ../HelloIOKit.kext
```

The machine prints the following and waits for a remote debugger connection:

```
Debugger(Debug)

Waiting for remote debugger connection.
Options.....   Type
------------   ----
continue....   'c'
reboot......   'r'
```

On Development Machine, Start Debugging the Device Driver

Now that the debugger has been loaded on the development machine, and the target machine is waiting for a debugger connection, you can actually start debugging.

On the development machine, do the following:

1. Attach to the target machine.

   Enter the following command at the GDB prompt:

   `attach` ***target-machine***

   The target machine prints:

   ```
   Connected to remote debugger.
   ```

2. Set any necessary breakpoints.

   This is the best place to set your breakpoints. For this tutorial, set a breakpoint at the `start` method. For example:

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

3. On the development machine, let the target machine continue. Enter the following command at the GDB prompt:

   ```
   continue
   ```

   The driver executes until it hits the breakpoint.

For example:

```
(gdb) continue
Continuing.

Breakpoint 1, 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 as you would any other executable.

Since 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 long since you must halt kernel to use GDB. In a short time, internal inconsistencies may appear that cause the target kernel to panic or wedge, forcing your to reboot.

For more information on GDB, use it's `help` command. Here are some things you can try to do.

1. Step into the superclass's init method.
2. Go through some code, line-by-line.
3. Step over a method.
4. Examine some data.

If you need to break into the debugger while the driver is running, go to the target machine and hold down Control-Power for two seconds. Note that you could reboot the machine if you hold the key down too long. Try doing it like this: While holding down the Control key, press the Power key and count aloud to two like this "One thousand one, one thousand two." Then release the Power key. A prompt should appear, saying "Debugger (Programmer Key)" and giving you the option to continue or reboot.

Stop the Debugger

When you've finished debugging, do the following:

1. On the target machine, hold down Control-Power for two seconds to break into the debugger.
2. On the development machine, enter this at the GDB prompt:

   ```
   detach
   ```

   The debugging session stops.