Copyright ©1995 by NeXT Computer, Inc.   All Rights Reserved.

4

Building, Configuring, and Debugging Drivers

This chapter tells you how to integrate your Driver Kit driver with the rest of the system. It first describes building the driver using Project Builder. It tells how to set up the initial configuration files and set the configuration parameters with the Configure application. Finally, it highlights some of the debugging aids available for finding driver bugs and tracing your driver's execution. Consult the other sources mentioned for in-depth information about the tools.

Also see Chapter 9, ªBuilding, Loading, and Debugging Loadable Kernel Serversº in *NEXTSTEP Operating System Software* for details on that topic. Look at **/NextLibrary/Documentation/NextDev/Examples/DriverKit/TestDriver** for an example of building, loading, and debugging a driver.

Driver Bundles

To load your driver into the kernelÐeven if only for testingÐyou need to create a *driver bundle* for it with Project Builder. A driver bundle contains all the files needed to load and configure a driver: Its relocatable code and configuration information. A bundle may also contain help information and a configuration inspector for Configure to access configuration data. A driver bundle is also called a *config bundle* because it contains configuration information for the driver and typically has the name *Driver***.config**, where *Driver* is the driver's name.

The driver name should be of the form

    <vendor><model><type>Driver

The driver name *Adaptec1542SCSIDriver* follows this form.

Bundle Locations

Driver bundles for each system deviceÐlike the mouse, display, network card, SCSI devices, and so onÐreside in a special directory called **/NextLibrary/Devices**. The bundles for each type of device are called *Driver***.config**, where *Driver* is a type of device or a device name. In addition, every system has a bundle called **System.config** that configures the whole system.

An average system's directory **/NextLibrary/Devices** might contain the following directories, each of which is a bundle for a specific device:

ATI.config	PS2Mouse.config

Adaptec1542B.config
Beep.config
BusMouse.config
CirrusLogicGD542X.config
CompaqAudio.config
DPT2012.config
EtherExpress16.config
EtherLink3.config
Floppy.config
IDE.config
IntelGXProAudio.config
JAWS.config
MSWSoundSystem.config
PS2Keyboard.config
ParallelPort.config
ProAudioSpectrum.config
QVision.config
S3.config
SCSITape.config
SMC16.config
SerialMouse.config
SerialPorts.config
System.config
TokenExpress.config
TsengLabsET4000.config
VGA.config
Wingine.config

/NextLibrary/Devices is a link to the **/private/Devices** directory, which is a link to the driver directory for the current architecture (for example, **/private/Drivers/i386**). This link is always valid.

What's in a Bundle

Each driver bundle (including **System.config**) can contain the following files and directories:

Default.table
Instance*n*.table (created by Configure)
*x***.table**
Display.modes
*x***.modes**
CustomInspector (optional binary)
*Language***.lproj/**
    *CustomInspector***.nib** (optional)
    Localizable.strings
    Help/ (replaces **Info.rtf**)
*Driver***_reloc** (omitted for NeXT drivers that are compiled into the kernel)
Pre-Load
Post-Load

Default.table is a commented, read-only file that gives the default configuration settings for a generic device. Configure uses **Default.table** to build **Instance*n*.table** files, which contain specific configuration information for each device you have. There may be other *x***.table** files, each expressing a different possible instance of the driver.

Each **.table** file is the ASCII representation of an NXStringTable object. Drivers and nondrivers can get access to these tables by using the IOConfigTable class. In addition, Driver Kit classes automatically interpret and use some of the standard keys in these tables.

Direct drivers have one **Instance*n*.table** for each device. For example, if you have two of the same card, Configure makes two files called **Instance0.table** and **Instance1.table** in the card's bundle. Indirect drivers and the system bundle have only one file, called **Instance0.table**.

Note: Because Configure's default device inspector has no way of knowing whether a device is direct or indirect, it can create more than one **Instance*n*.table** for an indirect driver. The consequence is that the driver's **probe:** method gets invoked more than once for each direct driver it might want to attach to. To get around this, you should either write your own device inspector or ensure that your driver's **probe:** method can handle more than one probe per direct driver.

The **Display.mode** and *x***.mode** files hold display mode information. Default information is in **Display.mode**, and *x***.mode** holds the information for other instances of the driver (just as *x***.table** expresses configuration information for other driver instances).

For each language, **Localizable.strings** contains the text strings that applications display about the device. For example, it includes the name of the device as it appears in the list of devices in Configure. The **Help/** directory contains files to inform the user about the driver and help them use it.

The *Driver*_**reloc** file is the relocatable object file of the device driver. The *CustomInspector* binary is the executable file for the Inspector panel; its name is the same as the bundle name (without the **.config** suffix). *CustomInspector***.nib** is the nib file for the Inspector panel.

The bundle may contain *Pre-Load* and/or *Post-Load* programs that are run before and/or after the driver is loaded.

Configuration Tables

Files with a **.table** suffix contain strings of key/value pairs that describe a configuration. See ªConfiguration Keysº in the Appendix for information on what these tables should contain.

You can use the **Default.table** of an existing driver as a starting point for a configuration. Later, you should let the Configure application (with your custom inspector, if any) create the **Instance***n***.table** files.

Here's a sample **Instance***n***.table** for a parallel port driver:

```
"Driver Name"          = "IOParallelPort";
"Title"           = "System Parallel";
"Location"             = "System Baseboard";
"Family"               = "Parallel";
"Version"              = "1.0";
"Server Name"          = "ParallelPort";
"Path 0"               = "/dev/pp0";
"Post-Load"            = "InstallPPDev";
"Memory Maps"          = "";
"Pre-Load"             = "RemovePPDev";
"DMA Channels"         = "";
"Minor Device Number"  = "0";
"Valid IRQ Levels"          = "7";
"I/O Ports"            = "0x378-0x37f";
"Instance"             = "0";
"Port Count"           = "1";
"IRQ Levels"           = "7";
```

Warning: C-style comment delimiters (that is, /* */) aren't recognized in configuration tables, such as **Default.table** or **Instance0.table**. Anything inside the delimiters will be parsed along with the rest of the file. This means that, for example, if you are testing a driver under development, you can't remove a key-value pair by simply commenting it out.

Other Configuration Tables

A bundle may also contain other configuration tables of the form *x***.table**, where *x* is a prefix such as ªPCIº. Each of these is a table like **default.table** but expresses a possible instance of the driver with a slightly different ªpersonalityº than **default.table**. For example, **PCI.table** might be identical to **Default.table** except that it contains a line specifying a PCI-compliant driver:

```
"Bus Type" = "PCI";
```

By convention, **Default.table** specifies an ISA or VL-bus compliant driverÐthe simplest case. The prefix *x* in *x***.table** usually designates the bus type.

These configuration table files should contain all information appropriate for the bus type. PCI-compliant drivers, for instance, contain a line specifying the auto detect IDs, such as this:

```
"Auto Detect IDs" = "0x71789004 0x0e111234";
```

Custom Device Inspector Files

For initial testing, you probably don't need a custom inspector. Instead, you can put the appropriate values directly into your test **Default.table** or **Instance*n*.table** files.

If you create a custom inspector, you should put the executable file and nib file in the places described in ªWhat's in a Bundle,º earlier in this chapter. Project Builder does this for you automatically. See ªWriting a Custom Inspectorº later in this chapter for information on creating custom inspectors.

Note: Project Builder creates an Inspector Panel executable file in the bundle and gives it the same name as the bundle (without the **.config** suffix). This executable loads the default inspector.

Localizable Strings File

This file should contain any strings you add to your Configure inspector's user interface, plus the following strings:

```
"Driver" = "UltimateTech XYZ-12";
     "Long Name" = "Ultimate Technologies XYZ-12 Transmogrifier";
```

where *Driver* is the name of the bundle (minus the **.config** suffix). Configure uses the string associated with the *Driver* key (ªUltimateTech XYZ-12º) whenever space is tight. When Configure has more space to display the driver's name, it uses the string associated with the ªLong Nameº key.

Display Mode Tables

If your driver is a display driver that supports multiple display modes, you need to specify which modes the user can choose. This information is supplied in the **Display.modes** file. Here's a sample file:

```
"Height: 600 Width: 800 Refresh: 60Hz ColorSpace: RGB:555/16";
"Height: 600 Width: 800 Refresh: 72Hz ColorSpace: RGB:555/16";
"Height: 768 Width:1024 Refresh: 60Hz ColorSpace: RGB:256/8";
"Height: 768 Width:1024 Refresh: 66Hz ColorSpace: RGB:256/8";
"Height: 768 Width:1024 Refresh: 72Hz ColorSpace: RGB:256/8";
"Height: 768 Width:1024 Refresh: 76Hz ColorSpace: RGB:256/8";
"Height: 768 Width:1024 Refresh: 60Hz ColorSpace: BW:8";
"Height: 768 Width:1024 Refresh: 66Hz ColorSpace: BW:8";
"Height: 768 Width:1024 Refresh: 72Hz ColorSpace: BW:8";
"Height: 768 Width:1024 Refresh: 76Hz ColorSpace: BW:8";
"Height: 768 Width:1024 Refresh: 60Hz ColorSpace: RGB:555/16";
"Height: 768 Width:1024 Refresh: 72Hz ColorSpace: RGB:555/16";
"Height:1024 Width:1280 Refresh: 68Hz ColorSpace: RGB:256/8";
"Height:1024 Width:1280 Refresh: 68Hz ColorSpace: BW:8";
"Height: 400 Width: 640 Refresh: 60Hz ColorSpace: RGB:888/32";
"Height: 400 Width: 640 Refresh: 70Hz ColorSpace: RGB:888/32";
"Height: 480 Width: 640 Refresh: 60Hz ColorSpace: RGB:888/32";
```

If your driver has more than one ªpersonality,º specify alternate display information in ***x*.modes** files where *x* is the appropriate prefix such as ªPCIº.

See the specification for the IODisplayInspector, IOFrameBufferDisplay, and IOSVGADisplay classes for more information on display modes.

Help Directory

This directory contains the help files supported by the NeXT help facility. You add this directory to your project with Project Builder's Add Help Directory command. For more information on adding help to your driver, see ªAttaching Help to Objectsº in Chapter 3, ªThe Interface Builder Applicationº of *NEXTSTEP Development Tools and Techniques*.

The **Help** directory replaces the **Info.rtf** file, formerly used to provide information about the driver.

Driver Relocatable Code

This file contains the driver's relocatable code. An example of building a driver relocatable object file is located in **/NextLibrary/Documentation/NextDev/Examples/DriverKit/TestDriver**.

Pre- and Post-Load Programs

Your driver may require some action to be taken before and/or after it is loaded. For instance, you may want to run a program after the driver is loaded to look up its major device number and create a device node for the driver. Use the ªPre-Loadº configuration key to specify a program that will run prior to your driver being loaded; use the ªPost-Loadº key to specify a program that runs after the driver is loaded.

The System Configuration Bundle

The **System.config** bundle is special in several ways. Its **Instance0.table** has default configuration information for the system as a whole. For example, it specifies which device drivers to load at boot time (ªBoot Driversº) and which to load later (ªActive Driversº). Here's a sample **Default._table** from a **System.config** bundle:

```
"Version" = "2.0";
"Boot Drivers" = "PS2Keyboard PS2Mouse BusMouse Adaptec1542B DPT2012 IDE
Floppy VGA";
"Active Drivers" = "SerialPorts SerialMouse ParallelPort";
"Kernel" = "mach_kernel";
"Kernel Flags" = "";
"Boot Graphics" = "No";
```

For writers of Driver Kit drivers, ªActive Driversº and ªBoot Driversº are the most important keywords. They specify which drivers are automatically loaded into the system the next time it's started. When someone uses Configure to add a device that has a loadable driver, the driver is added to one of these two lists. See the ªBoot Driversº and ªActive Driversº keys in the ªConfiguration Keysº section of the Appendix to see how to specify which list a driver should be in. This section also lists the other keywords for the system configuration table.

Note: Changes to system configuration information don't take effect until the system is restarted. However, you can load a driver without rebooting by using the **d** option of **driverLoader** (documented in ªLoading a Driver with driverLoaderº later in this chapter).

Creating a Driver Bundle

Create a project for your driver with Project Builder, and give the project the name you want your driver to have. Copy your driver files into the project by dragging them into the appropriate suitcase (header files to the Header suitcase and so on) or by using the Add command in the Files menu. Switch to the Builder view in the project window and select ªbundleº as the Target. Click the Build button. Project Builder builds the driver and puts it in a driver bundle called *Driver***.config** where *Driver* is the name you chose for the driver. Now you can configure and load the driver.

See *NEXTSTEP Development Tools and Techniques* for more information about using Project Builder. The example in **/NextLibrary/Documentation/NextDev/Examples/DriverKit/TestDriver** shows building a bundle with Project Builder.

Configuring Drivers

After you have built your driver, you need to configure it with the Configure application.

Configure Application

You can configure devices and add drivers with the Configure application. When you select a device, Configure loads the device's inspector, which provides a user interface for manipulating the device configuration (choosing its DMA channels, for example). If you don't supply a device inspector for your driver, Configure uses a default device inspector. See the IODeviceInspector class and IOConfiguration protocol specifications for more information on device inspectors.

The Configure application reads the key/value pairs from a driver bundle's **Default.table** and displays them in a Configuration Inspector Panel. The user interface allows the user to change the displayed parameters and warns of possible value conflicts. When the user finishes modifying the configuration, Configure writes the updated configuration to the indicated **Instance*n*.table** and configures the driver based on the information in the configuration and kernel tables.

When the system starts up, the kernel uses an IOConfigTable object to parse the configuration information in the **Instance*n*.table**. From this information, the kernel instantiates an IODeviceDescription object, which encapsulates information about the driver. The kernel passes the IODeviceDescription object as the parameter to the **probe:** method, which instantiates the driver object based on this information.

There's a list of standard key/value configuration pairs in the ªConfiguration Keysº section in the Appendix.

How Configuring Kernel-Level Driver Kit Drivers Differs from Configuring Other Loadable Kernel Servers

The configuration of Driver Kit kernel-level drivers differs from that of other Loadable Kernel Servers (LKSs) in the following ways:

- Each Driver Kit driver has its own configuration directory under **/NextLibrary/Devices**. Other LKSs have no standard way of getting configuration information.
- With the Configure application, users can add Driver Kit drivers to the system, as well as specify configuration information for each driver. Other LKSs are generally added to the system by adding a line to **/etc/kern_loader.conf**.
- Driver Kit drivers are allocated and loaded with the **driverLoader** command, which uses the information in the driver's configuration directory. You can load an LKS with the kernel-server utility, **kl_util**, but it doesn't cause the driver to be probed.
- Driver Kit drivers can't currently be unloaded, unlike other LKSs. For example, if you want to change a driver that's already running, you must restart the system to be able to load the new driver.

Writing a Custom Inspector

The Configure application uses inspectors to configure a driver. With the default inspector in Configure, you can configure values that belong to the standard set of keys with no further implementation effort. If you've added custom parameters, however, you need to implement a custom inspector to view and modify them.

You have two choices in implementing a custom inspector:

· Add an accessory view to the inspector, with an 80-pixel height limit.

· Replace the standard inspector completely. You're still limited to a 640×480 view. However, you can use a button to display a panel if you run out of space.

You implement an inspector by creating a subclass of IODeviceInspector. For example, you can create a subclass of IODisplayInspector (a subclass of IODeviceInspector) to implement a display inspector. For an example, study the inspector in **/NextLibrary/Documentation/NextDev/Examples/DriverKit/DriverInspector**.

Other classes relevant to creating an inspector include IOAddressRanger, IODeviceDescription, IODeviceMaster, and IOEISADeviceDescription. Some of these classes adopt the IOConfigurationInspector protocol.

Creating an Inspector

Override the following methods in the IODeviceInspector class and the IOConfigurationInspector protocol:

· **init**. Find and load the nib file that contains the accessory view using the bundle for your inspector. Initialize the user interface and find your driver.

· **inspectionView**. Override this if you're replacing the standard inspector.

· **setTable:**. Invoke the superclass's implementation:

```
[super setTable:]
```

  Invoke **setAccessoryView:** to specify and initialize the accessory View. Initialize the user interface settings from the table being inspected.

· **resourcesChanged:**. Update the user interface in response to resources being chosen or dropped in the inspector.

Modifying Custom Parameters

Implement a set of target/action methods to change the custom parameters. The user interface elements of the inspector invokes these methods. Convert the new parameter state to an appropriate string value for display, and insert it into the inspected table with **insertKey:value:**. The key must be a unique string, and you can use the **NXUniqueString()** function to generate a unique key based on the string argument. The value should be a copyÐuse **NXCopyStringBuffer()** to copy it:

```
[table insertKey:key value:NXCopyStringBuffer(value)];
```

Changing Driver Parameters with

IODeviceMaster

Besides Configure, another way to change parameters associated with a driver is through the IODeviceMaster class, which provides access to a driver instance. First, find your driver using the **lookUpByDeviceName:objectNumber:deviceKind:** method. Then manipulate parameters associated with that instance with these methods:

- **getCharValues:forParameter:objectNumber:count:**
- **setCharValues:forParameter:objectNumber:count:**
- **getIntValues:forParameter:objectNumber:count:**
- **setIntValues:forParameter:objectNumber:count:**

Active driver values should be displayed in the user interfaceÐeven if they differ from the current configuration table values. If you want the values you change to persist beyond the time the system is powered off or restarted, you must write them to the configuration table.

Loading a Driver with driverLoader

You can load your driver into an already running system. The **driverLoader** command loads or configures a driver after startup time. You initiate the command as follows (as superuser):

/usr/etc/driverLoader *option* [**v**] [*instance*]

Specifying **v** results in more verbose output from **driverLoader**. The *instance* argument can be used only with the **d** option, as described below.

The *option* is one of the following:

a	Configure all devices. This option is used when **driverLoader** is run during system boot (by **/etc/rc**).
i	Interactive mode. With this option, you can look at all active and boot drivers in the system configuration. Note that if you add a driver to the system, the driver isn't recognized as ªactiveº until you reboot.
d=*deviceName*	Configure one device interactively. This is how you load drivers that aren't specified in the system configuration. This is usually used for testing purposes. You can specify *instance* to use a specific **Instance***n***.table** file. For example, if you specify *instance* as 1, the driver is probed using the information in its **Instance1.table** file.

Here's an example of using the **d** option:

```
# /usr/etc/driverLoader d=myDriver
```

Here's an example of using the **d** option and specifying *instance*:

```
# /usr/etc/driverLoader d=fooDriver 1
```

For another example of using **driverLoader**, see **/NextLibrary/Documentation/NextDev/Examples/DriverKit**.

Recovering from a Bad Configuration

If you can't restart your system because of a bad configuration or because of bugs in your driver, try restarting with a default configuration. To do this, type the following at the **boot:** prompt when the system starts:

```
boot: config=Default
```

This causes the boot program to use **Default.table** in **System.config** as the system configuration, which usually works. Once you've started up, log in as **me** or **root** and use Configure to fix the rest of the configuration.

If you still can't start the system, try starting in single-user mode and editing the bundles by hand. This is risky since the configuring process has many ªrules of thumb,º and you might not know all the effects of a change. To restart in single-user mode, type the following at the **boot:** prompt after you restart:

```
boot: mach_kernel -s config=Default
```

You can then use a single-user mode editor (such as **vi** or **emacs**) to edit the configuration bundles.

Debugging a Driver

You have two choices for creating debugging messages: the **IOLog()** function and the Driver Debugging Module (DDM). Most drivers just use **IOLog()** until a need arises for the more powerful and complex DDM functions.

Another debugging tool, **gdb**, is described in *NEXTSTEP Development Tools and Techniques.* You can run the driver with **gdb** from Project BuilderÐthe example located in **/NextLibrary/Documentation/NextDev/Examples/DriverKit/TestDriver** shows how to do this. *NEXTSTEP Development Tools and Techniques* also describes Project Builder.

Using the IOLog Function

Using **IOLog()** is similar to using **printf()** to print error or debug messages. You can output strings and parameters, just as for **printf()**. One difference is that output is placed in the **/usr/adm/messages** file instead of the console window. Place a call to **IOLog()** anywhere in your driver where you want to get information about the driver stateÐor to indicate that the driver reached that point during execution.

IOLog() is useful both for status messages and as a basic debugging tool. Although **IOLog()** is useful for debugging, it can affect the timing of the driver. When timing is important, you should use DDM instead.

See ªFunctionsº in Chapter 5, ªDriver Kit Referenceº, for more information about **IOLog()**.

Using the Driver Debugging Module (DDM)

The Driver Debugging Module (DDM) provides support for viewing debugging information without disturbing the timing of the kernel. By using the DDMViewer application (in **/NextDeveloper/Demos**), you can specify which information should be stored in the event buffer and display debugging information from this buffer.

The core of DDM is a circular event buffer that stores the debugging information sent to it by drivers. Each entry in the buffer is timestamped (to the microsecond) and consists of a **printf**-style format string and up to five arguments associated with the format string. A call to the function that timestamps and stores one entry takes about 10 microseconds.

Gathering DDM Events

The function **IOAddDDMEntry()** adds an event to the DDM buffer. An event consists of a character string and several integer values. The **IODEBUG()** macro is provided to call **IOAddDDMEntry()**: A driver typically doesn't call **IOAddDDMEntry()** directly. Instead, the driver should define its own macros using the **IODEBUG()** macro, as in this example:

```
#define ddm_exp(x, a, b, c, d, e)                          \
   IODEBUG(A7770_DDM_INDEX, DDM_EXPORTED, x, a, b, c, d, e)
#define ddm_him(x, a, b, c, d, e)                          \
   IODEBUG(A7770_DDM_INDEX, DDM_HIM, x, a, b, c, d, e)
```

These macros can then be called like this:

```
ddm_him("abort_channel chan %d\n", channel, 2,3,4,5);

ddm_him("scb_int_preempt: scb 0x%x index %d haStat %s\n",
   scb_ptr, scb_index,
   IOFindNameForValue(compstat, scbHaStatValues),
   4,5);
```

A word of mask bits controls the collection of DDM entries. All calls to **IODEBUG()** don't add data to DDM's circular bufferÐonly those events whose mask bits are enabled are added. The mask bits are enabled and disabled by a user-level tool like DDMViewer. A driver isn't (and shouldn't be) concerned about which mask bits are enabled. Typically you turn on one or two bits of the mask word to study the trace information for a particular module.

See the SCSI example driver in **/NextDeveloper/Examples/DriverKit/Adaptec1542B**, which illustrates all aspects of using DDM.

Viewing DDM Events with DDMViewer

You can examine DDM traces at the user-level with the DDMViewer application, which is located in **/NextDeveloper/Demos**. You can also specify DDM mask bits with this application. DDMViewer can be run on any computer running NEXTSTEP, not just the machine being tested.

The DDMViewer window contains the following controls:

· **Device Name field**. Enter the name of the target to which you want to attach. The name is determined by the driver.

· **Host Name field**. Enter the name of the host on which the target is running. Leave it empty if you are debugging a driver or kernel on the current machine.

· **List button**. Click this button to start and stop the display of DDM entries. Entries are displayed starting from the last event in time and scrolling backward.

· **Set Mask button**. Click this button to send the mask defined in the Mask window (see below) to the target.

· **Disable button**. Click this button to freeze the state of the DDM buffer at the target. Click again to reenable.

· **Clear Window button**. Click this button to clear the display area.

· **Clear Buffer button**. Click this button to clear the target's circular DDM buffer.

You can specify the value of the DDM mask bits by name if you open a **.ddm** file that specifies the names of the mask bits. You create **.ddm** files with an editor such as Edit. Here's an example of a **.ddm** file:

```
#
```

```
#  DDMViewer data file for kernel devices.
#
Index : 0 : "Kernel Devices"
#
#  Common fields.
#
0x0001 : "Device Object"
0x0002 : "Disk Object"
0x0004 : "Net"
0x0020 : "DMA"
#
#  SCSI.
#
0x0100 : "SCSI Control"
0x0400 : "SCSI Disk"
```

Comments start with ª#º. The line that starts with ªIndexº defines which DDM Mask word is being defined (there are currently four mask words). The Index line also defines the name of the window associated with this set of mask bits. All other lines define one bit in the mask word, specifying the value of the bit and an ASCII name equivalent. The SCSI example driver in **/NextDeveloper/Examples/DriverKit/Adaptec1542B** has a sample **.ddm** file.