ChorusOS 4.0 Device Driver Framework Guide

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900
U.S.A.

Part Number 806-0616–10
December 1999

Adobe PostScript

Please Recycle

Contents

Preface

The *ChorusOS 4.0 Device Driver Framework Guide* explains the set of tools provided within the ChorusOS operating system to build device and bus drivers, and how to use them. It provides an overview of the device and bus driver architecture within the ChorusOS operating system environment, and explains how the Driver Framework can be used to build highly portable, platform-independent drivers, as well as highly tunable, processor-family specific drivers using the same software paradigm. It also contains a summary of the APIs that make up the Driver Framework, although details of the API calls are discussed in the man pages.

Who Should Use This Guide

This book is designed to be used by developers already familiar with building bus and device drivers. For this reason, there is no general description of the tasks involved in building bus or device drivers. It is recommended that the reader consult books and/or websites dealing specifically with the architecture of the different device, bus, and processor architectures for this kind of information.

Before You Read This Guide

Before starting to build drivers using the ChorusOS operating system, read the *ChorusOS 4.0 Introduction*. The introduction provides an overview of the features and components of the ChorusOS operating system and explains how to create an application that runs on the ChorusOS operating system.

How This Guide is Organized

This book is organized into the following sections.

Chapter 1 provides an introduction to the toolset, gives an overview of hardware representation in the ChorusOS operating system, and outlines some of the benefits of using the Driver Framework.

Chapter 2 outlines tasks and services common to device and bus driver production within the Driver Framework Device Kernel Interface API.

Chapter 3 provides a step-by-step overview of writing device drivers in the Driver Framework, using a working driver as an example.

Chapter 4 provides a step-by-step overview of writing bus drivers in the Driver Framework, using a working driver as an example.

Appendix A provides pointers to detailed information.

Related Reading

The *ChorusOS 4.0 Introduction* introduces the features and components of the ChorusOS operating system. It explains how to use ChorusOS and how to create an application that runs on the ChorusOS operating system.

The *ChorusOS Release Notes* contain information about new features and restrictions in this release of the product.

The following books describe how to use ChorusOS

- *ChorusOS 4.0 File Systems User's Guide* explains how to use the file systems provided with the ChorusOS operating system. It includes information about using the NFS server.

  ChorusOS 4.0 Installation Guide for Solaris Hosts explains how to download and install ChorusOS on a Solaris host.

  ChorusOS 4.0 Installation Guide for Windows NT Hosts explains how to download and install ChorusOS on a Windows NT host.
- *ChorusOS 4.0 Network Administration Guide* explains how to use various network protocols with the ChorusOS operating system, including the point-to-point protocol (PPP) and the serial line internet protocol (SLIP).
- *ChorusOS 4.0 Hot Restart Programmer's Guide* describes the support for hot restart provided in the ChorusOS operating system and explains how to use it.

Ordering Sun Documents

Fatbrain.com, an Internet professional bookstore, stocks select product documentation from Sun Microsystems, Inc.

For a list of documents and how to order them, visit the Sun Documentation Center on Fatbrain.com at `http://www1.fatbrain.com/documentation/sun`.

Accessing Sun Documentation Online

The docs.sun.com^{SM} Web site enables you to access Sun technical documentation online. You can browse the docs.sun.com archive or search for a specific book title or subject. The URL is `http://docs.sun.com`.

What Typographic Conventions Mean

The following table describes the typographic changes used in this book.

TABLE P–1 Typographic Conventions

Typeface or Symbol	**Meaning**	**Example**
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls –a` to list all files. `machine_name% you have mail.`

`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`**
		`Password:`

TABLE P–1 Typographic Conventions *(continued)*

Typeface or Symbol	Meaning	Example
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type **`rm`** *filename*.
AaBbCc123	Book titles, new words, or terms, or words to be emphasized.	Read Chapter 6 in *User's Guide*. These are called *class* options. You must be *root* to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

CHAPTER **1**

Introduction to the ChorusOS Driver Framework

Introduction

The ChorusOS system provides a driver framework, allowing the third-party developer to develop device drivers on top of a binary system distribution. The Driver Framework provides a well-defined, structured and easy-to-use environment to develop both new drivers and client applications for existing drivers.

Host bus drivers written with the Driver Framework are processor-family specific, meaning that they are portable within that processor family (UltraSPARC™, PowerPC, Intel ix86 processor families). Drivers that occupy a higher place in the hierarchical bus structure (sub-bus drivers and device drivers) are usually portable between processor families.

Device Driver implementation is based on services (provided by a set of APIs, such as PCI or ISA) which allow the developer to choose the optimizability and portability of the driver they create. This allows the driver to be written to the parent bus class, and not the underlying platform. Drivers written within the Driver Framework may also take advantage of processor-specific services, allowing maximum optimization for a particular processor family.

Benefits of Using the Driver Framework

Using the Driver Framework to build bus and device drivers in the ChorusOS operating system provides the following benefits to the user:

- A structured framework, easing the task of building drivers
- Hierarchical structure of drivers in Driver Framework mirrors hardware structure
- Ensures compliance and functionality within the ChorusOS operating system
- Enables the user to develop multi-bus device drivers, which may run on all buses supporting the Common Bus Driver Interface
- Drivers built with the Driver Framework are homogeneous across various system profiles (flat memory, protected memory, virtual memory)
- Allows dynamic configuration (and re-configuration) needed for plug-and-play, hot-plug and hot-swap support
- Supports the binary driver model
- APIs are version resilient
- Is adaptive (in terms of the memory footprint and complexity) to the various system profiles and customer requirements
- Supports the dynamic loading and unloading of driver components
- Meets real-time requirements, by providing non-blocking (asynchronous) run-time APIs

Framework Architecture Overview

In the ChorusOS operating system, a driver entity is a software abstraction of the physical bus or device. Creating a device driver using the Driver Framework allows the device or bus to be represented to and managed in the ChorusOS operating system. The hierarchical structure of the driver software within the ChorusOS operating system mirrors the structure of the physical device / bus.

Each device or bus is represented by its own driver. A driver's component code is linked separately from the microkernel as a supervisor actor, with the device-specific code strongly localized in the corresponding device driver.

Note that a supervisor actor containing a driver code should be considered as a *container only* rather than as a real supervisor actor with its own execution personality. Driver code runs either in the interrupt execution environment (typically up-calls) or in the driver client execution environment (typically down-calls). In other words, the driver component code logically belongs to the current driver client (microkernel module or supervisor actor).

Note - The driver is *always* considered a trusted system component.

This means that the Driver Framework defines a structure and principle, but since the driver is a trusted system component, parameter and logic checking are *not* performed on most drivers in release mode. Even if the task of creating drivers with the Driver Framework seems relatively simple, care should be taken to ensure that drivers are written in conformance with the framework. Some checking can be added in debug mode, but this can not replace writing the driver in compliance with the Driver Framework.

Driver components are organized, through a services-provider/user relationship, into hierarchical layers which mirror the hardware buses/devices connections.

Interactions between these drivers are implemented via simple indirect function calls (down-calls and up-calls).

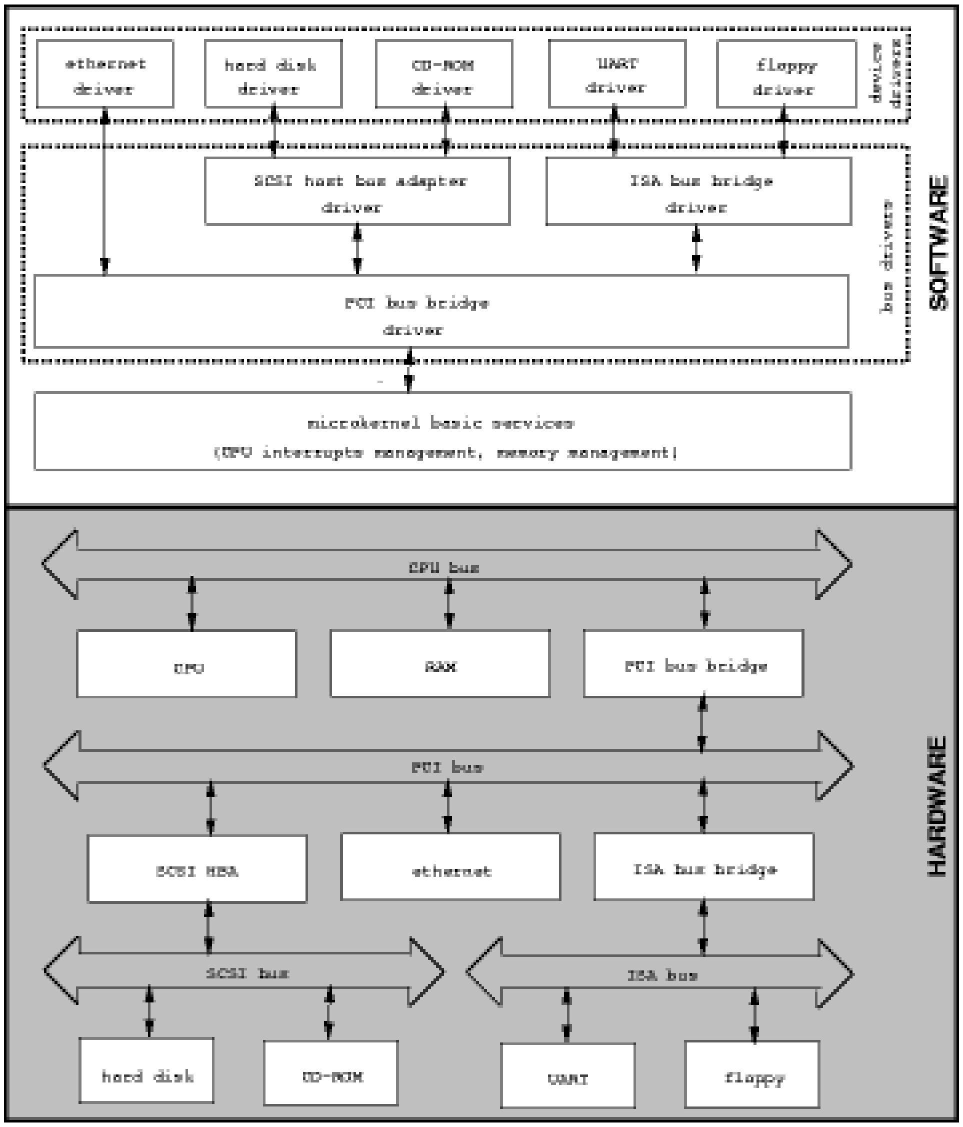

Figure 1–1 Bus/Device Hierarchy, in Hardware and Software

To sum up, the ChorusOS operating system Driver Framework can be considered in two ways:

- A hierarchical set of APIs which defines the services provided for and used by each bus or device driver at each layer of the architecture. This approach ensures portability and functionality across various platforms and continued validity of drivers across subsequent system releases.

- A set of mechanisms implemented by the ChorusOS microkernel, ensuring compliance and synchronicity with the ChorusOS operating system architecture and methods.

Figure 1–2 shows the objects involved in the ChorusOS Driver Framework:

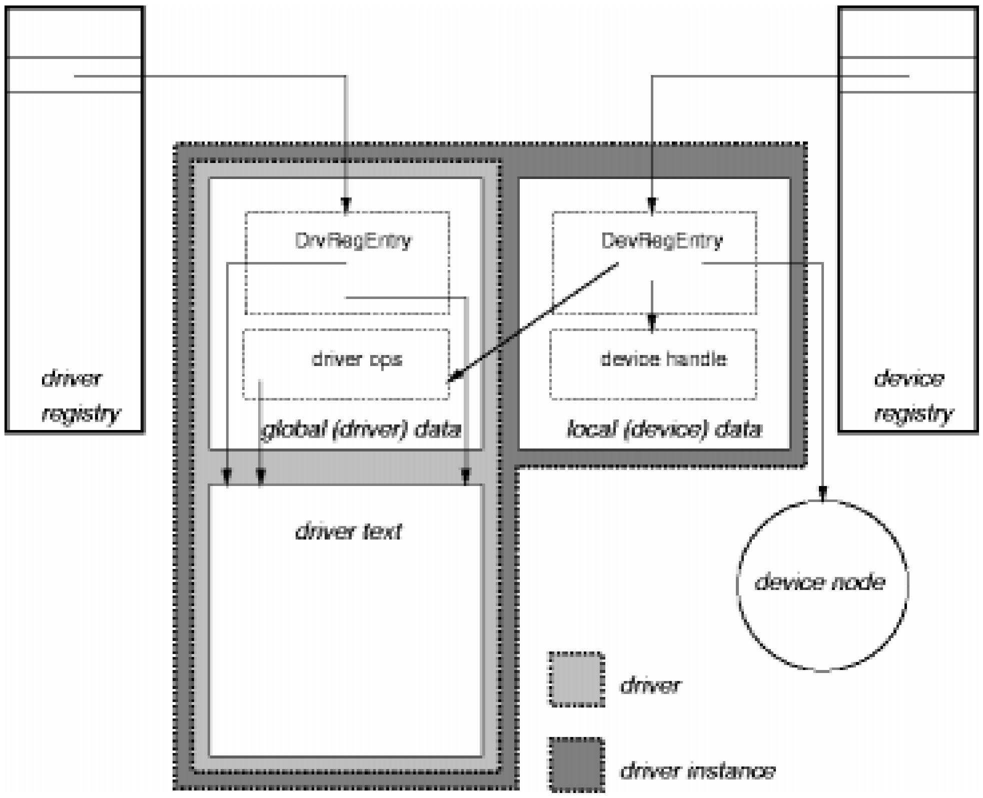

Figure 1–2 Driver Framework Objects

Driver Framework APIs

One of the key attributes allowing portability and modularity of devices constructed using the Driver Framework is the hierarchical structure of the APIs, which can also be seen as the layered interface. Within this model, all calls to the microkernel are performed through the Driver Kernel Interface (DKI) API, while all calls between drivers are handled through the Device Driver Interface (DDI) API.

The figure below represents the layered (hierarchical) structure of the Driver Framework APIs.

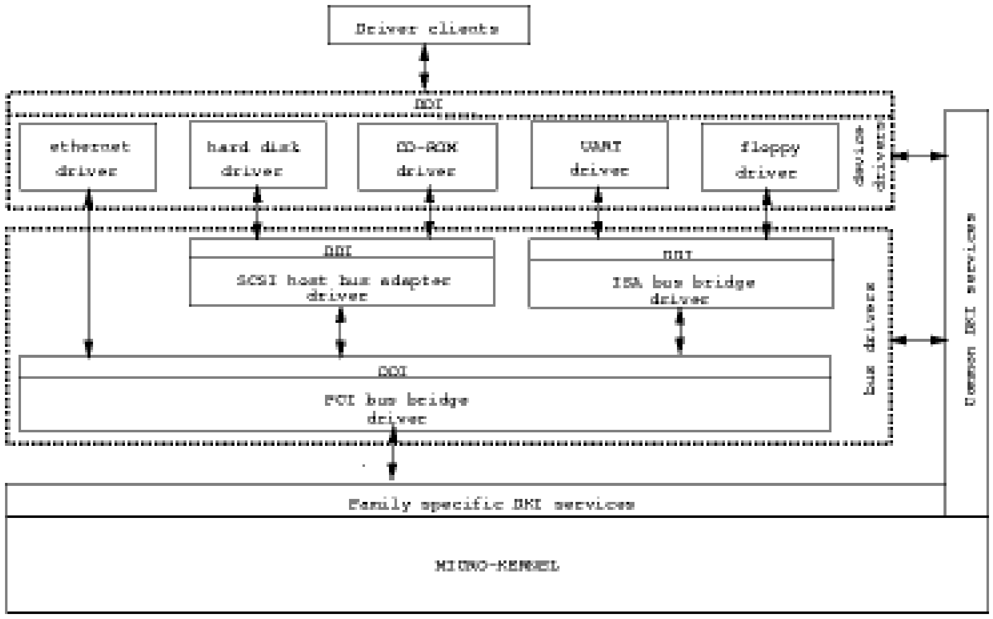

Figure 1–3 Device Interface Layering

Driver/Kernel Interface (DKI)

The DKI interface defines all services provided by the microkernel to driver components. Following the layered interface model, all services implemented by the DKI are called by the drivers, and take place in the microkernel.

Common DKI services are services common to all platforms and processors, usable by all drivers, no matter what layer in the hierarchical model they inhabit. These services are globally designed by the `DKI` class name.

Common DKI services cover:

- Synchronization through the DKI thread
- Device tree
- Driver registry
- Device registry
- General purpose memory allocation
- Timeout

- Precise busy wait
- Special-purpose physical memory allocation
- System event management
- Global interrupts masking
- Specific I/O services

Processor family specific DKI services are defined and available *only* for a given processor family and should be used only by the lowest-level drivers. Lowest-level drivers are those for buses and devices which are directly connected to the processor local bus. Note that these drivers typically use only the DKI services (no available layer of DDI). These services are globally designed by the `FDKI` class name (for Family DKI).

Processor family specific DKI (FDKI) services cover:

- Processor interrupts management
- Processor caches management
- Processor specific I/O services
- Physical to virtual memory mapping

All DKI services are implemented as part of the embedded system library (`libebd.s.a`). Most of them are implemented as microkernel system calls. Note that the `dki(9)` man page gives an entry point to a detailed description of all DKI APIs.

Device Drivers Interface (DDI)

The DDI defines several layers of interface between different layers of device drivers in the driver's hierarchy. Typically an API is defined for each class of bus or device, as a part of the DDI.

Note that a driver's client application may itself be a driver component (as a device driver is a client of the bus driver API). In this way, it can be seen that all DDI services are implemented by a driver component, and are in turn called by upper-layer drivers (or directly by the driver's client applications).

As illustrated earlier, in Figure 1–3, the DDI set of APIs is further divided along hierarchical lines into two principle interface layers — Bus Driver Interfaces and Device Driver Interfaces.

Bus Driver Interface APIs

This layer of interfaces is implemented by the lowest level layer of drivers, using DKI services. This set of drivers can itself be composed of multiple sub-layers to reflect the bus hierarchy of a given platform.

Typically, only the primary (host) bus driver is built solely using DKI services. Subsequent drivers, those occupying a “downstream” position in the hierarchy, interface with the primary (host) bus. As all different I/O buses share a subset of features, and then have their particular specificities, the bus driver interfaces layer offers a subset of services called "Common bus driver interface" (CBDI) , which is independent of the bus type, offering a set of services common for all bus classes.

In addition to the CBDI, there is of course a collection of bus specific interfaces (such as PCI, VME, ISA) to implement bus-specific driver services.

Device Driver Interface APIs

This layer of interfaces is implemented by the device drivers, and is built upon the lower layer of services (bus driver interfaces). This set of device drivers provides different interfaces for each different class/type of device. Typically, there are different interfaces for timer devices, UART devices, Ethernet devices and so on.

Each of these APIs may be used by the driver’s client application to manage the associated devices. Note that the `ddi(9)` manpage gives an entry point to a detailed description of all DDI APIs.

Driver Framework Mechanisms and Principles

As mentioned above, the ChorusOS operating system microkernel implements mechanisms to enforce a well-defined behavior regarding driver component initialization, dynamic loading/unloading, and bus events management. An overview of these mechanisms follows, while a more detailed examination is provided in “Chapter 2, Driver Kernel Interface Overview”,”Chapter 3, Writing Device Drivers” and “Chapter 4, Writing Bus Drivers.”

Driver Registration

The driver framework defines three device and driver registration entities which are managed through the DKI interface:

- the Driver Registry is used to register and manage loaded driver components

- the Device Tree is used to represent the hardware buses/devices hierarchy and defines the properties of each hardware chip
- the Device Registry is used to register/retrieve driver component's instances servicing a given hardware device (bound to and initialized to manage a given node in the device tree)

Driver Initialization

The microkernel initialization goes through the following steps:

1. device independent microkernel initialization
2. built-in device drivers initialization
3. device dependent microkernel initialization

At the first step, the microkernel performs initialization of device-independent modules like executive, memory management and so on.

At the second step, the microkernel installs and launches the built-in device driver actor(s) (drivers which are embedded in the ChorusOS operating system archive). Note that each driver actor's `main()` function is invoked sequentially by the microkernel initialization thread. The driver's `main()` function should perform a self-registration of the driver component within the system, by using the DKI interface.

When registering, the driver exports its properties to the system:

- information about the component (name, version)
- the required parent bus API class and version
- driver's entry points, which have a well-defined semantic

Once the driver component is self-registered, future management of the driver is controlled by its parent bus/nexus driver, using the properties registered.

The four possible entry points that a driver component may register are:

- a driver's probe function (`drv_probe`) to detect device(s) residing on the bus and to create device tree node(s) corresponding to these types of device(s)
- a driver's bind function (`drv_bind`) to bind a driver to a device tree node
- a driver's initialize function (`drv_init`) to initialize the hardware device, and to create a running instance of the driver component
- a driver's unload function (`drv_unload`) which is invoked by the driver registry module when an application has to unload the driver component from the system

Finally, once built-in driver components have been started, the microkernel performs initialization of device dependent modules (like the "TICK" module which relies on a TIMER class device).

Note - Interrupts are disabled at CPU level during the first, second and third initialization steps. Once the built-in drivers are initialized, interrupts are enabled at CPU level.

Once all of the driver's `main()` functions are invoked, the microkernel initiates the device initialization process. This can be seen as the microkernel implementing a local bus driver (bound to the device tree root node) for a DKI/FDKI bus class.

The initialization process starts from driver components servicing bus or device controllers directly connected to the CPU local bus; the driver registry is searched to find out the appropriate drivers and to call their registered entry points. Typically, the `probe` registered function is called for all driver components requiring a DKI/FDKI parent bus class. After probing, the `bind` function is called for all driver components requiring a DKI/FDKI parent bus class. Finally, after binding, the `initialize`registered function is called for all driver components requiring a DKI/FDKI parent bus class, that are bound to a child of the device tree root node (nodes representing a bus or a device controller directly connected to the CPU local bus).

Note - The `drv_probe`, `drv_bind` and `drv_init` routines are all optional

The `drv_probe` routine detects device(s) residing on the bus and creates corresponding device nodes in the device tree. The `drv_bind` routine allows drivers to perform a driver-to-device binding. The driver examines the properties attached to the device node in order to determine the type of device and to check whether the device may be serviced by the driver. If the check is positive, the driver attaches a driver property to the device node. The name of the driver node is "driver" and it has a string type value, specifying its name. The initialization process is propagated by the `drv_init` function of the bus/nexus drivers started by the microkernel.

In addition, when a driver instance is activated by a parent bus/nexus driver (through its registered `drv_init()` function), it establishes a connection to its parent bus driver (typically through an `open` service of the bus API) specifying a call-back event handler and a load handler. The parent bus/nexus driver uses the call-back event handler mechanism to propagate the bus events to the connected child driver instances. These events are typically bus-class specific, but are usually used to shut down child driver instances. The load handler is used (together with the unload entry point) to manage dynamic loading/unloading of the driver components.

Driver Framework Components

Source Files

Typically, a driver component is a ChorusOS operating system supervisor actor written in 'C' programming language. This type of component (named `devx` for the example) is usually composed of the following files:

- A header file (named `devxProp.h`) defining properties which are specific to this `devx` driver component. This file should be visible from the component(s) responsible for building the associated device tree node (to create the node's properties). This may be the boot program or any other "probe only" driver component.

- An implementation file (`devx.c`) which contains the C code for the driver component. Note that for very big driver components, there may be multiple `.c` files. Note also that the hardware related definitions and constants are sometimes extracted from the `.c` file and put in a header file (`devx.h`) where use is restricted to the implementation file.

- An Imakefile that is used to generate a Makefile (through the imake tool) in order to compile and link the driver component. Refer to the make and imake sections in *"ChorusOS 4.0 Introduction"*, and the imake header file (see the TOOLS directory: *installation_directory* / *platformtype-bin* / dtool) for details about the `imake` macros that are available to build driver components.

Below is a typical example of an Imakefile, which exports a `ravenProp.h` file, compiles a `raven.c` implementation file and then builds a `D_raven.r` driver actor which is embedded in the archive.

```
CSRCS = raven.c

OBJS = $(CSRCS:.c=.o)

BuiltinDriver(D_raven.r, $(OBJS), $(DRV_LIBS))

DistProgram(D_raven.r, $(DRV_DIST_BIN)$(REL_DIR))

Depend($(CSRCS))

FILES = ravenProp.h

DistFile($(FILES),)$(REL_DIR),$(DRV_DIST_INC)$(REL_DIR))
```

Organization (trees)

All files related to driver components are organized in 4 file trees:

- The 'dki' tree, populated by the ChorusOS operating system delivery and exporting the DKI set of APIs (header files onl).
- The 'ddi' tree, populated by the ChorusOS operating system delivery and exporting the DDI set of APIs (header files only).
- The 'drv' tree, populated by generic driver components and exporting the device-specific properties header files.
  Generic driver components are drivers which do not use the family specific DKI APIs, and therefore are portable across all families and platforms (header and 'c' files).
- The 'drv_f' tree, populated by processor family specific driver components and exporting device specific properties header files.
  Processor family-specific driver components are drivers which use one family specific DKI API, and therefore can run only on this processor family (header and 'c' files).

Note that both drv and drv_f trees are mainly populated by third party driver writers (although ChorusOS system deliveries contain drivers for the reference platform's devices).

The main functional components of the 'dki' tree are:

- `<dki/dki.h>` which defines the Common DKI API
- `<dki/f_dki.h>` which defines the processor family-specific DKI API

All other file trees are organized following the bus/device class provided APIs. In other words, there is a directory per class of bus and device, which contains the header file defining the API provided by this device class.

For drv and drv_f, in each bus/device class directory there is one directory per bus/device hardware controller for which a driver component is written.

Listed below are some path examples (header file paths are relative to the ChorusOS operating system delivery root directory):

```
include/chorus/ddi/bus/bus.h          -> DDI's Common bus class API
include/chorus/ddi/pci/pci.h          -> DDI's PCI bus class API
include/chorus/ddi/uart/uart.h        -> DDI's UART device class API

drv_f/src/pci/raven/ravenProp.h       -> family specific driver component
drv_f/src/pci/raven/raven.h              for the Motorola RAVEN PCI host
drv_f/src/pci/raven/raven.c              bridge
```

(continued)

```
drv_f/src/pci/raven/Imakefile

drv/src/uart/ns16550/ns16550Prop.h    -> Generic driver component for
drv/src/uart/ns16550/ns16550.h             NS16x50 compatible UART devices.
drv/src/uart/ns16550/ns16550.c
drv/src/uart/ns16550/Imakefile
```

Manpage Documentation

Typically, there is one manpage for each written driver component. The manpage file for a `devx` driver component is called '`devx.9drv`' and accessible through the `devx` name. This manpage contains the following information:

- the hardware that can be serviced by the driver
- the driver name
- the driver framework features and mechanisms that are implemented in the driver (probing, dynamic loading, and so on)
- the description of device tree node properties used by the driver

Device Driver Conventions

There are several conventions one should be aware of when writing device drivers, pertaining to:

- Driver Names
- Driver Information
- Message Logging
- Use of ASSERT Macro

Driver Names

Driver names in the Driver Framework must follow the following conventions, in this order:

1. driver vendor name
2. bottom interface used by driver
3. chip supported by driver
4. top interface used by driver

For example:

```
sun:bus-ns16550–uart
sun:pci-cheerio-ether
```

where ”sun” is the vendor name, “bus” and “pci” are the bottom interfaces used, “ns16550” and “cheerio” are the chips supported, and “uart” and “ether” are the top interfaces.

In the case of a driver providing several top interfaces, these interfaces are specified within parentheses, separated by commas:

```
sun:bus-mc146818–(rtc, timer)
```

Driver Information

Driver information is stored in the driver registry record, using the `drv_info` field. This info string must be built after the driver description and source management version information of the driver module.

For example:

NS16x50 UART driver [#ident \"@(#)ns16550.c 1.16 99/02/16 SMI\"]

Message Logging

Because messages are processed through the ChorusOS operating system, drivers must never use `sysLog` or `printf` directly to display messages. The ChorusOS operating system provides the following macros to handle message logging:

```
DKI_MSG  ((format, ...))     // typically does: printf
DKI_WARN ((format, ...))      // typically does: printf + syslog
DKI_PANIC((format, ...))      // typically does: printf + syslog + callDebug
DKI_ERR((format, ... ))      //typically does: printff + syslog
```

Moreover, message format conventions are as follows:

```
DKI_MSG    ->        "<name>: <message>"
DKI_WARN   ->      "<name>: warning – <message>"
DKI_ERR    ->        "<name>: error – <message>"
DKI_PANIC ->      "<name>: panic – <message>"
```

where `<name>` is either:

- the name of the driver (if the message is not related to a particular instance of the driver)
- the path of the device in the device tree (if the message is related to a driver instance)

Note - The `dtreePathLeng()` and `dtreePathGet()` calls can be used to get the device tree path for a particular instance.

Use of ASSERT Macro

ASSERT is a macro (fully defined in `util/macro.h`) which should only be used in situations which *should not logically be possible* in construction of the software.

Situations such as critical resource allocation failures should be handled with the `DKI_ERR` macro instead.

ASSERT is enabled at compile time only, with

```
#define DEBUG
```

CHAPTER 2

Driver Kernel Interface Overview

This chapter describes both the common and family-specific DKI services available within the Driver Framework. Understanding the functions in this overview will make the tasks shown in the next chapters (writing device drivers and writing bus drivers) much clearer.

Refer to section `9DKI` of the man pages for a complete description of each of the commands included in this chapter.

Common Driver Services

This section describes services that are common to all processor families.

Synchronization

Synchronization services (handling calls for the same services from different threads) are performed through the DKI thread. This thread is typically used for the shutdown and initialization of drivers, so it makes sense that synchronization services be handled within the DKI thread as well. The DKI thread is launched by the ChorusOS operating system microkernel at initialization time.

By ensuring this type of synchronization the DKI thread avoids using any other synchronization mechanism (locks) in the driver implementations.

The DKI thread acts as a synchronization mechanism in the following two cases:

Normal Case

In the normal case, all calls related to initialization/shutdown of the drivers are performed implicitly in the context of the DKI thread. This means that drivers need not be concerned with synchronization issues, because their routines are called directly from the DKI thread.

Special Cases

There are two special cases in which a driver must use DKI thread services to ensure synchronization:

- Hot-pluggable device drivers

  With a hot-pluggable device driver, the initialization/shutdown process must be executed at runtime (not as part of the kernel/drivers initialization process). In this case, drivers use DKI thread services (described below) to provide synchronization with any running drivers.

- Deferred driver initialization

  In some cases, a device driver may defer its initialization until it is opened. In this scheme, initialization/shutdown processes are executed at runtime (at time of open/close) and not as part of the kernel/driver's initialization process. Thus, this kind of driver uses thread services to synchronize with drivers that are already running.

  This is a way to resolve conflicts that arise when the same resources are used by multiple drivers. By using deferred driver initialization, drivers which share resources can be loaded at the same time (as long as they are not opened at the me time).

DKI thread related services are described below. See the man pages for complete descriptions of the commands listed:

svDkiThreadCall

synchronously invokes a routine in the context of the DKI thread.

svDkiThreadTrigger	asynchronously invokes a routine in the context of the DKI thread.

Device and Driver Registration

The driver and device registry mechanisms, described briefly in the first chapter, are explained in more detail below.

Device Tree

The device tree is a data structure providing a description of the hardware topology and device properties of a given device. The hardware topology is specified in terms of parent/child relationships. Device properties associated with each device node in the tree are device specific.

A device property is a name/value pair. The property name is a null terminated ASCII string. The property value is a stream of bytes specified by the length/address pair. Note that the property value format is property specific and has to be standardized between the property producer and its consumers.

For instance, among all device node properties, there are some related to the bus resources allocated to the device (for example, interrupt lines, I/O registers, DMA channels). These properties must be standardized to be understood by the bus driver, as well as any device drivers connected to the given bus.

The device tree data structure may be built either statically or dynamically.

- In the static case, the device tree is populated by the system booter.

  For instance, the system booter may include a pre-defined sequence of device tree function calls. Another possibility for the system booter is to build the device tree from a hardware description provided by firmware.

- In the dynamic case, the device tree is populated at system initialization time using an enumeration/probing mechanism. The device tree is populated by propagating from parent to children.

Note that it is possible to combine both methods. In other words, an initial (incomplete) device tree may be provided by the ChorusOS operating system booter, which will later be completed dynamically using an enumeration/probing mechanism. In any case, the device tree structure can be modified (extended/truncated) dynamically at run time using hot-plug insertion/removal service (for example, when using PCMCIA cards).

Device Tree related services are described below. See the man pages for complete descriptions of the commands listed:

Device Tree Browsing

dtreeNodeRoot	returns the root device node
dtreeNodeChild	returns the first child node
dtreeNodePeer	returns the next “sibling” device node
dtreeNodeParent	returns the parent device node
dtreePathLeng	returns the pathname length of the given device node
dtreePathGet	returns, in *buf*, the absolute pathname of the given device node. The trailing part of the pathname is the name of the node and is read in a `node` property. If this property does not exist, the trailing part of the returned pathname is set to ’???’.

Device Tree Modification

dtreeNodeAlloc	allocates a new device node object
dtreeNodeFree	releases all memory and properties attached to the node
dtreeNodeAttach	adds a child node to the specified parent
dtreeNodeDetach	detaches a node from its parent

Device Node Properties

dtreePropFind	return the first property of a node
dtreePropFindNext	return the next property of a node
dtreePropLength	returns the property value length (in bytes)

dtreePropValue	returns a pointer to the first byte of the property value
dtreePropName	returns a pointer to the property name
dtreePropAlloc	allocates a new device property object
dtreePropFree	releases the memory allocated by the property object
dtreePropAttach	attaches a property object to a device node
dtreePropDetach	detaches a property object from a device node

Device tree high-level services

dtreeNodeAdd	adds a named device node to the tree
dtreeNodeFind	looks for a named node in the list of children of a given device node
dtreePropAdd	allocates a new property, sets its value and attaches it to a given device node

Driver Registry

The driver registry module implements a data base of drivers registered in the ChorusOS operating system. The driver registry data base is populated by drivers which perform self-registration (using `svDriverRegister`) at driver initialization time.

The bus/nexus drivers perform a search in the driver registry data base to find a driver they are interested in. Typically, there are two kinds of searches used by the bus/nexus drivers. The first one is done at device enumeration/probing time when the bus/nexus driver is interested in all drivers matching the bus/nexus class (specified as the parent device class). The second is at device instance creation time, when the bus/nexus driver looks for a driver which must be started for a particular device node.

Driver Registry related services are described below. See the man pages for complete descriptions of the commands listed:

svDriverRegister	adds a driver entry to the driver registry

svDriverLookupFirst	returns the id of the first driver entity
svDriverLookupNext	returns the id of the next driver entity
svDriverRelease	releases the lock of a driver
svDriverEntry	returns a pointer to the driver entry structure (using an id)
svDriverCap	returns a pointer to the driver actor capability (using an id)
svDriverUnregister	removes a driver entry from the registry

Device Registry

The device registry microkernel module implements a data base of driver instances servicing devices currently supported by the system. The device registry data base is populated by drivers that perform self-registration (using `svDeviceRegister`) at device initialization time.

The device registry data base is accessed by driver clients in order to obtain a pointer to the driver instance servicing a given (logical) device.

The device registry API is described in detail in the man pages. Note that only the `svDeviceLookup`, `svDeviceRelease` and `svDeviceEntry` microkernel calls should be used by driver clients. The rest of API is dedicated to device drivers.

Device Registry related services are described below. See the man pages for complete descriptions of listed commands:

svDeviceAlloc	allocates a device registry entry for a given device driver instance
svDeviceRegister	adds a given entry to the device registry
svDeviceUnregister	removes an entry from the device registry
svDeviceEvent	notifies the device registry module that a given event has occurred
svDeviceFree	releases a previously allocated device registry entry
svDeviceLookup	searches a device entry in the registry, matching given device class and logical unit.

svDeviceEntry	returns the device entry associated to a client identifier returned by `svDeviceLookup`
svDeviceRelease	releases the lock on a looked-up device entry

General Purpose Memory Allocation

The microkernel provides general purpose memory management services for device drivers that need to dynamically allocate and free pieces of memory in supervisor memory space. As initialization schemes are normally dynamic, device drivers need to dynamically allocate and free small pieces of supervisor data.

Typically, a device driver needs to dynamically allocate data associated to each instance that it will register in the Device Registry at initialization time. Moreover, most of the DDI services called from base level by the driver clients lead to the dynamic allocation and freeing of certain linked list elements for internal management purposes.

Note - The memory allocated using these services is anonymous. That means it is not associated to any actor context. For this reason, all the allocated memory must be freed by drivers before they terminate, as the kernel won't be able to do it at actor deletion time.

General purpose memory allocation related services are described below. See the man pages for complete descriptions of listed commands:

svMemAlloc	allocates a specified amount of memory from the supervisor address space
svMemFree	frees memory previously allocated with `svMemAlloc`

Special Purpose Physical Memory Allocation

Typically, different I/O buses impose different constraints on the memory used by their devices for Direct Memory Access (DMA), such as alignment, specific boundary crossing, maximum size, or specific location within the physical memory space.

To satisfy all constraints on physical memory imposed by the different I/O buses, (mainly for DMA purposes), the DKI provides an interface to allocate and free special purpose physical memory that satisfy the given constraints.

Special purpose memory allocation related services are described below. See the man pages for complete descriptions of listed commands:

svPhysAlloc	allocates contiguous physical memory
svPhysFree	frees memory allocated with `svPhysAlloc`

Timeouts

Device drivers may need timeout services to check whether there is activity on a device, or to verify that a started action will terminate before a given time limit is reached.

Note - As these services should be implemented using drivers, they are not available and must not be used by drivers at initialization time.

Timeout related services are described below. See the man pages for complete descriptions of listed commands:

svTimeoutSet	sets a timeout request
svTimeoutCancel	cancels a timeout request
svTimeoutGetRes	returns the smallest possible difference between two distinct “time” values

Precise Busy Wait

Device drivers may use precise busy wait services to wait for a very short time. Note that busy wait means that the caller waits without releasing the CPU, as if executing a busy loop.

Note that these services may be used during the driver initialization process.

Precise busy wait related services are described below. See the man pages for complete descriptions of listed commands:

usecBusyWait	waits for (at least) the given number of micro-seconds

System Event Management

System event management services are provided by the microkernel to the lowest-layer drivers. They are intended to register event handlers for all the running drivers, and to start propagating events from the microkernel.

Typically a system reboot starts propagating a specific event from the microkernel to the lowest-layer drivers. Those drivers then recursively propagate the event to the upper layer drivers by calling their event handler (`BusEventHandler`) registered at open time).

System event management related services are described below. See the man pages for complete descriptions of listed commands:

svDkiOpen	establishes connection between a child device driver and the DKI
svDkiClose	releases the DKI/driver connection
svDkiEvent	starts the propagation of an event to the device driver hierarchy

Global Interrupts Masking

Some of the Interrupt Management Service (IMS) routines are included as part of the DKI to provide drivers with global interrupts masking services.

These services may be used by a driver to protect a critical section from interrupts, if needed.

Global interrupts masking related services are described below. See the man pages for complete descriptions of listed commands:

imsIntrMask_f	masks all maskable interrupts at processor level, and increments `imsIntrMaskCount_f` kernel variable
imsIntrUnmask_f	unmasks interrupts at processor level (if calls are not nested)

Thread Preemption Disabling

The DKI API provides a means for a driver to disable/enable the preemption of the current thread. These services may be useful for a driver to prevent the current thread being preempted while interrupts are masked at bus/device level. Note that these services are implemented as macros.

DISABLE_PREEMPT()	disables preemption of the currently executed thread. Basically, this macro increments a per-processor preemption mask count. When the preemption mask count is not zero, the ChorusOS scheduler is locked, such as when there is a preemption request, the scheduler just raises a pending preemption flag deferring the real thread preemption until the preemption mask count drops to zero
ENABLE_PREEMPT()	enables preemption of the currently executed thread which has been previously disabled by DISABLE_PREEMPT(). Basically, this macro decrements the preemption mask count and, if it drops to zero, checks whether the current thread should be preempted because the pending preemption flag is raised.

Note that, as DISABLE_PREEMPT()/ENABLE_PREEMPT() rely on the preemption mask count, a driver may issue nested calls to these services.

Specific Input/Output Services

The DKI provides specific I/O routines optimized to handle byte swapping. Typically, these services are intended to be used by a host bus driver to handle different byte ordering between the processor bus and the host bus.

Specific I/O services are defined below as sets of routines where the _xx suffix indicates the bit length of the data on which the services apply. This suffix may take one of the following values:

- _16 for 16-bit data
- _32 for 32-bit data
- _64 for 64-bit data

Specific input/output related services are described below. See the man pages for complete descriptions of listed commands:

loadSwap_xx	loads data from a given address and returns the corresponding byte swapped value. The *addr* argument specifies the address to read from
storeSwap_xx	stores into a given address the value byte swapped
swap_xx	swap in place the bytes of the data stored at a given address

Processor Family Specific DKI Services

Processor family specific DKI services are available only on a given processor family, and should be used only by the drivers servicing devices directly connected to the local CPU bus. Drivers using these DKI services become "processor specific" and therefore can not be considered common.

Note that the availability of services is different between processor families, and not all services are listed here. An overview of commonly available services is provided below. For an accurate indication of what services are provided for each processor family see the “Processor-Specific DKI Services” section in Appendix A (or the appropriate `9DKI` man pages).

Depending on the processor family architecture, the family-specific DKIs may offer the following services:

Interrupt Management

All processor families offer DKI services to manage interrupts. These services allow the driver to perform the following tasks:

- attach a handler to a given interrupt
- mask an interrupt attached to a handler
- unmask an interrupt attached to a handler
- detach an interrupt handler

Cache management	Allows the host bus to manage memory coherence for DMA purposes by flushing and/or invalidating caches
Specific I/O services	Provides interface to processor specific I/O instructions, managing: ■ I/O ports ■ Synchronization of memory mapped I/O operations
Physical to virtual memory mapping	allows device drivers to map physical space to virtual memory space. These services are used mainly by the host bus driver to map bus I/O space or DMA memory.

CHAPTER **3**

Writing Device Drivers

This chapter takes a procedural approach to writing device drivers, using as its example a leaf device driver. It follows a step-by-step, tutorial approach, explaining at each step the Driver Framework APIs and mechanisms involved.

The ChorusOS operating system Driver Framework provides numerous device and bus drivers as part of its installed package, and more are being developed constantly. It is recommended that you take a look at the provided drivers as an example of how to build your own. For information on the drivers provided with this release, as well as the location of the code and header file for the drivers, see the 9DRV library of the man pages.

Include the Appropriate APIs (DKI/ DDI)

As the first step, include the header files for the DKI and DDI APIs involved in the device driver's implementation. A device driver mainly uses its parent bus DDI API (and some generic DKI services like memory allocator).

The driver implementation must include:

- The parent bus class API header file(s) (DKI and DDI) used by the driver.

  Note that the services defined in these header files are those available to the writer to implement its device driver.
- The device class API header file(s) (DDI).

  Note that these header files define the sets of routines that have to be written in the driver component, to be compliant with the Driver Framework.

Here is an example for a NS16x50 compatible UART device driver that uses DKI and "Common bus driver" APIs, and provides "UART device driver" DDI API.

```
#include    <dki/dki.h>
#include    <ddi/bus/bus.h>
#include    <ddi/uart/uart.h>
...
#include    "ns16550.h"
#include    "ns16650Prop.h"
```

Register the Driver (using `main` function)

A driver component may be downloaded in various ways. It may be built into the system bootable image, or it may be downloaded dynamically as a supervisor actor using the `afexec` system call.

In either case, the driver code must contain the `main` routine which is called by the system once the driver component is downloaded. The only task of the driver `main` routine is to perform the self-registration of the driver component within the ChorusOS system.

To accomplish this task, the driver invokes the `svDriverRegister` microkernel call passing as an argument a `DrvRegEntry` data structure which specifies the driver component properties. Once the driver component is self-registered, the future driver management is largely undertaken by its parent bus/nexus driver using the properties specified in the `DrvRegEntry` structure.

`DrvRegEntry` specifies four driver entry points as follows:

drv_probe	invoked by the parent bus/nexus driver when *bus_class* specified in the registry entry matches the parent bus/nexus driver class
drv_bind	invoked by the parent bus/nexus driver when *bus_class* specified in the registry entry matches the parent bus/nexus driver class

drv_init	invoked by the parent bus/nexus driver when *bus_class* specified in the registry entry matches the parent bus/nexus class and there is a node in the device tree to which the driver is bound (a hardware device to be managed by the driver exists)
drv_unload	invoked by the driver registry module when an application wishes to unload the driver component from the system

```
    /*
     * NS16550 driver registry entry.
     *
     * The driver requires the common bus driver interface
     * implemented by the parent bus driver. Thus, the driver
     * should work on any bus providing such an API (e.g. ISA,
     * PCI, PCMCIA).
     *
     * Note that the driver does not provide any probe routine because
     * the probing mechanism is bus class specific. Thus, if one wants
     * to implement the NS16550 device probing, it may either add probe
     * routine(s) to the driver code or implement probe-only driver(s)
     * for NS16550 device.
     */
static DrvRegEntry ns16_drv = {
    NS16_DRV_NAME,              /* drv_name    */
    "NS16x50 UART driver [#ident \"@(#)ns16550.c 1.16 99/02/16 SMI\"]",
    BUS_CLASS,           /* bus_class   */
    BUS_VERSION_INITIAL,        /* bus_version */
    NULL,            /* drv_probe   */
    NULL,            /* drv_bind   */
    ns16_init,           /* drv_init    */
    ns16_unload          /* drv_unload  */
};

    /*
     * Driver main() routine.
     * Called by microkernel at driver initialization time.
     */
    int
main ()
{
    KnError res;
   /*
    * Register the driver component in the driver registry.
    */
    res = svDriverRegister(&ns16_drv);
    if (res != K_OK) {
       DKI_ERR(("%s: error -- svDriverRegister() failed (%d)\n",
                  NS16_DRV_NAME, res));
   }
   return res;
}
```

Write Device Driver-Class-Specific Functions

In this next step, you will write an implementation of the services specific to the driver device class for a hardware device of the given class.

Once the code is written, these functions must be provided (made available) to the device driver's clients.

Note - None of these functions are directly exported, but all of them are defined as static functions, and then grouped as indirect calls in an "operations data structure" (typed by the device class API). The device driver component then provides this "operations data structure" as a property when registering an instance of itself at initialization time.

This way of providing the device driver operations allows for a dynamic binding mechanism between device drivers and driver's clients.

Each device class API is different. Thus, the functions to write are different for different classes of device API. The complete list of the currently defined device class APIs may be found in the ddi(9) man page.

The following code example illustrates this step for an NS16x50 compatible UART device driver. In this example, the provided device class API is for the UART device class.

Note - For clarity, only the code pertinent to this explanation is presented. Please refer to the complete implementation file for more details.

The NS16_DEV_REMOVAL compilation flag is used to allow downsizing of the driver component, at compile time, in case the device removal mechanism is not needed.

```
    /*
     * Open device.
     */
    static int
ns16_open (UartId id, ...) { ... }
    /*
     * Disable device interrupts.
     */
    static void
ns16_mask (UartId id) { ... }
    /*
```

```
     * Enable device interrupts.
     */
    static void
ns16_unmask (UartId id) { ... }
    /*
     * Send a buffer.
     */
    static void
ns16_transmit (UartId id, ...) { ... }
    /*
     * Abort an output in progress.
     */
    static void
ns16_abort (UartId id) { ... }
    /*
     * Send a break
     */
    static void
ns16_txbreak (UartId id) { ... }
    /*
     * Set/reset the modem control signals.
     */
    static void
ns16_control (UartId id, ...) { ... }
    /*
     * Set receive buffer.
     */
    static void
ns16_rxbuffer (UartId id, ...) { ... }
    /*
     * Close device.
     */
    static void
ns16_close (UartId id) { ... }
    /*
     * NS16550 service routines:
     */
static UartDevOps ns16_ops =
{
    UART_VERSION_INITIAL,
    ns16_open,
    ns16_mask,
    ns16_unmask,
    ns16_transmit,
    ns16_abort,
    ns16_txbreak,
    ns16_control,
    ns16_rxbuffer,
    ns16_close
};
    /*
     * Init the NS16x50 uart. Called by BUS driver.
     */
    static void
ns16_init (DevNode node, void* pOps, void* pId)
{
    BusOps*       busOps = (BusOps*)pOps;
    Ns16_Device*  dev;
    ...
        /*
```

```
         * Allocate the device descriptor
         * (i.e. the driver instance local data)
         */
    dev = (Ns16_Device*)svMemAlloc(sizeof(Ns16_Device));
    ...
    dev->entry.dev_class = UART_CLASS;
    dev->entry.dev_id    = dev;
    dev->entry.dev_node  = node;
    ...
#if defined(NS16_DEV_REMOVAL)
    bcopy(&ns16_ops, &(dev->devOps), sizeof(ns16_ops));
    dev->entry.dev_ops = &(dev->devOps);
#else
    dev->entry.dev_ops = &ns16_ops
#endif
        /*
         * Allocate the device driver instance descriptor in the
         * device registry.
         * Note that the descriptor is allocated in an invalid state
         * and it is not visible for clients until svDeviceRegister()
         * is invoked.
         * On the other hand, the allocated device entry allows the
         * event handler (ns16_event) to invoke svDeviceEvent() on it.
         * If svDeviceEvent() is called on an invalid device entry,
         * the shutdown processing is deferred until svDeviceRegister().
         * In other words, if a shutdown event occurs during the
         * initialization phase, the event processing will be deferred
         * until the initialization is done.
         */
    dev->regId = svDeviceAlloc(&(dev->entry),
                               UART_VERSION_INITIAL,
                               FALSE,
                               ns16_release);
    ...
        /*
         * Finally, we register the new device driver instance
         * in the device registry. In case when a shutdown event
         * has been signaled during the initialization, the device entry
         * remains invalid and the ns16_release() handler is invoked
         * to shutdown the device driver instance. Otherwise, the device
         * entry becames valid and therefore visible for driver clients.
         */
    svDeviceRegister(dev->regId);

    DKI_MSG(("%s: %s driver started\n", dpath, NS16_DRV_NAME));
}
```

Write Device Driver Registry Functions

Write the Probe Function

The purpose of the probe routine is to detect devices residing on the bus and to create device nodes corresponding to these devices.

The probe routine is optional, in cases where it is not provided (NULL entry point), a device node should be statically created at boot time, or should be created by another "probe only" driver component to activate the bus driver.

Actions taken by a probe routine may be summarized as follows:

- The probe routine creates nodes if, and only if, they do not already exist. In other words, the probe routine is forbidden to create redundant nodes.
- The probe routine specifies a physical device ID as a device node property so that the bus driver can find the appropriate device driver for this device node. Note that the device ID is bus class specific. For instance, on PCI bus, the device ID is the vendor/device IDs pair.
- The probe routine specifies resource requirements as device node properties so that the bus driver can reserve resources required to initialize the device.

Basically, there are two kinds of probe routines:

- generic (bus class specific only)
- device specific (bus class and device specific)

A self-identifying bus (such as PCI) enumerator is a typical example of the generic probe routine.

A device probing code on an ISA bus is a typical example of the device specific probe routine.

Note that multiple probe routines for a given bus may be found in the driver registry. The Driver Framework does not specify the order in which the probe routines will be run. In addition, the probe routines may be invoked at run time when, for example, the device insertion is detected on the bus. In the latter case, the probe routines must be extremely careful about active device nodes (existing device nodes for which the device drivers have been already started and may be already in use).

The following rules must be respected by generic and device specific probe routines:

- The generic and specific probe routines must access the device hardware only through the bus service routines. The bus resources needed to access the device hardware (such as I/O registers) must be allocated through the bus service routines (`resource_alloc`). This prevents the probe routine from accessing hardware which is currently in use. Upon unsuccessful probing, the used hardware resources must be released through the bus service routines (`resource_free`).
- Neither generic nor specific probe routines are allowed to delete active device nodes or modify their properties. An active device node is defined as a node for which a device driver is already running. These nodes are flagged with the `"active"` property.
- Device specific probe routines are allowed to override properties in an existing node or to delete existing nodes.
- Generic probe routines are not allowed to override properties in existing nodes or to delete existing nodes. In other words, device specific probe routines have higher priority than generic ones.
- No probe routine is allowed to create redundant nodes. To run a probe routine, either you must be positive that no other node exists for this device, or you must be able to find any other nodes for this device. If for some reason it is impossible to avoid creating redundant nodes, you cannot probe.

Write the Bind Function

The bind routine enables the driver to perform a driver-to-device binding. Typical actions taken by a bind routine may be summarized as follows:

The driver examines properties attached to the device node to determine the type of device and to check whether the device may be serviced by the driver. Note that the properties examined by the driver are typically bus architecture specific. For instance, a PCI driver would examine the vendor and device identifier properties.

If the check is positive, the driver attaches a "`driver`" property to the device node. The property value specifies the driver name.

The parent bus/nexus driver should use the `"driver"` property to determine the name of the driver servicing the device. The child driver gives its name to the parent bus driver, through the `"driver"` property, asking the parent bus driver to invoke the `drv_init` routine on that device.

Note that, if a `"driver"` property is already present in the device node, then the `drv_bind` routine can not continue; `drv_bind` should not override an existing driver-to-device binding.

The driver-to-device binding mechanism used in the framework enables multiple implementations. A simple bind routine may be implemented by a device driver.

Such an implementation would be device specific, only taking into account the devices known by the driver to be compatible with the driver's reference device.

Let us consider systems that support after-market, hot-plug devices and consult a network lookup service to locate the driver for a new device. It would be reasonable to provide a separate binder driver that would implement a smart driver-to-device mapping and a driver component download. Note that such a (generic) binder appears in the driver framework as a normal driver component. The binder driver provides the bind routine only and does not provide the probe and initialize routines.

Write the Init Function

The initialization routine of a device driver component is optional. In case it is not provided (NULL entry point), the driver is typically a "probe only" driver.

The initialization process (`drv_init`) of a leaf device driver goes through the following steps:

- establishes connection to the parent bus/nexus driver
- allows access to the device hardware
- initializes the device hardware to an operational state
- registers the device driver instance in the device registry

First of all, the driver must establish connection to the parent driver by calling `open` . In `open`, the driver specifies call-back handlers which will be used by the parent driver to manage the device instance driver (such as to shutdown the driver instance). In addition, global bus events (such as a catastrophic bus error) are delivered to the driver through a call-back handler.

Once the child-to-parent connection is established, the driver may use services provided by the parent driver. Typically, at this point, the driver asks its parent driver to make the bus hardware resources needed for the device available.

Note - The bus resources needed for the device are specified as properties in the device node. These resources are already allocated by the parent bus/nexus driver prior to the `drv_init` invocation. To make a given bus resource available (such as device I/O registers), the driver obtains an appropriate property value ("io-regs") and calls an appropriate bus/nexus service routine (such as `io_map`).

Once access to the device hardware is allowed, the driver initializes it to an operational state. Once initialized, the driver performs self-registration in the device registry to declare itself as a new device (such as a new device driver instance) within the system.

Once the device (the device driver instance) is registered, a driver client may find it in the registry (using the device registry API) and may perform operations on device-calling driver service routines exported through the device registry entry.

Note - The driver is not necessarily required to register any device driver instances or offer any device service routines *through the device registry*. Device driver instance registration is required only for clients that find their devices through the device registry. If other client-to-driver binding mechanisms are in use, the associated devices need not take part in the device registry.

The `drv_init` routine is called in the context of the DKI thread. This makes it possible to directly invoke the bus / nexus and DKI services allowed in the DKI thread context.

Below is an example of the initialization function of the NS16x50 compatible UART device driver. The NS16_DEV_REMOVAL and NS16_DRV_UNLOAD compilation flags are used to allow downsizing of the driver component at compile time in case these mechanisms are not needed.

```
    /*
     * Init the NS16x50 uart. Called by BUS driver.
     */
    static void
ns16_init (DevNode node, void* pOps, void* pId)
{
    BusOps*       busOps = (BusOps*)pOps;
    DevProperty   prop;
    Ns16_Device*  dev;
    void*         ioRegs;
    void*         intr;
    KnError       res;
    char*         dpath;
    int           dpathLeng;

    dpathLeng = dtreePathLeng(node);
    dpath     = (char*) svMemAlloc(dpathLeng);
    if (!dpath) {
        DKI_ERR(("%s: error -- no enough memory\n", NS16_DRV_NAME));
        return;
    }
    dtreePathGet(node, dpath);

        /*
         * Allocate the device descriptor
         * (i.e. the driver instance local data)
         */
    dev = (Ns16_Device*)svMemAlloc(sizeof(Ns16_Device));
    if (dev == NULL) {
        DKI_ERR(("%s: error -- no enough memory\n", dpath));
        return;
    }

    bzero(dev, sizeof(Ns16_Device));

    dev->dpath           = dpath;
    dev->dpathLeng       = dpathLeng;
    dev->busOps          = busOps;
    dev->devEvent        = DEV_EVENT_NULL;
    dev->entry.dev_class = UART_CLASS;
    dev->entry.dev_id    = dev;
    dev->entry.dev_node  = node;

        /*
         * If the hot-plug removal is supported, the device driver ops
         * are located within the device descriptor. It makes possible
         * to substitute the service routines (to an empty implementation)
         * when a hot-plug removal occurs.
         */
#if defined(NS16_DEV_REMOVAL)
    bcopy(&ns16_ops, &(dev->devOps), sizeof(ns16_ops));
    dev->entry.dev_ops = &(dev->devOps);
#else
    dev->entry.dev_ops = &ns16_ops
#endif

        /*
         * Allocate the device driver instance descriptor in the
         * device registry.
         * Note that the descriptor is allocated in an invalid state
         * and it is not visible for clients until svDeviceRegister()
         * is invoked.
         * On the other hand, the allocated device entry allows the
         * event handler (ns16_event) to invoke svDeviceEvent() on it.
         * If svDeviceEvent() is called on an invalid device entry,
         * the shutdown processing is deferred until svDeviceRegister().
         * In other words, if a shutdown event occurs during the
         * initialization phase, the event processing will be deferred
         * until the initialization is done.
         */
    dev->regId = svDeviceAlloc(&(dev->entry),
                               UART_VERSION_INITIAL,
                               FALSE,
                               ns16_release);
    if (!dev->regId) {
        DKI_ERR(("%s: error -- no enough memory\n", dpath));
        svMemFree(dev, sizeof(Ns16_Device));
        svMemFree(dpath, dpathLeng);
        return;
    }

        /*
         * Retrieve the device I/O base addr from device tree.
         */
    prop = dtreePropFind(node, BUS_PROP_IO_REGS);
    if (prop == NULL) {
        DKI_ERR(("%s: error -- no ’%s’ property\n", dpath, BUS_PROP_IO_REGS));
        svDeviceFree(dev->regId);
        svMemFree(dev, sizeof(Ns16_Device));
        svMemFree(dpath, dpathLeng);
        return;
    }
    ioRegs = dtreePropValue(prop);

        /*
         * Retrieve the device interrupt source from device tree.
         */
```

```
prop = dtreePropFind(node, BUS_PROP_INTR);
if (prop == NULL) {
    DKI_ERR(("%s: error -- no '%s' property\n", dpath, BUS_PROP_INTR));
    svDeviceFree(dev->regId);
    svMemFree(dev, sizeof(Ns16_Device));
    svMemFree(dpath, dpathLeng);
    return;
}
intr = dtreePropValue(prop);

    /*
     * Retrieve the device clock frequency from device tree.
     * (if not specified, the default clock frequency is used)
     */
prop = dtreePropFind(node, PROP_CLOCK_FREQ);
if (prop == NULL) {
    dev->clock = NS16_CLOCK_FREQ;
} else {
    dev->clock = *(PropClockFreq*)dtreePropValue(prop);
}

    /*
     * Open a connection to the parent bus.
     */
res = busOps->open(pId, node, ns16_event, NULL, dev, &dev->devId);
if (res != K_OK) {
    DKI_ERR(("%s: error -- open() failed (%d)\n", dpath, res));
    svDeviceFree(dev->regId);
    svMemFree(dev, sizeof(Ns16_Device));
    svMemFree(dpath, dpathLeng);
    return;
}

    /*
     * Map the device I/O registers.
     */
res = busOps->io_map(dev->devId, ioRegs, ns16_bus_error, dev,
                     &dev->ioOps, &dev->ioId);
if (res != K_OK) {
    DKI_ERR(("%s: error -- io_map() failed (%d)\n", dpath, res));
    busOps->close(dev->devId);
    svDeviceFree(dev->regId);
    svMemFree(dev, sizeof(Ns16_Device));
    svMemFree(dpath, dpathLeng);
    return;
}

    /*
     * Connect interrupt handler to the bus interrupt source
     * (mask interrupts at chip level first).
     *
     * Note that the mask() routine is invoked indirecty because
     * it may be substituted by the event handler (if a device
     * removal event has been already occured).
     */
UART_OPS(dev->entry.dev_ops)->mask((UartId)dev);
res = busOps->intr_attach(dev->devId, intr, ns16_intr, dev,
                          &dev->intrOps, &dev->intrId);
if (res != K_OK) {
    DKI_ERR(("%s: error -- intr_attach() failed (%d)\n", dpath, res));
```

```
        busOps->io_unmap(dev->ioId);
        busOps->close(dev->devId);
        svDeviceFree(dev->regId);
        svMemFree(dev, sizeof(Ns16_Device));
        svMemFree(dpath, dpathLeng);
        return;
    }

        /*
         * If the driver unloading is supported, the list of active
         * device driver instances is handled.
         * Thus, we should add the new driver instance to the list.
         */
#if defined(NS16_DRV_UNLOAD)
    dev->next = ns16_devs;
    ns16_devs = dev;
#endif

        /*
         * Finally, we register the new device driver instance
         * in the device registry. In case when a shutdown event
         * has been signaled during the initialization, the device entry
         * remains invalid and the ns16_release() handler is invoked
         * to shutdown the device driver instance. Otherwise, the device
         * entry becames valid and therefore visible for driver clients.
         */
    svDeviceRegister(dev->regId);

    DKI_MSG(("%s: %s driver started\n", dpath, NS16_DRV_NAME));
}
```

Write Unload Function

drv_unload is called by the driver registry module (more precisely by the svDriverUnregister routine) when an application wishes to unload the driver component from the system. The drv_unload routine is called in the context of theyell0w DKI thread. This makes it possible to directly invoke the bus/nexus and DKI services allowed in the DKI thread context.

The purpose of drv_unload is to check that the driver component is not currently in use. For drv_unload to succeed, the driver clients must have closed their connections with the driver and released the device registry lock (svDeviceRelease). On success, drv_unload returns K_OK, otherwise K_EBUSY is returned.

The drv_unload routine is optional. In cases when drv_unload is not provided, the driver code cannot be unloaded.

The drv_unload is a global, per driver component routine. Therefore, to implement unloading, the driver should handle a list of driver instances. When drv_unload is called, the driver should go through the list, and for each driver instance, should check whether the driver instance is currently in use.

Note - Once the check is positive, (a given instance is not used), the driver instance must become invisible to potential clients. In other words, if `drv_unload` returns `K_OK`, all previously created driver instances (if any) must be deleted and all previously allocated system resources (if any) must be released.

If `drv_unload` returns `K_EBUSY`, the driver component will not be unloaded. In this case, the driver component state must not be changed by `drv_unload` . For instance, all registered driver instances must be in place.

Consider the driver unloading implementation for a driver using the standard client-to-driver binding mechanism based on the device registry. In cases where another client-to-driver binding mechanism is used, the driver unloading implementation is binding mechanism dependent.

The `drv_unload` routine of a (leaf) device driver typically takes the following actions:

1. Checks that the driver component is not in use.
   `drv_unload` iterates through the driver instances list and, for each driver instance, invokes `svDeviceUnregister` to remove the driver instance entry from the registry.

   Once `svDeviceUnregister` fails (returns `K_EBUSY`), the iteration is aborted and `drv_unload` proceeds to step 3. Otherwise (if all device instances are successfully unregistered), `drv_unload` proceeds to step 2.
2. Releases resources associated to the driver component.
   `drv_unload` iterates through the driver instances list and, for each driver instance, releases system resources associated to the instance (`io_unmap`, `mem_unmap`, ...) and, finally, closes the connection to the parent bus. Once the iteration is finished, `drv_unload` returns `K_OK`.
3. Restores the initial state of the driver component.
   `drv_unload` iterates through the driver instances list and, for each driver instance which has been unregistered at step 1, invokes `svDeviceRegister` to register the driver instance again. Once the iteration is finished, `drv_unload` returns `K_EBUSY`.

Note that `drv_unload` runs in the DKI thread context. This guarantees the stability of driver instances during the `drv_unload` execution. In fact, a new driver instance may be created only by the `drv_init` routine, which is also invoked in the DKI thread context. In this way, `drv_init` is serialized with `drv_unload` by the DKI thread.

The following example shows the unload function of the NS16x50 compatible UART device driver. The NS16_DRV_UNLOAD compilation flag is used to allow downsizing of the driver component, at compile time, in case the driver unload mechanism is not needed.

```
    /*
     * Unload the NS16x50 uart driver.
     * This routine is called by the driver registry when an application
     * wishes to unload the NS16550 driver component.
     */
    static KnError
ns16_unload ()
{
        /*
         * The driver unloading is an optional feature.
         * In case when such a feature is not supported, ns16_unload()
         * just returns K_EBUSY preventing the driver component to be
         * unloaded.
         */
#if defined(NS16_DRV_UNLOAD)
    Ns16_Device*  dev;
    Ns16_Device*  udev;
        /*
         * Go through the driver instances list and try to unregister
         * all instances. Note that the device registry entry becomes
         * invalid if svDeviceUnregister() returns K_OK. The iteration
         * is aborted if svDeviceUnregister() fails.
         */
    udev = ns16_devs;
    while (udev && (svDeviceUnregister(udev->regId) == K_OK)) {
        udev = udev->next;
    }
        /*
         * If all driver instances are unregistered successfully,
         * we invoke ns16_shutdown() for each instance in order to
         * shutdown it. Note that some shutdown events may be signaled by the
         * parent bus driver after the device entry has been unregistered.
         * In such a case, these events will be ignored. Indeed, once
         * unregistered, the device registry entry becomes invalid.
         * For invalid device entries, the device registry defers the events
         * processing until svDeviceRegister(). But, the entries will be
         * released (svDeviceFree) by ns16_shutdown() rather than registered
         * again.
         */
    if (!udev) {
        while (ns16_devs) {
            ns16_shutdown(ns16_devs);
        }
        return K_OK;
    }
        /*
         * If there is a driver instance in use (i.e. svDeviceUnregister()
         * failed) , the driver component cannot be unloaded.
         * We must register again the driver instances unregistered above.
         * Note that shutdown events may be signaled by the parent bus driver
         * after the device entry has been unregistered.
         * In such a case, these events will be processed at this moment.
         * Indeed, once unregistered, the device registry entry becomes
         * invalid. For invalid device entries, the device registry defers
         * the events processing until svDeviceRegister().
         */
    dev = ns16_devs;
    while (dev != udev) {
        svDeviceRegister(dev->regId);
        dev = dev->next;
    }
#endif

    return K_EBUSY;
}
```

Write Bus Events Handler Function

The event handler is invoked by the parent bus/nexus driver when a bus/nexus event occurs. The event handler address is given to the bus/nexus driver when a connection is established between the child driver and its parent bus/nexus driver. The event handler may be called as an interrupt handler and therefore the event handler implementation must be restricted to the API allowed at interrupt level.

Among all events which are mostly bus/nexus class specific, there are three shutdown related events (specified by the common bus API) which are discussed in this section:

SYS_SHUTDOWN	system emergency shutdown The `SYS_SHUTDOWN` event notifies the driver instance that the system is going to be shutdown. The parent bus/nexus driver requires the child driver instance to perform an emergency shutdown of the device hardware.
DEV_SHUTDOWN	normal device shutdown The `DEV_SHUTDOWN` event notifies the driver instance that a normal device shutdown is requested by the bus/nexus driver.
DEV_REMOVAL	surprise device removal The `DEV_REMOVAL` event notifies the driver instance that the associated device has been removed from the bus/nexus and therefore the driver instance has to be shutdown.

Note - The omitted prefix (`DKI_`, `BUS_`, ...) means that the event semantics are the same for all events.

In the standard case, the shutdown event processing goes through three phases:

1. shutdown prolog
2. shutdown mode
3. shutdown epilog

The shutdown prolog phase is processed synchronously within the event handler. The main purpose of the shutdown prolog is to notify driver clients that the shutdown event has occurred, in other words, to propagate the shutdown prolog downstream (from driver to clients).

Once the shutdown event is processed by driver clients, the driver enters the shutdown epilog phase. Basically, the shutdown epilog is invoked when the last reference on the driver instance goes away. Between the shutdown prolog and epilog, the driver operates in a special mode (called shutdown mode). In this mode, the driver accepts only a subset of operations from clients allowing proper closure of connections to the driver.

The table below shows typical actions taken by the shut-down prolog depending on the event type:

Action	SYS_SHUTDOWN	DEV_SHUTDOWN	DEV_REMOVAL
notify driver clients (with svDeviceEvent)	-	+	+
abort operations in progress	-	-	+
reset hardware	+	-	-

The `SYS_SHUTDOWN` prolog of a leaf (device) driver does not notify driver clients about the system shutdown event. The driver simply puts the hardware into a clean state.

Note that the `SYS_SHUTDOWN` event is processed synchronously, within the event handler. In other words, the system shutdown epilog is empty.

The only purpose of the `SYS_SHUTDOWN` event is to put the board hardware into a clean state to perform the system reboot (or restart) correctly.

The `DEV_SHUTDOWN` prolog simply notifies the driver clients that the `DEV_SHUTDOWN` event has occurred. Actual shutdown of the device is deferred until the `DEV_SHUTDOWN` epilog.

The `DEV_REMOVAL` prolog is closed to the `DEV_SHUTDOWN` one. In addition, the `DEV_REMOVAL` prolog aborts all I/O operations in progress (otherwise, these operations would never be completed).

Aborted operations return to callers with an error code.

As soon as the shutdown prolog is processed, the driver changes its internal state to enter into a shutdown mode. In this mode, the driver accepts only a subset of operations from client drivers:

- to abort queued operations
- to release previously allocated resources
- to close connection to the driver

All other operations (like opening a new connection, starting an I/O operation) are refused by the driver. In other words, in shutdown mode, the driver is waiting until a shutdown epilog condition is met. This allows clients to close existing connections to the driver correctly. The shutdown epilog condition is met within a leaf device driver when the device entry is released by the last driver client, and the callback release handler is invoked by the device registry.

Note - The call-back release handler is called in the DKI thread context. Therefore, the shutdown epilog is processed in the DKI thread context. This makes it possible to directly invoke the parent bus/nexus and DKI services allowed in the DKI thread context.

The table below shows typical actions taken by the shut-down epilog depending on the event type:

Action	DEV_SHUTDOWN	DEV_REMOVAL
reset hardware	+	-
release system resources	+	+
close connection to the parent driver	+	+

The `DEV_SHUTDOWN` epilog puts hardware into a clean state, releases system resources used by the driver instance (`io_unmap`, `mem_unmap`, ...) and finally, closes connection to the parent driver (close).

The DEV_REMOVAL epilog is similar to the DEV_SHUTDOWN one, except the device hardware is not touched by the driver because the device hardware is no longer present on the parent bus.

When a shutdown epilog closes the last connection to the parent bus driver, the shutdown epilog condition may be met in the parent driver too. In such a way, the shutdown epilog is propagated upstream (from child to parent).

Note - If one of the driver clients does not implement the shutdown procedure properly (for example, if it simply does not support the shutdown), the driver may be caught in shutdown mode forever. This type of driver would then never meet the shutdown epilog condition.

In the following example, the bus events management code from the NS16x50 compatible UART device driver is shown.

The NS16_DEV_REMOVAL compilation flag is used to allow downsizing of the driver component, at compile time, in case the device removal mechanism is not needed.

```
    /*
     * ns16_down_xxx stubs are used in the device removal mode in order
     * to avoid to access the device hardware.
     */
#if defined(NS16_DEV_REMOVAL)

    static void
ns16_down (UartId id)
{
}

#define ns16_down_mask          ns16_down
#define ns16_down_unmask        ns16_down
#define ns16_down_txbreak       ns16_down

    /*
     * Open device.
     */
    static int
ns16_down_open (UartId        id,
                UartConfig*   cfg,
                void*         cookie,
                UartCallBack* client_ops,
                uint32_f*     signals)
{
    return K_EFAIL;
}

    /*
     * Send a buffer.
     */
    static void
ns16_down_transmit (UartId      id,
                    uint8_f*    buffer,
                    uint32_f    count)
{
}

    static void
ns16_down_control (UartId   id,
                   uint32_f signals)
{
}

static UartDevOps ns16_down_ops =
{
    UART_VERSION_INITIAL,
    ns16_down_open,
    ns16_down_mask,
    ns16_down_unmask,
    ns16_down_transmit,
    ns16_abort,
    ns16_down_txbreak,
    ns16_down_control,
    ns16_rxbuffer,
    ns16_close
};
    /*
     * This routine aborts a pending operation (if any) on the device.
     * The driver client is notified by a call-back handler about
     * the operation failing.
     *
     * This routine is only used by the hot-plug management code
     * in order to abort an operation in progress when a hot-plug removal
     * occurs.
     */
    static void
ns16_removal (Ns16_Device* dev)
{
    if (dev->tx_csize || dev->tx_signals) {
        uint32_f count   = dev->tx_isize - dev->tx_csize;
        uint32_f signals = dev->tx_signals | UART_SIG_TX_ABORTED;
        dev->tx_cbuff    = NULL;
        dev->tx_csize    = 0;
        dev->tx_isize    = 0;
        dev->tx_signals  = 0;
        dev->clientOps->txdone(dev->cookie, count, signals);
    }
}

#endif

    /*
     * NS16550 event handler
     *
     * The event handler is invoked by the parent bus driver when a bus
     * event occurs in the bus.
     *
     * The NS16550 UART driver always supports the BUS_SYS_SHUTDOWN and
     * BUS_DEV_SHUTDOWN events. The BUS_DEV_REMOVAL support is optional and
     * is provided only when NS16_DEV_REMOVAL is defined.
     */
    static KnError
ns16_event (void*    id,
            BusEvent event,
            void*    arg)
```

```
{
    KnError      res  = K_OK;
    Ns16_Device* dev  = NS16_DEV(id);
    DevNode      node = dev->entry.dev_node;

    switch (event) {
            /*
             * In case of system (emergancy) shutdown,
             * we only disable the device interruts, in order to
             * properly perform the system reboot (or restart).
             *
             * Note that the mask() routine is invoked indirecty because
             * it may be substituted by the event handler.
             */
        case BUS_SYS_SHUTDOWN: {
            UART_OPS(dev->entry.dev_ops)->mask(id);
            break;
        }

            /*
             * The normal device shutdown is processed only from the
             * normal mode. In other words, this event is
             * ignored if the driver already operates in the device
             * shut-down or removal mode.
             *
             * Here, we just flag that the device is entered into shutdown
             * mode (dev->devEvent) and ask the device registry to
             * notify clients about it. The real shutdown procedure will
             * ne done by the ns16_release() handler. This handler is called
             * by device registry when the the reference to the driver
             * instance goes away (i.e. when svDeviceRelease() is called by
             * client).
             */
        case BUS_DEV_SHUTDOWN: {
            if (dev->devEvent == DEV_EVENT_NULL) {

                dev->devEvent = DEV_EVENT_SHUTDOWN;
                svDeviceEvent(dev->regId, DEV_EVENT_SHUTDOWN, NULL);

                DKI_MSG(("%s: entered into shut-down mode\n", dev->dpath));
            }
            break;
        }

#if defined(NS16_DEV_REMOVAL)
            /*
             * The device removal is processed from either the
             * normal mode or shutdown mode. In other words,
             * this event is ignored if the driver already operates in the
             * device removal mode.
             *
             * Here, we flag that the device is entered into removal
             * mode (dev->devEvent). In addition, the device ops are
             * substituted to empty routines in order to avoid to access
             * the hardware which has been disappeared from the bus.
             * Once ops are substituted, we ask the device registry to
             * notify clients about the device removal event.
             * The real shutdown procedure will be done by the ns16_release()
             * handler. This handler is called by device registry when the
             * the reference to the driver instance goes away (i.e. when
             * svDeviceRelease() is called by client).
             * ns16_removal() is called in order to abort a transmission
             * in progess (if any).
             *
             * Note that, receiving DEV_EVENT_REMOVAL, the driver client must
             * update pointers to the device service routines (ops) if they
             * have been previously copied by the client.
             */
        case BUS_DEV_REMOVAL: {
            if (dev->devEvent != DEV_EVENT_REMOVAL) {

                dev->devEvent = DEV_EVENT_REMOVAL;
                bcopy(&ns16_down_ops, &(dev->devOps), sizeof(ns16_down_ops));
                svDeviceEvent(dev->regId, DEV_EVENT_REMOVAL, NULL);

                ns16_removal(dev);

                DKI_MSG(("%s: entered into removal mode\n", dev->dpath));
            }
            break;
        }
#endif

        default: {
            res = K_ENOTIMP;
            break;
        }
    }

    return res;
}

    /*
     * The error handler is called by the parent bus driver
     * if a bus error occurs when accessing the device registers.
     * In the current implementation, we consider that the device
     * is not present on the bus if such an error occurs.
     * Thus, an I/O error is equivalent to the device removal event.
     */
    static void
ns16_bus_error (void*     id,
                BusError* err)
{
    DKI_ERR(("%s: error -- bus error (%d, 0x%x)\n",
             NS16_DEV(id)->dpath, err->code, err->offset));

    (void) ns16_event(id, BUS_DEV_REMOVAL, NULL);
}

    /*
     * ns16_shutdown() implements the real shutdown of a given
     * device driver instance.
     * This routine is called either by ns16_release() or ns16_unload().
     * In both cases, this routine is invoked in the DKI thread context.
     * Note that the device driver instance has been unregistered by the
     * device registry, i.e. the corresponding device registry entry is
     * invalid.
     */
    static void
ns16_shutdown (Ns16_Device* dev)
```

```
{
        /*
         * When the driver unloading is supported, we must remove
         * the driver instance from the list.
         */
#if defined(NS16_DRV_UNLOAD)
    Ns16_Device*  cdev;
    Ns16_Device** link = &ns16_devs

    while ((cdev = *link) != dev) {
        link = &(cdev->next);
    }

    *link = dev->next;
#endif
        /*
         * Release bus resources and close connection to the bus.
         */
    dev->busOps->intr_detach(dev->intrId);
    dev->busOps->io_unmap(dev->ioId);
    dev->busOps->close(dev->devId);
        /*
         * Release the device registry entry.
         */
    svDeviceFree(dev->regId);
    DKI_MSG(("%s: %s driver stopped\n", dev->dpath, NS16_DRV_NAME));
        /*
         * Finally, free memory allocated for the device descriptor.
         */
    svMemFree(dev->dpath, dev->dpathLeng);
    svMemFree(dev, sizeof(Ns16_Device));
}

    /*
     * The release handler is called by the device registry when
     * a DEV_EVENT_SHUTDOWN or DEV_EVENT_REMOVAL event has been signaled
     * (via svDeviceEvent()) and the (last) reference to the device driver
     * instance goes away.
     */
    static void
ns16_release (DevRegEntry* entry)
{
    ns16_shutdown(NS16_DEV(entry->dev_id));
}
```

CHAPTER 4

Writing Bus Drivers

This chapter takes a procedural approach to writing both host and subsequent level bus drivers.

The small differences in writing a driver for a host bus and for a sub-bus (bus to bus bridge) are outlined in steps where they occur. Also, this chapter indicates the additional steps required to provide the common bus driver API and to allow multi-bus device drivers to run on top of your bus driver.

Include the Appropriate APIs (DKI/ DDI)

The first step to building a bus driver is to include the header files for the DKI and DDI APIs involved in the bus driver's implementation.

A host bus driver implementation uses only the DKI interface, because there is no other driver component between the host bus and the microkernel API. On the other hand, a bus-to-bus bridge driver typically uses its parent bus DDI API (and some generic DKI services, like memory allocation).

In either case, the driver implementation must include:

- The parent bus class API header file(s) (DKI and/or DDI).

  Note that the services available for bus driver implementation are defined in these header files.
- The bus class API header file(s) (DDI).

Note that these header files define the routines that must be written for the driver component to be compliant with the Driver Framework.

Shown below is an example for a PCI-to-ISA bridge bus driver that uses both DKI and "PCI bus driver" APIs, and that provides both "Common bus driver" and "ISA bus driver" APIs.

```
#include <dki/dki.h>
#include <ddi/pci/pci.h>
#include <ddi/isa/isa.h>
...
#include "w83c553.h"
#include "w83c553Prop.h"
```

Register the Driver (using `main` function)

A driver component may be downloaded in various ways. It may be built into the system bootable image or it may be downloaded dynamically as a supervisor actor using the `afexec` system call. In either case, the driver code must contain the `main` routine which will be called by the system once the driver component is downloaded.

The only task of the driver's `main` routine is to perform the self-registration of the driver component within the system. To accomplish this task, the driver invokes the `svDriverRegister` microkernel call, passing as an argument a `DrvRegEntry` data structure which specifies the driver component properties.

Once the driver component is self-registered, any future driver management is mostly undertaken by its parent bus/nexus driver (or the DKI module for a host bus driver) using the properties specified in the `DrvRegEntry` structure. The `DrvRegEntry` specifies four driver's entry points as follows:

drv_probe	`drv_probe` is invoked by the parent bus/nexus driver (or the DKI module for a host bus driver) when *bus_class* specified in the registry entry matches the parent bus/nexus driver class.
drv_bind	`drv_bind` is invoked by the parent bus/nexus driver (or the DKI module for a host bus driver)

	when *bus_class* specified in the registry entry matches the parent bus/nexus driver class.
drv_init	`drv_init` is invoked by the parent bus/nexus driver (or the DKI module for a host bus driver) when *bus_class* (specified in the registry entry) matches the parent bus class and the driver is bound to a node in the device tree. (Or put more simply, when a hardware device to be managed by the driver exists.)
drv_unload	`drv_unload` is invoked by the driver registry module when an application wishes to unload the driver component from the system.

```
    /*
     * Driver registry entry for Winbond w83c553 PCI/ISA bridge
     */
static void    drv_bind   (DevNode myNode);
static void    drv_init   (DevNode myNode, void* pciOps, void* pciId);
static KnError drv_unload ();

static DrvRegEntry w83c553Drv = {
    W83C553_DRV_NAME,
    "Winbond w83c553 PCI to ISA bridge [#ident \"@(#)w83c553.c 1.5 99/02/23 SMI\"]",
    PCI_CLASS,           /* parent bus class      */
    PCI_VERSION_INITIAL, /* required bus version  */
    NULL,                /* probe method          */
    drv_bind,            /* bind method           */
    drv_init,            /* init method           */
    drv_unload           /* unload method         */
};
    /*
     * Driver main() routine.
     * Called by kernel at driver startup time.
     */
    int
main (int argc, char** argv)
{
    KnError res = svDriverRegister(&w83c553Drv);

    if (res != K_OK) {
        DKI_ERR(("%s: error -- svDriverRegister() failed (%d)\n",
                 w83c553Drv.drv_name, res));
    }
    return res;
}
```

Write Bus Driver Class-Specific Functions

At this step, you write the implementation for a specific hardware bus controller. In other words, the code has to be written for each function of the specified bus class, as defined in the API. These functions must then be provided to the subsequent level device drivers.

Note - None of these functions is directly exported, but that all of them are defined as static functions, and then grouped as indirect calls in an "operations data structure" typed by the bus class API. The bus driver then gives this "operations data structure" as an argument to its child device driver's "probe" and "initialize" registered functions (see "Write Registry Functions" on page 69, below).

In this way you can ensure that the visibility and use of the bus API is restricted to device drivers which are servicing a device connected to this bus.

Each bus class API is different. Thus, the functions to write are different for different classes of bus API. The complete list of the currently defined bus class APIs may be found in the `ddi(9)` man page.

Note - All of these steps are performed in the example provided for the "Write General Functions" section below. See specifically the sections dealing with the ISA bus.

Write General Functions

Once the bus class services are implemented in a bus driver component, you can also write the additional functions needed to provide the Common bus API services (in addition to those provided by the bus class API). This allows child device drivers to be written to the common bus-driver interface (CBDI), making them bus class independent (bottom-interface transparent).

In most cases, these services have already been implemented in the previous step. There are normally only three functions that must be added to allow the subsequent device drivers to perform independently of their bus type. These functions essentially allow the bus driver's clients to retrieve property elements from arrays.

The following code example illustrates this for a PCI-to-ISA bridge bus driver. In this example, the ISA bus class is provided, and the additional functions are written to provide the CBDI.

Note - For clarity, only the code pertinent to this explanation is shown. Please refer to a complete implementation file for more details.

```
    /*
     * ISA bus provided API
     */
    static KnError
open (IsaId isaId, ...) { ... }
    static void
close (IsaDevId devId) { ... }
    /*
     * Interrupt management
     */
    static void
mask (IsaIntrId intrId) { ... }
    static void
unmask (IsaIntrId intrId) { ... }
    static IsaIntrStatus
enable (IsaIntrId intrId) { ... }

static IsaIntrOps w83c553IntrOps = {
    mask,
    unmask,
    enable,
    unmask
};
    static KnError
intr_attach (IsaDevId devId, ...) { ... }
    static void
intr_detach (IsaIntrId intrId) { ... }
    /*
     * I/O management
     */
    static KnError
io_map (IsaDevId devId, ...) { ... }
    /*
     * Note that there is no io_unmap() method implementation.
     * The PCI bridge io_unmap() method is directly used instead.
     * This PCI method is set in the IsaBusOps at drv_init() time.
     */
    /*
     * DMA management
     */
    static KnError
dma_attach (IsaDevId devId, ...) { ... }
    static void
dma_detach (IsaDmaId dmaId) { ... }
    /*
     * Memory management
     */
    static KnError
mem_map (IsaDevId devId, ...) { ... }
    /*
     * Note that there is no mem_unmap() method implementation.
     * The PCI bridge mem_unmap() method is directly used instead.
     * This PCI method is set in the IsaBusOps at drv_init() time.
     */
    /*
     * Dynamic resource allocation
     */
    static KnError
resource_alloc (PciDevId devId, DevProperty prop) { ... }
    static void
resource_free  (PciDevId devId, DevProperty prop) { ... }

    /*
     * Common bus interface miscellaneous routines
     */
    static void*
intr_find (void* prop, int index)
{
    return ((IsaPropIntr*)prop) + index;
}
    static void*
io_regs_find (void* prop, int index)
{
    return ((IsaPropIoRegs*)prop) + index;
}
    static void*
mem_rgn_find (void* prop, int index)
{
    return ((IsaPropMemRgn*)prop) + index;
}

    /*
     * W83C553 driver initialization method
     */
    static void
drv_init (DevNode myNode, void* busOps, void* busId)
{
    PciBusOps*           pciOps = (PciBusOps*)busOps;
    PciBusOps*           pciId  = (PciId)busId;
    DevRegEntry*         w83c553Entry;
    W83c553Data*         w83c553;
    ...
        /*
         * Allocate driver instance data
         */
    w83c553 = w83c553Alloc(path, pathSize);
    if (w83c553 == NULL) {
        DKI_ERR(("%s: error -- not enough memory\n", path));
        return;
    }
    ...
        /*
         * Initialize my base level ISA bus operations
         */
    w83c553->isaOps->version        = ISA_VERSION_INITIAL;
    w83c553->isaOps->open           = open;
    w83c553->isaOps->close          = close;
    w83c553->isaOps->intr_attach    = intr_attach;
    w83c553->isaOps->intr_detach    = intr_detach;
    w83c553->isaOps->io_map         = io_map;
    w83c553->isaOps->io_unmap       = pciOps->io_unmap;  /* parent bus ops */
    w83c553->isaOps->dma_attach     = dma_attach;
    w83c553->isaOps->dma_detach     = dma_detach;
    w83c553->isaOps->mem_map        = mem_map;
    w83c553->isaOps->mem_unmap      = pciOps->mem_unmap; /* parent bus ops */
    w83c553->isaOps->resource_alloc = resource_alloc;
    w83c553->isaOps->resource_free  = resource_free;
        /*
         * Initialize my base level Common bus operations
         */
    w83c553->busOps->version        = BUS_VERSION_INITIAL;
    w83c553->busOps->open           = open;
    w83c553->busOps->close          = close;
    w83c553->busOps->intr_attach    = (KnError(*)()) intr_attach;
    w83c553->busOps->intr_detach    = intr_detach;
    w83c553->busOps->io_map         = (KnError(*)()) io_map;
    w83c553->busOps->io_unmap       = pciOps->io_unmap;  /* parent bus ops */
    w83c553->busOps->mem_map        = (KnError(*)()) mem_map;
    w83c553->busOps->mem_unmap      = pciOps->mem_unmap; /* parent bus ops */
    w83c553->busOps->intr_find      = intr_find;
    w83c553->busOps->io_regs_find   = io_regs_find;
    w83c553->busOps->mem_rgn_find   = mem_rgn_find;
    ...
        /*
         * Call children drv_probe() / drv_init() methods, if any.
         */
    childrenProbeInit(w83c553);
}
```

Write Registry Functions

Write the Probe Function

The purpose of the bus probe routine is to detect devices residing on the bus and to create device nodes corresponding to these devices. The probe routine is optional. In case it is not provided (NULL entry point), a device node should be statically created at boot time, or should be created by another "probe only" driver component to activate the bus driver.

Actions taken by a probe routine may be summarized as follows:

- The probe routine creates nodes if, and only if, they do not already exist. In other words, the probe routine is forbidden to create redundant nodes.
- The probe routine specifies a physical device ID as a device node property so that the bus driver can find the appropriate device driver for this device node.

  Note that the device ID is bus class specific. For instance, on a PCI bus, the device ID is the vendor/device IDs pair.
- The probe routine specifies resource requirements as device node properties so that the bus driver can reserve resources required to initialize the device.

There are two kinds of probe routines:

- generic (bus class specific only)

  A self-identifying bus (such as PCI) enumerator is a typical example of the generic probe routine.
- device specific (bus class and device specific)

  A device probing code on ISA bus is a typical example of the device specific probe routine.

Note that multiple probe routines for a given bus may be found in the driver registry. The Driver Framework does not specify the order in which the probe routines will be run. In addition, the probe routines may be invoked at run time (when, for example, the device insertion is detected on the bus).

When invoked at runtime, the probe routines must exercise extreme care with regard to active device nodes. Active device nodes are those for which the device drivers have been already started and may already be in use.

The following rules must be respected by generic and device specific probe routines:

- The generic and specific probe routines must access the device hardware only through the bus service routines. The bus resources needed to access the device hardware (such as I/O registers) must be allocated through the bus service routines (`resource_alloc`). This prevents the probe routine from accessing hardware which is currently in use. Upon unsuccessful probing, the hardware resources used must be released through the bus service routines (`resource_free`).
- Neither generic nor specific probe routines are allowed to delete active device nodes or to modify their properties. (Such nodes are flagged with the `active` property.)
- Device specific probe routines are allowed to override properties in an existing node or to delete existing nodes.
- Generic probe routines are not allowed to override properties in existing nodes or to delete existing nodes. In other words, device specific probe routines have a higher priority than generic ones.
- No probe routine is allowed to create redundant nodes. To run a probe routine, either you must be positive that no other node exists for this device, or you must be able to find any other nodes for this device. If for some reason it is impossible to avoid creating redundant nodes, you cannot probe.

Write the Bind Function

The bind routine enables the driver to perform a driver-to-device binding. Typical actions taken by a bind routine may be summarized as follows:

The driver examines properties attached to the device node to determine the type of device and to check whether the device may be serviced by the driver. Note that the properties examined by the driver are typically bus architecture specific. For instance, a PCI driver would examine the vendor and device identifier properties.

If the check is positive, the driver attaches a "`driver`" property to the device node. The property value specifies the driver name.

The parent bus/nexus driver should use the `"driver"` property to determine the name of the driver servicing the device. So, the child driver gives its name to the parent bus driver, through the `"driver"` property, asking the parent bus driver to invoke the `drv_init` routine on that device.

Note that, if a `"driver"` property is already present in the device node, then the `drv_bind` routine can not continue; `drv_bind` should not override an existing driver-to-device binding.

The driver-to-device binding mechanism used in the framework enables multiple implementations. A simple bind routine may be implemented by a device driver. Such an implementation would be device specific, only taking into account the devices known by the driver to be compatible with the driver's reference device.

Let us consider systems that support after-market, hot-plug devices and consult a network lookup service to locate the driver for a new device. It would be reasonable to provide a separate binder driver that would implement a smart driver-to-device mapping and a driver component download. Note that such a (generic) binder appears in the driver framework as a normal driver component. The binder driver provides the bind routine only and does not provide the probe and initialize routines.

```
  /*
     *  W83C553 driver bind method
     */
    static void
drv_bind (DevNode node)
{
    pciDevDrvBind(node, W83C553_VEND_ID, W83C553_DEV_ID, w83c553Drv.drv_name);
}
```

The `pciDevDrvBind()` function, detailed below, is implemented in the `libebd.s.a` library, not in the driver code itself.

```
  /*
     * Try to bind a given PCI driver to a given PCI device.
     * Basically, the driver is bound to the device node if
     * and only if the vendor/device ID pair specified by the
     * driver matches the vendor/device ID pair specified in
     * the device node.
     * This function is typically called by a drv_bind() method
     * of a PCI driver.
     */
    void
pciDevDrvBind (DevNode       dev_node,
               PciPropVendId drv_vid,
               PciPropDevId  drv_did,
               char*         drv_name)
{
    PciPropVendId dev_vid;
    PciPropDevId  dev_did;
    DevProperty   prop;

        /*
         * Do not bind the driver to an active device node.
         */
    if (dtreePropFind(dev_node, PROP_ACTIVE)) {
        return;
    }
        /*
         * Do not override an existing binding.
         */
    if (dtreePropFind(dev_node, PROP_DRIVER)) {
        return;
    }
        /*
         * Do not bind the driver if one of vendor/device ID's (or both)
         * is not specified.
         */
    prop = dtreePropFind(dev_node, PCI_PROP_VEND_ID);
    if (!prop) {
        return;
    }
    dev_vid = *(PciPropVendId*)dtreePropValue(prop);

    prop = dtreePropFind(dev_node, PCI_PROP_DEV_ID);
    if (!prop) {
        return;
    }
    dev_did = *(PciPropDevId*)dtreePropValue(prop);
        /*
         * Do not bind the driver if the device vendor/device IDs
         * do not match the driver ones.
         */
    if ((dev_vid != drv_vid) || (dev_did != drv_did)) {
        return;
    }
        /*
         * Bind the driver to the device...
         */
    dtreePropAdd(dev_node, PROP_DRIVER, drv_name, strlen(drv_name)+1);
}
```

Write the Init Function

The initialization routine of a bus driver component is optional. In case it is not provided (NULL entry point), the driver is typically either a probe only or bind only driver. That is, either a driver that probes a bus to discover devices and create associated device nodes, or a driver that examines device node properties to perform driver-to-device binding.

The initialization process (`drv_init`) of a bus/nexus driver mainly goes through the following steps:

1. bus/nexus device initialization
2. child device nodes creation (enumeration/probing)
3. bus resource allocation across child device nodes
4. driver-to-device binding
5. driver instances creation for child device nodes

A bus/nexus driver, like any device driver, needs to access its hardware (internal bus bridge registers).. It therefore needs to establish connection to its parent bus/nexus driver and needs to use services implemented by its parent driver.

Note - The bus/nexus driver does not need to register a new bus/nexus instance in the device registry because the standard child-to-parent driver binding mechanism does not use the device registry. The bus/nexus driver gives a pointer to its service routines vector and its identifier to the child driver, when the `drv_probe` or `drv_init` routine of the child driver is invoked.

Once the bus/nexus device is initialized, the bus/nexus driver searches drivers in the driver registry which match the given bus/nexus class and implement the `drv_probe` entry point.

The probe routine gives the child driver an opportunity to discover a device (serviceable by that driver) residing on the bus/nexus and to create the device node (associated to this device) in the device tree (see the section Write the Bind Function for an example).

A device node specifies bus resource properties required for the associated device. Note that some bus resources may be hardwired (such as fixed interrupt requests (IRQs)), while for other bus resources, some constraints may be specified (such as device address decoder constraints). When presented with configurable constraints, the bus/nexus driver iterates through existing child nodes to allocate configurable resources (with respect to constraints) and to check possible resource conflicts.

Note - If a resource conflict is detected, the bus/nexus driver behavior is implementation specific. In any case, the bus/nexus driver must not activate any driver on a node for which bus resources are not allocated successfully.

Once the bus resources allocation is done, the bus/nexus driver searches the driver registry for drivers that implement the `drv_bind` entry point and that match the given bus/nexus class. Once a driver is found, its `drv_probe` routine is invoked (once for each existing child device node). The drv_bind routine gives the child driver an opportunity to bind itsekf to the device by attaching to its name.

When the driver-to-device binding is complete, the bus/nexus driver iterates through the child nodes and, for each device node, tries to determine a driver component to apply to the given device. Once a driver component is found, its `drv_init` routine is invoked by the bus/nexus driver.

If the child device is not a leaf one, the initalization process is recursively continued by the `drv_init` routine of the child driver.

The example below presents a `drv_init` routine for a PCI-to-ISA bus driver. The W83C553_DRV_UNLOAD compilation flag is used to allow downsizing of the driver component at compile time, in case the unload mechanism is not needed.

`childrenPropbeInit()()`

```
static void
childrenProbeInit (W83c553Data* bus)
{
        /*
         * Call probe methods for both supported bus classes: pci and bus.
         */
    genBusDrvProbe(ISA_CLASS, ISA_VERSION_INITIAL,
                   bus->node, bus->isaOps, bus);
    genBusDrvProbe(BUS_CLASS, BUS_VERSION_INITIAL,
                   bus->node, bus->busOps, bus);

        /*
         * Should check for resource overlapping between devices
         * and allocate each resources through resource_alloc()
         */

        /*
         * Call bind methods for both supported bus classes: pci and bus.
         */
    genBusDrvBind(ISA_CLASS, ISA_VERSION_INITIAL, bus->node);
    genBusDrvBind(BUS_CLASS, BUS_VERSION_INITIAL, bus->node);
        /*
         * Call init methods for both supported bus classes: pci and bus.
         */
    genBusDrvInit(ISA_CLASS, ISA_VERSION_INITIAL,
                  bus->node, bus->isaOps, bus);
    genBusDrvInit(BUS_CLASS, BUS_VERSION_INITIAL,
                  bus->node, bus->busOps, bus);
}
```

The `genBusDrvProbe()`, `genBusDrvBind()` and `genBusDrvInit()` functions, detailed below, are implemented in the `libebd.s.a` library, not in the driver code itself.

```
   /*
    * A generic (standard) probing loop performed by a bus driver.
    * Go through the driver registry. Check for each entry whether
    * it matches the bus class and provides a drv_probe() method.
    * If so, apply the drv_probe() method to the bus node.
    */
    void
genBusDrvProbe (char*   bus_class,
                int     bus_version,
                DevNode bus_node,
                void*   bus_ops,
                void*   bus_id)
{
    DrvRegId     drv_curr;
    DrvRegId     drv_prev;
    DrvRegEntry* entry;

    drv_curr = svDriverLookupFirst();
    while (drv_curr) {
        entry = svDriverEntry(drv_curr);
        if (entry->drv_probe && !strcmp(bus_class, entry->bus_class) &&
            (bus_version >= entry->bus_version)) {
            entry->drv_probe(bus_node, bus_ops, bus_id);
        }
        drv_prev = drv_curr;
        drv_curr = svDriverLookupNext(drv_curr);
        svDriverRelease(drv_prev);
    }
}

   /*
    * A generic (standard) binding loop performed by a bus driver.
    * Go through the driver registry. Check for each entry whether
    * it matches the bus class and provides a drv_bind() method.
    * If so, apply the drv_bind() method to each child node
    * attached to the bus node.
    */
    void
genBusDrvBind (char*   bus_class,
               int     bus_version,
               DevNode bus_node)
{
    DevNode      dev_node;
    DrvRegId     drv_curr;
    DrvRegId     drv_prev;
    DrvRegEntry* entry;

    drv_curr = svDriverLookupFirst();
    while (drv_curr) {
        entry = svDriverEntry(drv_curr);
        if (entry->drv_bind && !strcmp(bus_class, entry->bus_class) &&
            (bus_version >= entry->bus_version)) {
            dev_node = dtreeNodeChild(bus_node);
            while (dev_node) {
                entry->drv_bind(dev_node);
                dev_node = dtreeNodePeer(dev_node);
            }
        }
        drv_prev = drv_curr;
        drv_curr = svDriverLookupNext(drv_curr);
        svDriverRelease(drv_prev);
    }
}

   /*
    * A generic (standard) initialization loop performed by a bus driver.
    * Go through the child device nodes. Check for each node whether
    * it is inactive and has a driver bound to. If so, go through the
    * driver registry. Check for each entry whether it matches the bus
    * class, provides a drv_init() method and matches the driver name.
    * If so, apply the drv_init() method to the child node. The iteration
    * through the driver registry is aborted once the device node becomes
    * active.
    */
    void
genBusDrvInit (char*   bus_class,
               int     bus_version,
               DevNode bus_node,
               void*   bus_ops,
               void*   bus_id)
{
    DevNode      dev_node;
    DrvRegId     drv_curr;
    DrvRegId     drv_prev;
    char*        drv_name;
    DrvRegEntry* entry;
    DevProperty  prop;

    dev_node = dtreeNodeChild(bus_node);
    while (dev_node) {
        if (!dtreePropFind(dev_node, PROP_ACTIVE)) {
            prop = dtreePropFind(dev_node, PROP_DRIVER);
            if (prop) {
                drv_name = (char*)dtreePropValue(prop);
                drv_curr = svDriverLookupFirst();
                while (drv_curr) {
                    entry = svDriverEntry(drv_curr);
                    if (entry->drv_init &&
                        !strcmp(bus_class, entry->bus_class) &&
                        (bus_version >= entry->bus_version) &&
                        !strcmp(drv_name, entry->drv_name)) {
                        entry->drv_init(dev_node, bus_ops, bus_id);
                        if (dtreePropFind(dev_node, PROP_ACTIVE)) {
                            svDriverRelease(drv_curr);
                            break;
                        }
                    }
                    drv_prev = drv_curr;
                    drv_curr = svDriverLookupNext(drv_curr);
                    svDriverRelease(drv_prev);
                }
            }
        }
        dev_node = dtreeNodePeer(dev_node);
    }
}
```

Write the Unload Function

drv_unload is called by the driver registry module (more precisely, by the svDriverUnregister routine) when an application wishes to unload the driver component from the system. The drv_unload routine is called in the context of the DKI thread.

This makes it possible to invoke directly the bus/nexus and DKI services allowed in the DKI thread context only. The purpose of `drv_unload` is to check that the driver component is not currently in use. On success, `drv_unload` returns `K_OK` , otherwise `K_EBUSY` is returned.

The `drv_unload` routine is optional. If `drv_unload` is not provided, the driver code cannot be unloaded. `drv_unload` is a global, per driver component routine. Therefore, to implement unloading, the driver should handle a list of driver instances. When `drv_unload` is called, the driver should go through the list, and for each driver instance, check whether the driver instance is currently in use.

Note - Once the check is positive (a given instance is not used), the driver instance must become invisible for potential clients. In other words, if `drv_unload` returns `K_OK` , all previously created driver instances (if any) must be deleted and all previously allocated system resources (if any) must be released.

If `drv_unload` returns `K_EBUSY`, the driver component will not be unloaded. In this case, the driver component state must not be changed by `drv_unload`.

The `drv_unload` routine of a bus/nexus driver typically takes the following actions:

1. Checks that the driver component is not in use.
   `drv_unload` iterates through the driver instances list and, for each driver instance, checks whether a connection is opened to the driver instance.

   Once a driver instance with an open connection is found, the iteration is aborted and `K_EBUSY` is returned. Otherwise, `drv_unload` proceeds to step 2.
2. Releases resources associated to the driver component.
   `drv_unload` iterates through the driver instances list and, for each driver instance, releases all system resources associated to the instance (`io_unmap` , `mem_unmap` , ...) and, finally, closes the connection to the parent bus. Once the iteration is finished, `drv_unload` returns `K_OK`.

Note that `drv_unload` runs in the DKI thread context.

This guarantees stability of the driver instances and open connections during the `drv_unload` execution. Indeed, a new driver instance may be created only by the `drv_init` routine and a new parent-to-child connection may be opened only by the `drv_init` or `drv_probe` routines.

Both `drv_init` and `drv_probe` are invoked in the DKI thread context. Thus, `drv_init` and `drv_probe` are serialized with `drv_unload` by the DKI thread.

On the other hand, if a bus/nexus driver supports hot-pluggable devices, it is up to the bus/nexus driver to implement a synchronization mechanism with a hot-plug insertion interrupt which may occur during the driver unloading.

In the following example, the W83C553_DRV_UNLOAD compilation flag is used to allow downsizing of the driver component, at compile time, if the unload mechanism is not needed.

```
    /*
     * Unload the W83C553 driver.
     * This routine is called by the driver registry when an application
     * wishes to unload the driver component.
     */
    static KnError
drv_unload ()
{
        /*
         * The driver unloading is an optional feature.
         * In case when such a feature is not supported, drv_unload()
         * just returns K_EBUSY preventing the driver component to be
         * unloaded.
         */
#if defined(W83C553_DRV_UNLOAD)
    W83c553Data*  udev;
        /*
         * Go through the driver instances list and check if it is unused
         * i.e. if the list of connected devices is empty.
         */
    udev = w83c553Devs;
    while (udev && (udev->dev == NULL)) {
        udev = udev->next;
    }
        /*
         * If all driver instances are unused, we invoke shutdown()
         * to shutdown each instance.
         */
    if (!udev) {
        while (w83c553Devs) {
            shutdown(w83c553Devs);
        }
        return K_OK;
    }
        /*
         * If there is a driver instance in use, we cannot unload the
         * driver, and return K_EBUSY.
         */
#endif

    return K_EBUSY;
}
```

Write Event Handler Function

The event handler is invoked by the parent bus/nexus driver when an event occurs. The event handler address is given to the parent bus/nexus driver when a connection is established between the bus driver and its parent bus/nexus driver.

The event handler may be called as an interrupt handler and therefore the event handler implementation must be restricted to the API allowed at interrupt level. Among all events which are mostly bus/nexus class specific, there are three shutdown related events (specified by the common bus API) which are discussed in this section:

SYS_SHUTDOWN	system (emergency) shutdown The `SYS_SHUTDOWN` event notifies the driver instance that the system is going to be shutdown. The parent bus/nexus driver requires the child driver instance to perform an emergency shutdown of the device hardware.
DEV_SHUTDOWN	normal device shutdown The `DEV_SHUTDOWN` event notifies the driver instance that a normal device shutdown is requested by the bus/nexus driver.
DEV_REMOVAL	surprise device removal The `DEV_REMOVAL` event notifies the driver instance that the associated device has been removed from the bus/nexus and therefore the driver instance has to be shutdown.

Note - The omitted prefix (`DKI_` , `BUS_` , ...) means that the event semantics are the same for all such events.

In general, the shutdown event processing goes through three phases:

1. shutdown prolog
2. shutdown mode
3. shutdown epilog

The first phase (called shutdown prolog) is processed synchronously within the event handler. The main purpose of the shutdown prolog is to notify child drivers that the shutdown event has occurred (to propagate the shutdown prolog downstream, from parent to child). Once the shutdown event is processed by child drivers, the driver begins the shutdown epilog. The shutdown epilog is invoked when the last connection to the driver instance is closed. Between the shutdown prolog and epilog, the driver operates in a special mode (called shutdown mode). In shutdown mode, the driver accepts only a subset of operations from child drivers allowing to close connections to the driver correctly.

The table below shows typical actions taken by the shutdown prolog depending on the event type:

Action	SYS_SHUTDOWN	DEV_SHUTDOWN	DEV_REMOVAL
notify child drivers (by calling children event handler)	+	+	+
abort operations in progress	-	-	+
reset hardware	+	-	-

The `SYS_SHUTDOWN` prolog of a bus/nexus driver invokes the event handlers of child drivers connected to it. Once the invocation is done, the bus/nexus driver puts the hardware into a clean state.

Note that the `SYS_SHUTDOWN` event is processed synchronously, (that is, within the event handler). The only purpose of the `SYS_SHUTDOWN` event is to put the board hardware into a clean state to perform the system reboot (or restart) correctly.

The `DEV_SHUTDOWN` prolog notifies child drivers that the `DEV_SHUTDOWN` event has occurred. The actual device shutdown is deferred until the `DEV_SHUTDOWN` epilog.

The `DEV_REMOVAL` prolog is closed to the `DEV_SHUTDOWN` prolog. In addition, the `DEV_REMOVAL` prolog aborts all I/O operations in progress, because otherwise these operations will never be completed. Aborted operations return an error code to callers.

As soon as the shutdown prolog is processed, the driver changes its internal state to enter into shutdown mode. In shutdown mode, the driver accepts only a subset of operations from child drivers:

- to abort queued operations
- to release previously allocated resources
- to close the connection to the driver

All other operations (like opening a new connection, or starting an I/O operation) are refused by the driver. When in shutdown mode, the driver must wait until a shutdown epilog condition is met. The shutdown epilog condition is met when the last connection to the driver's instance is closed. Because the shutdown epilog is processed in the DKI thread context, it can call all services and avoid any synchronization issues, even DKI services which can only be called in the DKI thread.

The following table shows typical actions taken by the shutdown epilog by event type.

Action	DEV_SHUTDOWN	DEV_REMOVAL
reset hardware	+	-
release system resources	+	+
close connection to the parent driver	+	+

The `DEV_SHUTDOWN` epilog puts hardware into a clean state, releases system resources used by the driver instance (`io_unmap`, `mem_unmap`, ...) and, finally, closes connection to the parent driver (`close`).

When a shutdown epilog closes the last connection to the parent driver, the shutdown epilog condition may be met in the parent driver too. In such a way, the shutdown epilog is propagated upstream (from child to parent). Note that if one of the child drivers does not shutdown properly, the driver may get lost in the shutdown mode forever, and never meet the shutdown epilog condition.

In the following example of a PCI bus events handler of a PCI-to-ISA bridge driver, the W83C553_DEV_REMOVAL compilation flag is used to allow downsizing of the driver component at compile time in case the device removal mechanism is not needed.

```
    /*
     * The event handler is invoked by the parent driver (i.e. PCI bus)
     * when an event occurs.
     *
     * The W83C553 driver always supports the PCI_SYS_SHUTDOWN and
     * PCI_DEV_SHUTDOWN events. The PCI_DEV_REMOVAL support is optional and
     * is provided only when W83C553_DEV_REMOVAL is defined.
     */
    static KnError
pciEventHandler (void* cookie, PciBusEvent event, void* arg)
{
    KnError      res     = K_OK;
    W83c553Data* w83c553 = (W83c553Data*)cookie;
    IsaDev*      isaDev;

    switch (event) {
        /*
         * Mask all ISA interrupts.
         * Then propagate the event to connected device drivers
         * and disconnect the controller from parent bus.
         */
    case PCI_SYS_SHUTDOWN: {
        w83c553->pciIntrOps->mask(w83c553->pciIntrId);
        isaDev = w83c553->dev;
        while (isaDev) {
            if (isaDev->evtHandler) {
                isaDev->evtHandler(isaDev->cookie, ISA_SYS_SHUTDOWN, arg);
            }
            isaDev = isaDev->next;
        }
        w83c553->pciConfOps->store_16(w83c553->pciConfId, PCI_COMMAND, 0);
        break;
    }
        /*
         * The normal device shutdown is processed only from the
         * normal mode. In other words, this event is
         * ignored if the driver already operates in the device
         * shut-down or removal mode.
         *
         * Here, we just flag that the device is entered into shutdown
         * mode and notify child drivers about it.
         * The real shutdown procedure will be done by the last call to
         * close().
         */
    case PCI_DEV_SHUTDOWN: {
        if (! w83c553->evtState) {
            w83c553->evtState = PCI_DEV_SHUTDOWN;
            isaDev = w83c553->dev;
     while (isaDev) {
      if (isaDev->evtHandler) {
       isaDev->evtHandler(isaDev->cookie, ISA_SYS_SHUTDOWN, arg);
      }
             isaDev = isaDev->next;
            }
            DKI_MSG(("%s: entered into shut-down mode\n", w83c553->path));
        }
        break;
    }
#if defined(W83C553_DEV_REMOVAL)
        /*
         * The device removal is processed from either the
         * normal mode or shutdown mode. In other words,
         * this event is ignored if the driver already operates in the
         * device removal mode.
         *
         * Here, we flag that the device is entered into removal
         * mode. In addition, the device ops (isa and bus) are
         * substituted to empty routines in order to avoid to access
         * the hardware which has been disappeared from the bus.
         * Once ops are substituted, we propagate the event to the
         * connected ISA drivers.
         * The real shutdown procedure will be done by the last call
         * to close().
         * removal() is called in order to abort any operation
         * in progess.
         *
         * Note that, receiving ISA_DEV_REMOVAL, the driver client must
         * update pointers to the device service routines (ops) if they
         * have been previously copied by the client.
         */
    case PCI_DEV_REMOVAL: {
        if (w83c553->evtState != PCI_DEV_REMOVAL) {
            w83c553->evtState = PCI_DEV_REMOVAL;
            w83c553->isaOps->open           = (KnError(*)())downError;
            w83c553->isaOps->intr_attach    = (KnError(*)())downError;
            w83c553->isaOps->io_map         = (KnError(*)())downError;
            w83c553->isaOps->mem_map        = (KnError(*)())downError;
            w83c553->isaOps->dma_attach     = (KnError(*)())downError;
            w83c553->isaOps->resource_alloc = (KnError(*)())downError;
            w83c553->busOps->open           = (KnError(*)())downError;
            w83c553->busOps->intr_attach    = (KnError(*)())downError;
            w83c553->busOps->io_map         = (KnError(*)())downError;
            w83c553->busOps->mem_map        = (KnError(*)())downError;
            isaDev = w83c553->dev;
            while (isaDev) {
                isaDev->evtHandler(isaDev->cookie, ISA_DEV_REMOVAL, arg);
                isaDev = isaDev->next;
            }
            removal(w83c553);
            DKI_MSG(("%s: entered into removal mode\n", w83c553->path));
        }
        break;
    }
#endif
        /*
         * W83C553 does not drive the SERR# pin (p 55)
         * and don’t care about palette snoop
         */
    default:
        res = K_ENOTIMP;
    }

    return res;
}
```

Hot-Plug Removal

The hot-plug removal event is typically reported via interrupts.

To be notified when the hot-plug removal occurs, the bus driver connects an interrupt handler to an appropriate interrupt source. Consider two typical mechanisms of hot-plug removal:

surprise removal	Surprise removal means a device can be removed at any time with no warning. For instance, PCMCIA is a surprise removal device.
non-surprise removal	Non-surprise removal means that the device cannot be removed until the system is prepared

for it. For instance, Hot-Plug CompactPCI is a non-surprise removal device.

Surprise Removal

The surprise removal interrupt notifies the bus driver that a device has been removed from the bus. The bus driver interrupt handler usually detects the removed device (and associated device node) using bus specific status register(s).

Once the device is detected, the interrupt handler checks whether the device node is active. If the device node is inactive (there is no driver instance servicing the device), the only task of the bus driver is to update the device tree removing the device node. This frees all bus resources associated with the node .

Note - The bus driver is not able to accomplish this task immediately at interrupt level because the services used are typically not available at interrupt level. These types of services can typically be called in the DKI thread context only.

To satisfy the invocation context requirements, the bus driver calls `svDkiThreadTrigger` requesting the DKI thread to invoke the removal procedure. Using the DKI thread also allows you to serialize all actions related to initialization and termination operations.

If the device node is active, the bus driver must shutdown the corresponding device driver instance prior to invoking the removal procedure. To accomplish this task, the bus driver invokes the device driver event handler signaling the `DEV_REMOVAL` event. In fact, the bus driver performs the shutdown prolog for the given driver instance (see the "Write Event Handler Function" on page 79 section). In other words, the bus driver initiates the shutdown process for the given device sub-tree. (The removed device node is the root of the sub-tree.)

As the last action in the shutdown event process, the child device driver closes the connection to the bus driver and, at this moment, the bus driver performs the removal procedure. Note that the removal procedure is executed in the DKI thread context because the `close` service routine is called in the DKI thread context.

Non-Surprise Removal

The non-surprise removal interrupt requests the bus driver to enable the device removal from the bus. This is discussed at length above (in the Surprise Removalsection). The difference between surprise and non-surprise removal is that in non-surprise removal, the bus driver requests the normal device shutdown service (`DEV_SHUTDOWN`) rather than the device removal service (`DEV_REMOVAL`).

In addition, once the device tree is updated, the bus driver enables the device removal. Device removal enabling is usually signaled by an LED, and/or by a card ejection mechanism.

Write Load Handler Function

The load handler function resides in the bus driver, and is invoked by the parent bus/nexus driver when a new driver appears in the system (as when a new driver is downloaded at run time). The load handler address is given to the bus/nexus driver when a connection is established between the child driver and its parent bus/nexus driver.

Note - The load handler is optional. Load handler functions are usually used by a bus-to-bus bridge which needs to apply a newly downloaded driver to its child devices.

The actions taken by the load handler are essentially the same as those in the initialization process, even though the bus/nexus device is already initialized:

1. child device nodes creation (enumeration/probing)
2. resource allocation for the newly created child nodes
3. driver-to-device binding
4. driver instances creation for non-active child device nodes
5. child load handlers invocation

Firstly, the bus/nexus driver invokes the probe routines registered in the driver registry matching the bus/nexus class. The probing loop example given above in "Write the Probe Function" on page 69 may be used as-is within the load handler. Note that the probe routines already invoked at initalization time will probably be invoked again.

Note that (as described in the "Write the Probe Function" on page 69 section) a probe routine *must* be aware of existing device nodes to avoid the creation of redundant nodes. In addition, a probe routine must explicitly ask for bus resources (being used for probing) to avoid conflicts with bus resources currently in use.

In this way, the active device nodes and associated running driver instances are protected against any disturbance caused by run-time probing. The probing process may create new child device nodes because a new probe routine (implemented by a newly downloaded driver) may be executed and, as a consequence, it may discover a device previously unknown in the system.

For this reason, the bus/nexus driver has to check/allocate the bus resources required for these device nodes. Note that to satisfy this run-time resource request, the bus/nexus driver may need to confiscate resources already assigned to existing device nodes.

The bus/nexus driver is not allowed to confiscate resources in use (resources assigned to active device nodes). Driver instances associated with an active node must be shutdown by the bus/nexus driver before they can be re-allocated. To shutdown a driver instance, the bus/nexus driver sends a shutdown event to the child driver, requesting closure of the child-to-parent driver connection (by invoking `close`).

Once the connection is closed, the resources are freed (and may be re-allocated). The bus/nexus driver should start the driver instance again, invoking the driver `drv_init` routine.

Once the bus resource allocation is done, the bus/nexus driver calls the drv_bind routines registered in the driver registry for each inactive device node (as explained in the "Write the Bind Function" on page 71 section).

Once the driver-to-device binding is done, the bus/nexus driver iterates through the child nodes and, for each inactive device node, determines the driver component to be applied to the given device. Once a driver component is found, the bus/nexus driver calls the driver `drv_init` routine (see section entitled "Write the Init Function" on page 72).

Finally, the bus/nexus driver invokes the child load handlers (if any) to propagate the loading process downstream (from parent to child). In this way, the loading process is recursively continued by the child driver load handler. The load handler is called in the context of DKI thread. This means that it can call all services, without worrying about synchronization, even those DKI services which can only be called in the DKI thread.

The following example shows the load handler of the PCI-to-ISA bridge driver.

```
    /*
     * The load handler is invoked by the parent driver (i.e. PCI bus)
     * when a new driver has been registered in the system (e.g. a
     * loadable driver has been downloaded).
     */
    static void
pciLoadHandler (void* cookie)
{
    W83c553Data* w83c553 = (W83c553Data*)cookie;
    IsaDev*      isaDev;
        /*
         * Perform inactive children probing / initialization to start
         * instances of the new loaded driver
         */
    childrenProbeInit(w83c553);
        /*
         * Propagate to all load handlers connected by the device drivers.
         */
    isaDev = w83c553->dev;
    while (isaDev) {
        if (isaDev->loadHandler) {
            isaDev->loadHandler(isaDev->cookie);
        }
        isaDev = isaDev->next;
    }
}
```

APPENDIX **A**

Further Information

Type of Information	Information Location
List of common DKI services available	`intro`(9DKI) man page
List of Processor-family specific DKI services available	`intro`(9DKI) man page
Details regarding one processor-family specific DKI service	Appropriate 9DKI man page
List of defined DDI APIs	`intro`(9DDI) man page
Detailed information regarding one specific DDI API	Appropriate 9DDI man page
List of existing device drivers	Intro.9drv man page
Details regarding the configuration and implementation of a driver for a particular device	Appropriate 9DRV man page
Header file(s) for a particular DDI interface class (c)	`include/chorus/ddi/c/*.h`
Header file(s) - for common DKI services - for family-specific DKI services	`include/chorus/dki/dki.h` `include/chorus/dki/f_dki.h`
Property header file for a chip driver implementation of class 'c'	`include/chorus/drv/c/chip/chipProp.h`
Implementation files for a chip device driver	`src/nucleus/bsp/drv/src/c/chip/*` `src/nucleus/bsp/family/drv_f/src/c/chip/*`

Index